CTWASM

WebAssembly, Formal Methods, and Secure Cryptography

Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, Deian Stefan

University of Cambridge; University of California, San Diego

Extended from slides by John Renner

The web has evolved

The web has evolved

Peter Sewell

Professor of Computer Science, Computer Laboratory, University of Cambridge
Member of the Cambridge Programming, Logic, and Semantics Group
Fellow of Wolfson college

Here are my contact details, a photo, short bio, and CV

PhD students, RAs, and Co-authors   Meetings   Funding   Papers (by date)   Papers (by topic)

Teaching

- The 2017-18 Part 1B *Semantics of Programming Languages* course.
- The 2017-18 *Multicore Semantics and Programming (R204)* ACS MPhil module
- ...previous teaching

http://www.cl.cam.ac.uk/ ~ pes20/

The web has evolved

Peter Sewell

Professor of Computer Science, Computer Laboratory, University of Cambridge
Member of the Cambridge Programming, Logic, and Semantics Group
Fellow of Wolfson college

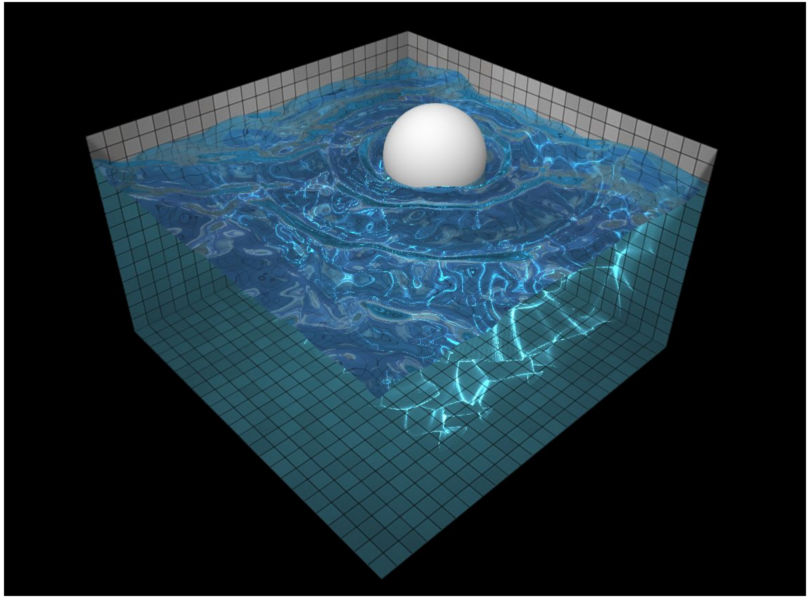
Here are my contact details, a photo, short bio, and CV

PhD students, RAs, and Co-authors   Meetings   Funding   Papers (by date)   Papers (by topic)

Teaching

- The 2017-18 Part 1B *Semantics of Programming Languages* course.
- The 2017-18 *Multicore Semantics and Programming (R204)* ACS MPhil module
- ...previous teaching

http://www.cl.cam.ac.uk/ ~ pes20/



https://github.com/evanw/webgl-water

The web has evolved

Peter Sewell

Professor of Computer Science, Computer Laboratory, University of Cambridge
Member of the Cambridge Programming, Logic, and Semantics Group
Fellow of Wolfson college

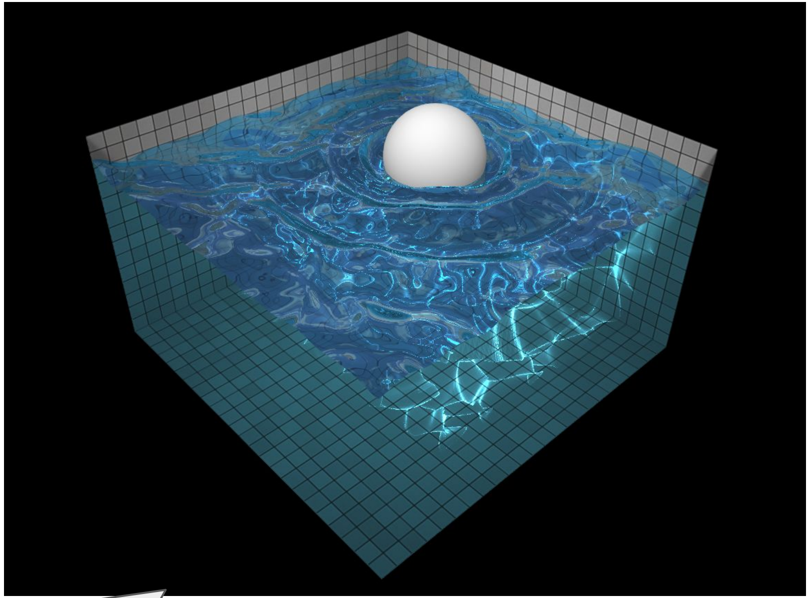
Here are my contact details, a photo, short bio, and CV

PhD students, RAs, and Co-authors Meetings Funding Papers (by date) Papers (by topic)

Teaching

- The 2017-18 Part 1B *Semantics of Programming Languages* course.
- The 2017-18 *Multicore Semantics and Programming (R204)* ACS MPhil module
- ...previous teaching

http://www.cl.cam.ac.uk/ ~ pes20/



https://github.com/evanw/webgl-water

The web has evolved

- We want richer web apps -
  3D rendering, physics, 60 FPS.
- asm.js exists but is limited by
  being built on top of JavaScript.
- We're at the limits of JavaScript -
  need a purpose-built language.

The web has evolved

- A web-friendly bytecode.
- Runs on any browser.
- “Near-native” performance.
- Targetted by LLVM.
- Formally specified!

