Language-Independent Sandboxing of Just-In-Time Compilation and Self-Modifying Code

Jason Ansel
MIT

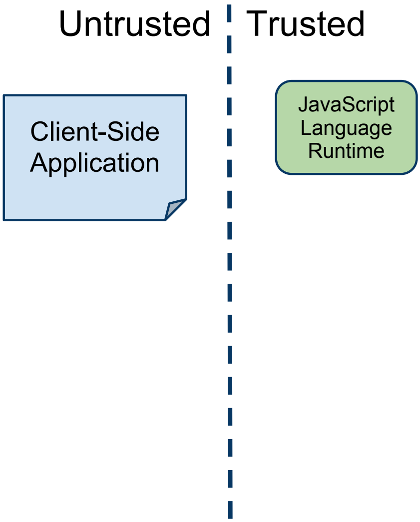
Petr Marchenko
University College London

Úlfar Erlingsson Elijah Taylor Brad Chen Derek Schuff
David Sehr Cliff Biffle Bennet Yee
Google, Inc.

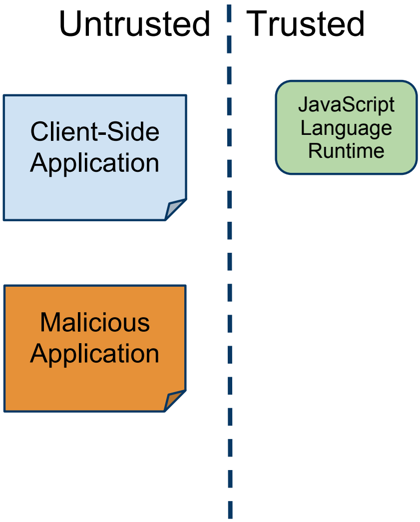
June 7, 2011

Outline

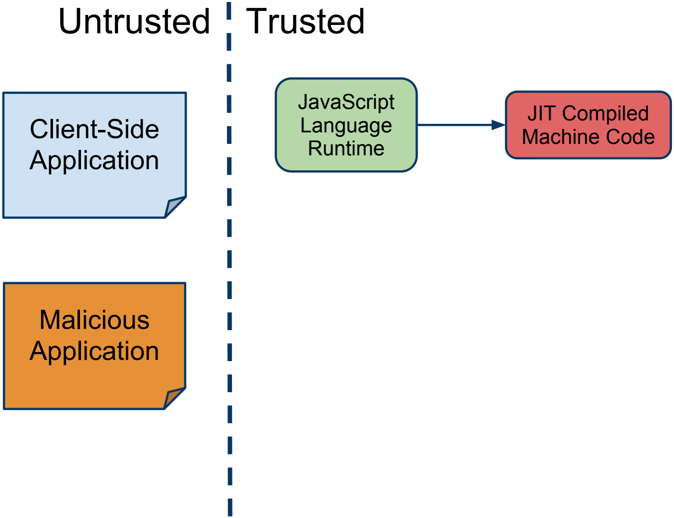
Web browser security model



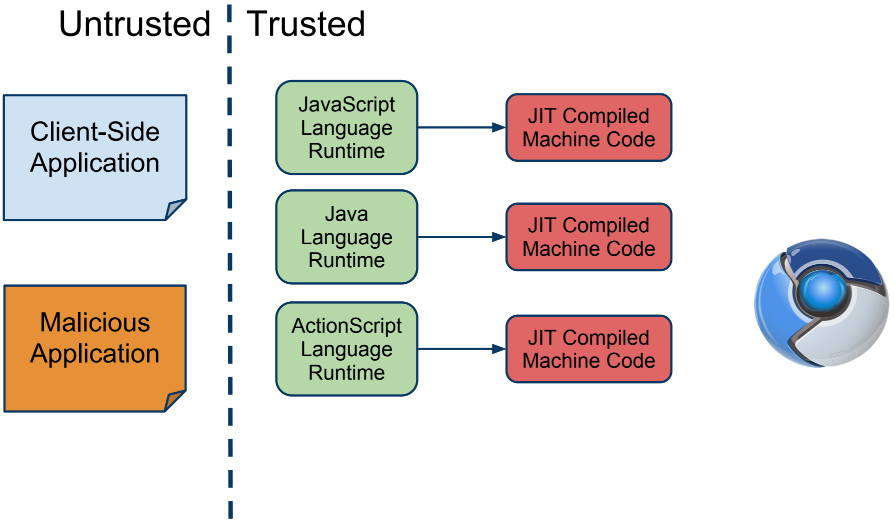
Web browser security model



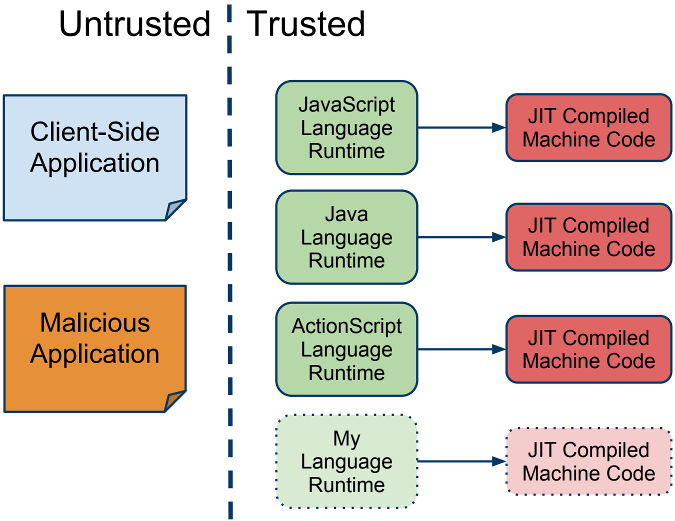
Web browser security model