  Bringing the web up to speed with WebAssembly [Haas et al. 2017]

What does WebAssembly look like?

```
(module
  (func $add (param i32 i32) (result i32)
    (local.get 0)
    (local.get 1)
    (i32.add)
    (return))
  (export "add_ints" (func $add))
```

What does WebAssembly look like?

```
(module
  (func $add (param i32 i32) (result i32)
    (local.get 0)
    (local.get 1)
    (i32.add)
    (return))
  (export "add_ints" (func $add)))
```

What does WebAssembly look like?

```
(module
  (func $add (param i32 i32) (result i32)
    (local.get 0)
    (local.get 1)
    (i32.add)
    (return))
  (export "add_ints" (func $add)))
```

What does WebAssembly look like?

```
(module
  (func $add (param i32 i32) (result i32)
    (local.get 0)
    (local.get 1)
    (i32.add)
    (return))
  (export "add_ints" (func $add)))
```

What does WebAssembly look like?

```
(module
  (func $add (param i32 i32) (result i32)
    (local.get 0)
    (local.get 1)
    (i32.add)
    (return))
  (export "add_ints" (func $add)))
```

WebAssembly execution, formally

$$F; e^* \hookrightarrow F'; e'^*$$

$$t \triangleq \texttt{i32} \mid \texttt{i64} \mid \texttt{f32} \mid \texttt{f64}$$

$$\begin{aligned} e \triangleq\ & t.\texttt{const}\ n \mid t.\texttt{add} \mid t.\texttt{load} \mid t.\texttt{store} \\ & \mid \texttt{local.get}\ n \mid \texttt{local.set}\ n \mid \ldots \end{aligned}$$

Addition

```
(i32.const 4)
(i32.const 2)
(i32.const 1)
(i32.add)
(i32.add)
```

↪

```
(i32.const 4)
(i32.const 3)
(i32.add)
```

↪

```
(i32.const 7)
```

Load

$$F;\ \begin{array}{l}(\texttt{i32.const}\ k)\\ (t.\texttt{load})\end{array} \hookrightarrow F;\ (t.\texttt{const}\ v)$$

if

$$\begin{array}{l}F = \{\\ \quad \dots\\ \quad \text{memory} = mem\\ \}\end{array}$$

and

$$v = \text{load}(mem,\ k,\ t)$$

Store

$$F;\ \begin{array}{l}(\texttt{i32.const}\ k)\\(t\texttt{.const}\ v)\\(t\texttt{.store})\end{array} \hookrightarrow F';\ .$$

if

$$F = \{ \begin{array}{l} \dots \\ \text{memory} = \mathit{mem} \end{array} \}$$

and

$$F' = \{ \begin{array}{l} \dots \\ \text{memory} = \\ \quad \text{store}(\mathit{mem}, k, v) \end{array} \}$$

Get local

$$F;\ (\mathtt{local.get}\ k) \hookrightarrow F;\ (t\mathtt{.const}\ v)$$

if

$$F = \{$$

$$\quad \ldots$$

$$\quad \mathrm{local}[k] = (\ v :: t\ )$$

$$\}$$

Set local

$$F;\ \begin{matrix}(t.\mathtt{const}\ v)\\(\mathtt{local.set}\ k)\end{matrix} \hookrightarrow F';\ .$$

if

$$F = \{\ \begin{matrix}\ldots\\ \mathrm{local}[k] = \ldots\end{matrix}\ \}$$

and

$$F' = \{\ \begin{matrix}\ldots\\ \mathrm{local}[k] = (\ v :: t\ )\end{matrix}\ \}$$

Structured control flow

- WebAssembly has no “goto” operation
- Instead, directly encode control flow constructs
- Only structured control flow is allowed

$$\begin{aligned} e \triangleq \ldots\ & \mathtt{block}\ \mathit{tf}\ e^*\ \mathtt{end} \\ \mid\ & \mathtt{loop}\ \mathit{tf}\ e^*\ \mathtt{end} \\ \mid\ & \mathtt{if}\ \mathit{tf}\ e^*\ \mathtt{else}\ e^*\ \mathtt{end} \end{aligned}$$

Type system

All WebAssembly programs must be validated (type checked) before execution.

$$tf \triangleq (t^* \rightarrow t^*)$$

`(i32.const 4)`

Type:

$([] \rightarrow [\mathtt{i32}])$

Type system

All WebAssembly programs must be validated (type checked) before execution.

$tf \triangleq (t^* \to t^*)$

```
(i32.const 4)
```

Type:

([] → [i32])

```
(i32.add)
(i32.add)
```

Type:

([i32,i32,i32] → [i32])

Type system

All WebAssembly programs must be validated (type checked) before execution.

$tf \triangleq (t^* \to t^*)$

```
(i32.const 4)
```

Type:

([] → [i32])

```
(i32.add)
(i32.add)
```

Type:

([i32, i32, i32] → [i32])

```
(f64.const 2)
(i32.const 1)
(i32.add)
```

Ill-typed

Type soundness

- Preservation
    - If a program (F; e*) is validated with a type ([] → [t*]), any program obtained by reducing (F; e*) to (F'; e'*) can also be validated with type ([] → [t*]).

- Progress
    - For any validated program (F; e*) that has not terminated with a result, there exists (F'; e'*) such that (F; e*) reduces to (F'; e'*).

A Syntactic Approach to Type Soundness [Wright and Felleisen. 1994]

Mechanisation

- An unambiguous formal specification and correctness condition.

- Perfect for mechanisation!

- ~11,000 lines of Isabelle/HOL definitions and proofs.

Mechanisation

- Found several errors in the draft specification.
- With fixes, proof complete!
- Also included: verified interpreter and type-checker

Mechanisation

Two categories of error were found.

- Trivial “syntactic” errors
    - typos
    - missing cases

- Deeper “semantic” errors
    - Edge-cases where formal rules get stuck
    - Sound interop with JavaScript/host

Mechanisation

Two categories of error were found.

- Trivial “syntactic” errors
  - simply copying definitions down
  - don’t need a full prover

- Deeper “semantic” errors
  - Discovered during soundness proof
  - Hard to find by hand

CTWASM

Writing crypto code is hard

Functional correctness is not enough

`enc(m,prv1)` 25ns

`enc(m,prv2)` 20ns

Timing discrepancies leak information about inputs

Remote Timing Attacks are Practical [Brumley et al. 2003]

Remote Timing Attacks are Still Practical [Brumley et al. 2011]

Hey, You, Get Off of My Cloud [Ristenpart et al. 2009]

Lucky Thirteen [Farden et al. 2013]

Lucky Thirteen Strikes Back [Irazoqui et al. 2015]

JavaScript Runtime

code.js

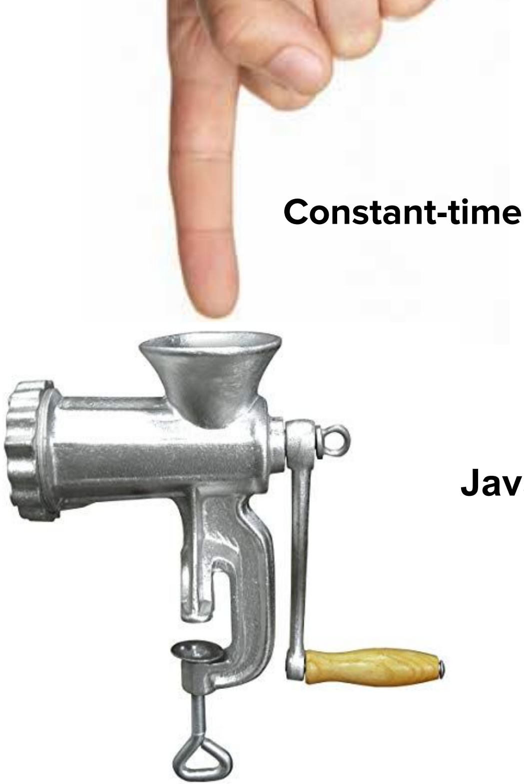
JavaScript Runtime

`code.js`

Fast machine code

```
10101001  11111111  10001101  10000100
00000011  10101001  11001000  10001101
10001000  00000011  10101001  00000000
10001101  10000110  00000011  10001101
10001010  00000011  10001101  10001011
```

JavaScript Runtime



Constant-time code

Fast machine code

```
10101001  11111111  10001101  10000100
00000011  10101001  11001000  10001101
10001000  00000011  10101001  00000000
10001101  10000110  00000011  10001101
10001010  00000011  10001101  10001011
```

JavaScript Runtime

Signal

crypto-js

JavaScript library of crypto standards.

444,906

Existing JS Crypto Solutions

Platform Crypto (WebCrypto, node.js crypto)

- **Missing modern algorithms (Poly1305)**
- **Unreliable support**

Native C Modules

- **Doesn’t work in browsers**

So what can we do?

WA

WEBASSEMBLY

Statically Typed

Low-level

Portable

WebAssembly is not enough

Doesn’t stop you from writing leaky code

Runtime can still introduce vulnerabilities

Solution:
Make Secrecy Explicit

Secret Types

`s32`
`s64`

All other types are public:
`i32, i64, f32, f64`

Turn vulnerabilities into type errors

Inform the runtime

Key Insights

Observe “best-practice” cryptography code

https://cryptocoding.net/index.php/Coding_rules

Restrictions are course-grained - simple type system!

Prevent Explicit Leaks

Prevent Implicit Leaks

Prevent Leaks via Timing

Direct leakage as type errors

```
(local $pub i32)
(local $sec s32)

(local.set $pub (get_local $sec))
```

Preventing explicit leaks

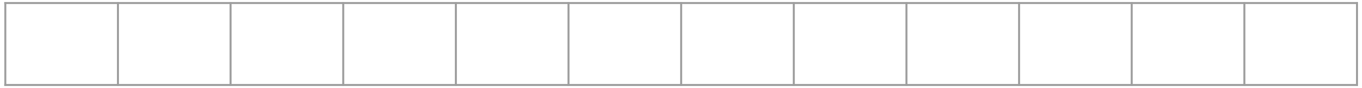
```
(local $pub i32)
(local $sec s32)

(local.set $pub (get_local $sec))
```

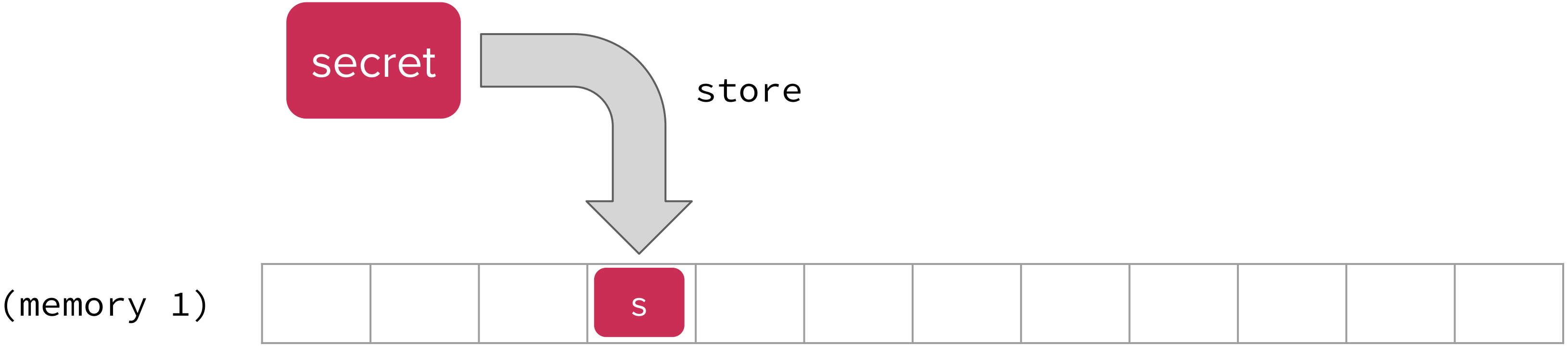
```
Error: type mismatch in set_local,
       expected i32 but got s32
```

What about memory?

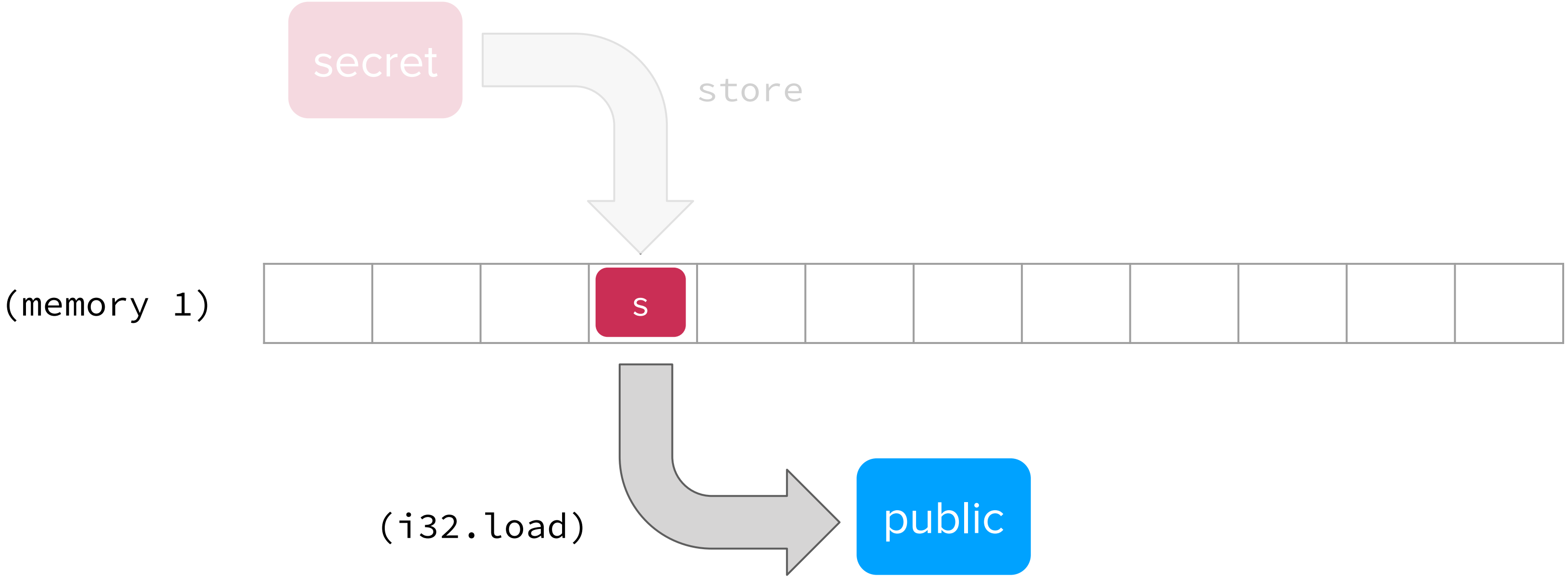
`(memory 1)`



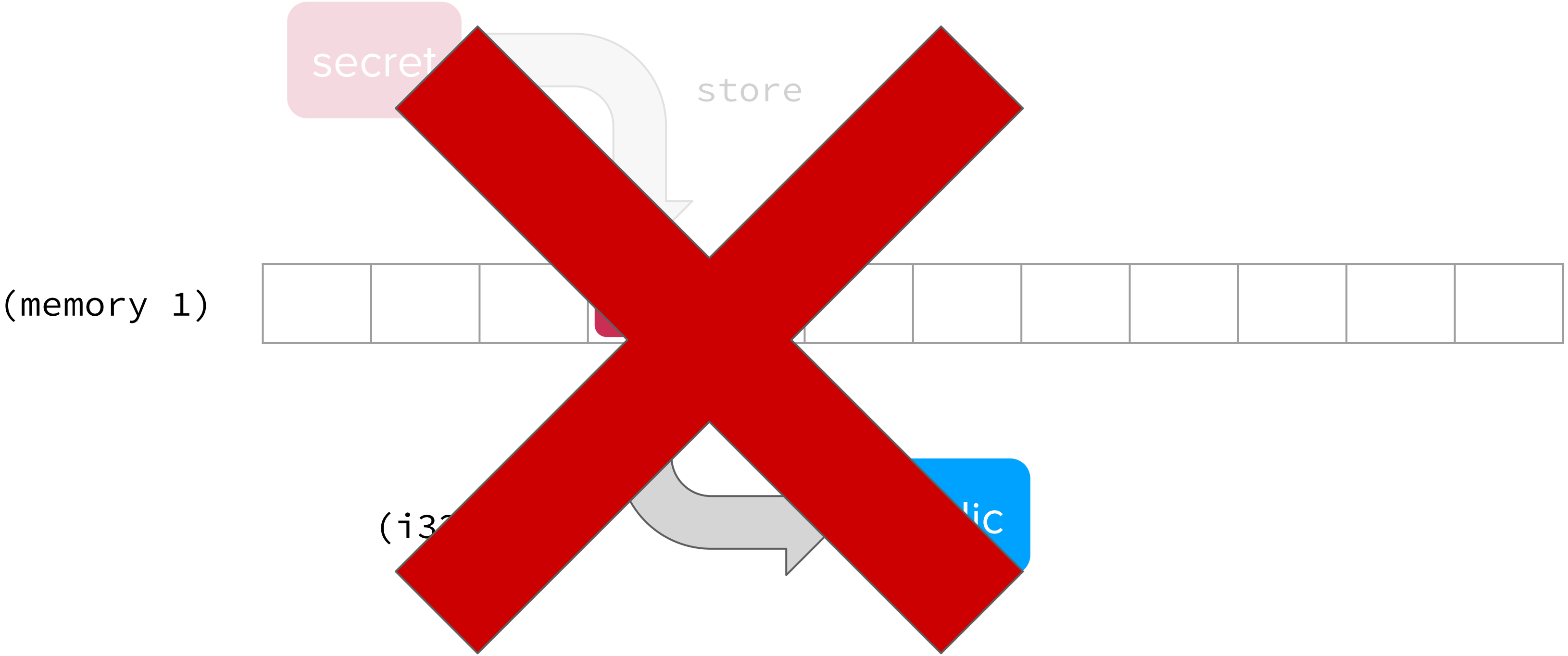
What about memory?



What about memory?



What about memory?



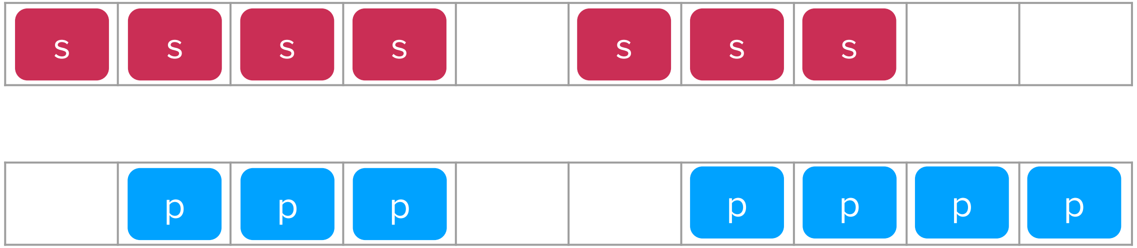
Secret Memory

```
(memory secret 1)
```

Secret opcodes

```
s32.load
s32.store
...
```

Secret Memory



`(memory secret 1)`	s	s	s	s		s	s	s		
`(memory 1)`		p	p	p			p	p	p	p

Prevent Explicit Leaks

Prevent Implicit Leaks

Prevent Leaks via Timing

Preventing leaks via control flow

```
(if (local.get $sec)

(then

    (local.get $pub ...))

(else

    (local.get $pub ...)))
```

Leaks:

- Indirect flow
- Timing (Conditional jump)

Preventing implicit leaks

```
(if (local.get $sec)
(then
    (local.get $pub ...))
(else
    (local.get $pub ...)))
```

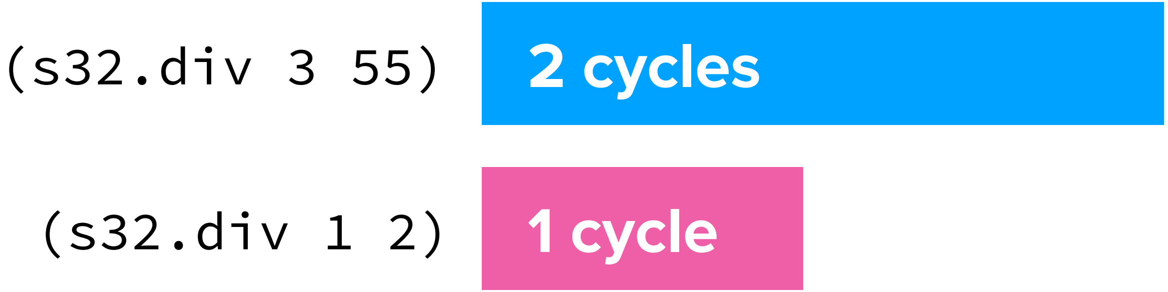
```
TypeError
‘if’ requires i32
     found    s32
```

Prevent Explicit Leaks

Prevent Implicit Leaks

Prevent Leaks via Timing

Certain instructions are leaky



`(s32.div 3 55)` **2 cycles**

`(s32.div 1 2)` **1 cycle**

Some operations are constant-time:
`add, xor, sub, mul, …`

Others are not:
`div, rem`

Floating point arithmetic

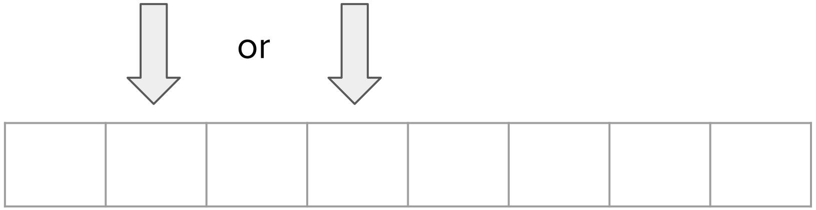
Preventing cache timing attacks

All memory operations both mutate and leak cache state

Cache state must be independent of secrets

Disallow secrets as memory indices

`(s32.load (get_local $sec))`



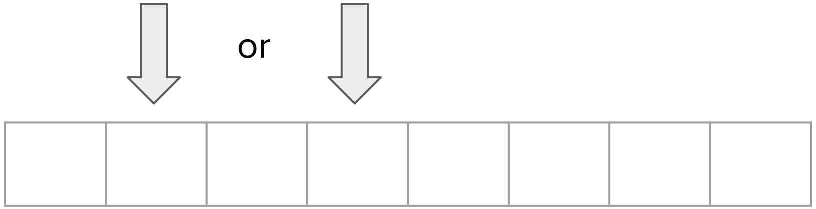
Preventing cache timing attacks

All memory operations both mutate and leak cache state

Cache state must be independent of secrets

Disallow secrets as memory indices

```
(s32.load (get_local $sec))
```



```
TypeError
's32.load' requires i32
           found    s32
```

Too safe for our own good

secret

encrypt (secret)

encrypt (secret) → public

encrypt (secret) → public

```
i32.declassify :: s32 → i32
```

Limiting `declassify` to trusted code

New Function Type

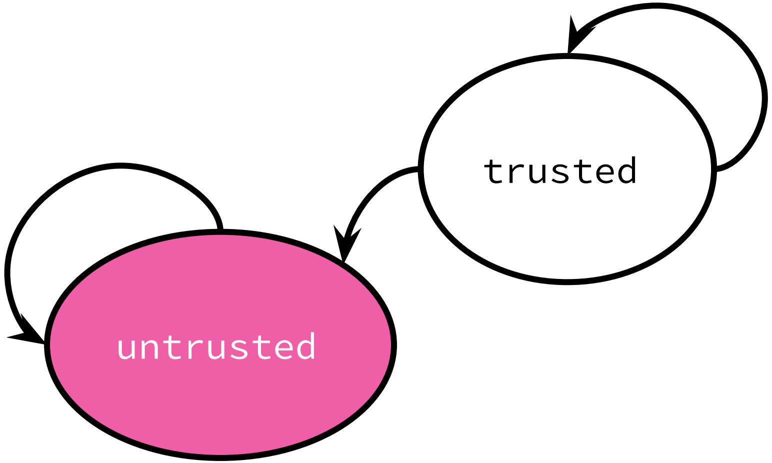
`(func untrusted ...)`

Can't `declassify`

Can only call other `untrusted` functions

Most crypto code is `untrusted`

Untrusted functions can’t `declassify`



Secure linking with untrusted

```
(import "crypto_lib" “handle_secret"
   (func untrusted (param s32)))
```

Explicit import types assert trust

Typechecker ensures library can’t leak

Verified in Isabelle

- Build on top of existing mechanisation
- ~5,800 lines of alterations/insertions
- Non-interference, constant-time

Non-Interference

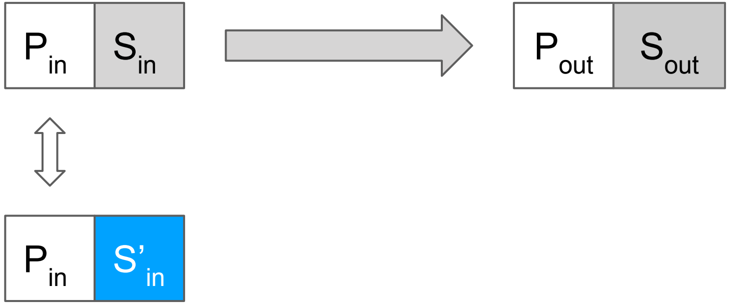
A computation's public outputs
are independent from secret inputs

Non-Interference

A computation’s public outputs
are independent from secret inputs

Non-Interference

A computation's public outputs
are independent from secret inputs



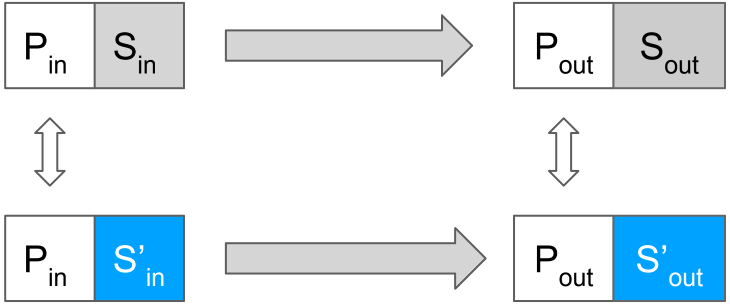
P_{in} S_{in}

P_{out} S_{out}

P_{in} S'_{in}

Non-Interference

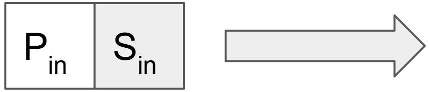
A computation's public outputs
are independent from secret inputs



Constant-Time

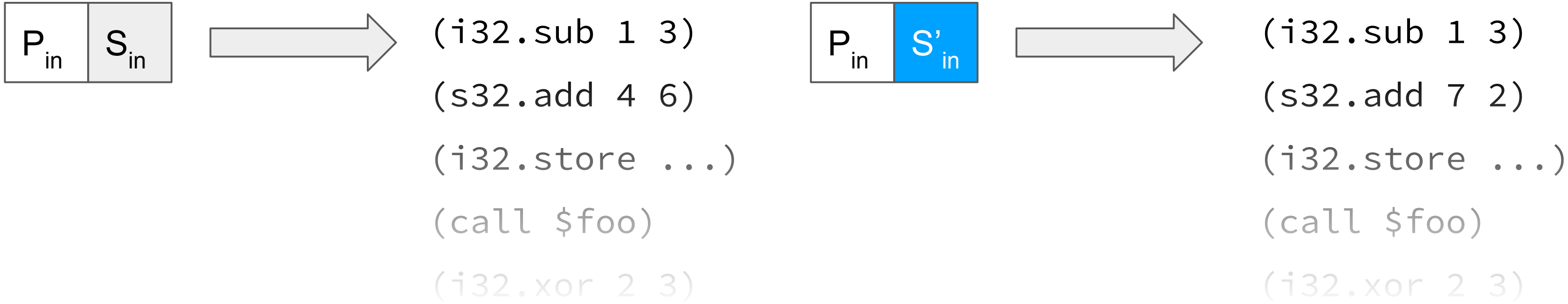
A program’s leakage
is independent from secret inputs

Constant-Time


P_{in} S_{in}

```
(i32.sub 1 3)
(s32.add 4 6)
(i32.store ...)
(call $foo)
(i32.xor 2 3)
```

Constant-Time



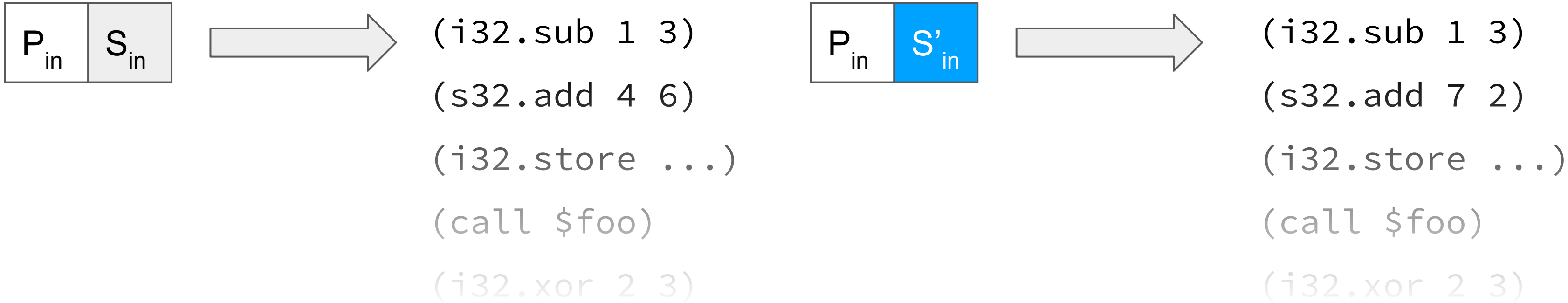
P_{in} S_{in}

```
(i32.sub 1 3)
(s32.add 4 6)
(i32.store ...)
(call $foo)
(i32.xor 2 3)
```

P_{in} S'_{in}

```
(i32.sub 1 3)
(s32.add 7 2)
(i32.store ...)
(call $foo)
(i32.xor 2 3)
```

Constant-Time



P_{in} S_{in}

```
(i32.sub 1 3)
(s32.add 4 6)
(i32.store ...)
(call $foo)
(i32.xor 2 3)
```

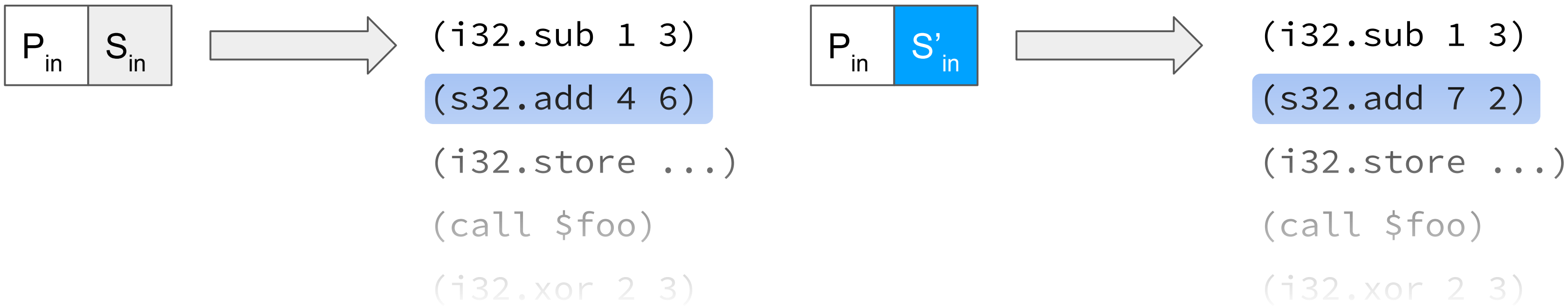
P_{in} S'_{in}

```
(i32.sub 1 3)
(s32.add 7 2)
(i32.store ...)
(call $foo)
(i32.xor 2 3)
```

Recall: No branching on secrets

Program traces only differ on secret values

Constant-Time



P_{in} S_{in}

```
(i32.sub 1 3)
(s32.add 4 6)
(i32.store ...)
(call $foo)
(i32.xor 2 3)
```

P_{in} S'_{in}

```
(i32.sub 1 3)
(s32.add 7 2)
(i32.store ...)
(call $foo)
(i32.xor 2 3)
```

Recall: No branching on secrets

Program traces only differ on secret values

Constant-time Proof

P_{in} S_{in}

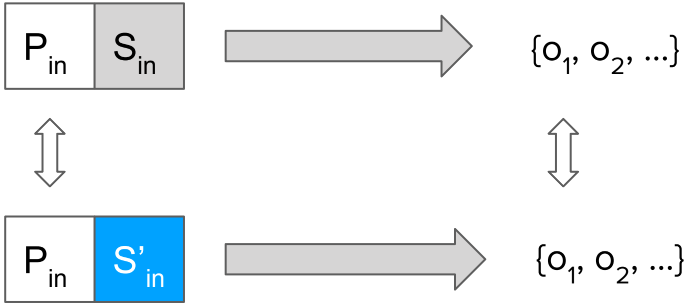
$\{o_1, o_2, \ldots\}$

Constant-time Proof

Observations:

- Branch conditions
- Memory access patterns
- Non-CT operands

Constant-time Proof



Observations:

- Branch conditions
- Memory access patterns
- Non-CT operands

Does it work in practice?

Implementations

Reference Interpreter	**V8**
Written in OCaml	Written in C++
Extended test suite	Production-quality runtime
Verified typechecker	Empirically checked

Evaluation

Performance & Security

TEA

SHA-256

Salsa20

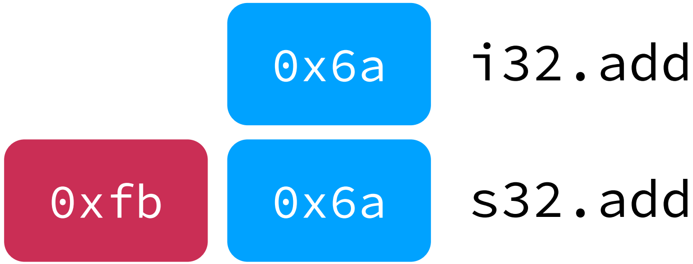
TweetNaCl

Evaluation

Performance & Security

- TEA
- SHA-256
- Salsa20
- **TweetNaCl**
    - X25519, Poly1305, XSalsa20
    - SHA-512, Ed25519

Binary size overhead



~15% overhead in practice

0% overhead for vanilla Wasm code

Performance overhead

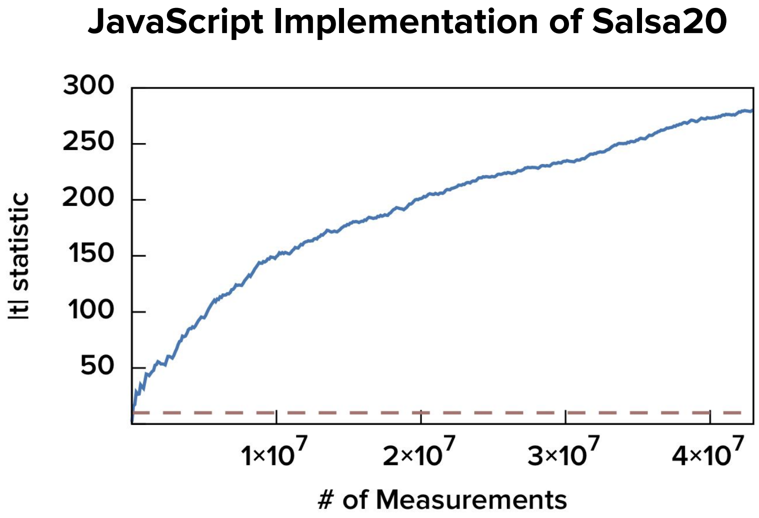
Runtime: < 1 %

Typechecking (Wasm): 14%

Typechecking (CT-Wasm): 20%

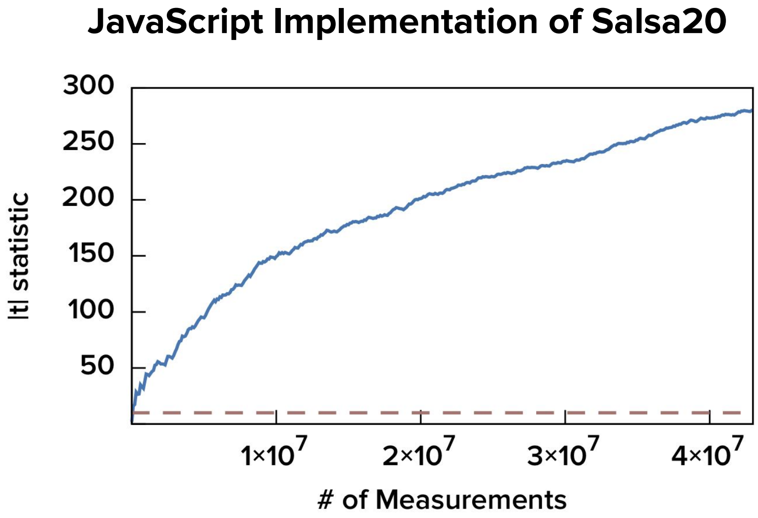
In practice: submillisecond validation of TweetNaCl

Statistical analysis with dudect



Dude, is my code constant time? [Reparaz et al. 2017]

Statistical analysis with dudect

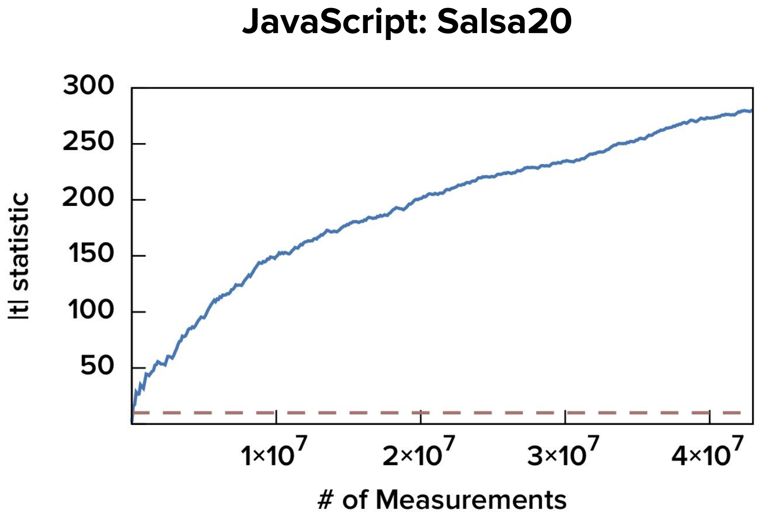


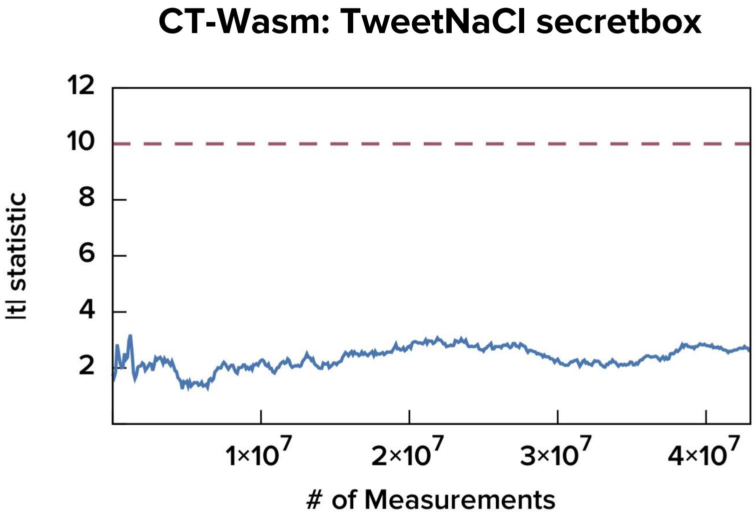
Why?

- Algorithm encodes sequences of 32-bit numbers
- V8 boxes >31-bit numbers
- Not obvious!

Dude, is my code constant time? [Reparaz et al. 2017]

Statistical analysis with dudect





Dude, is my code constant time? [Reparaz et al. 2017]

But can I use CT-Wasm today?

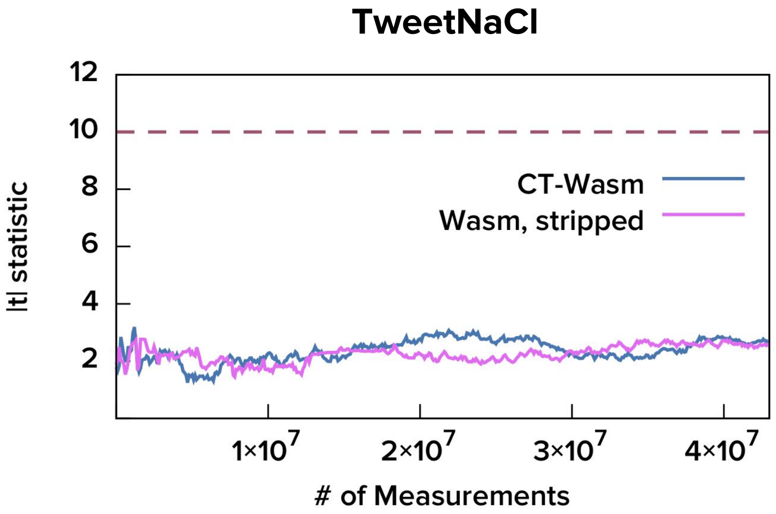
`ct2wasm`

Convert typechecked CT-Wasm to standard Wasm

Guarantees of constant-time structure

Best effort safety from runtime

`select` transform



Going Forward

Verified Compilation

CT-Wasm as a crypto IR

Label Inference

WA

CTWASM

Type system for secure crypto

Formally verified in Isabelle

Performant implementation in V8

Usable today through `ct2wasm`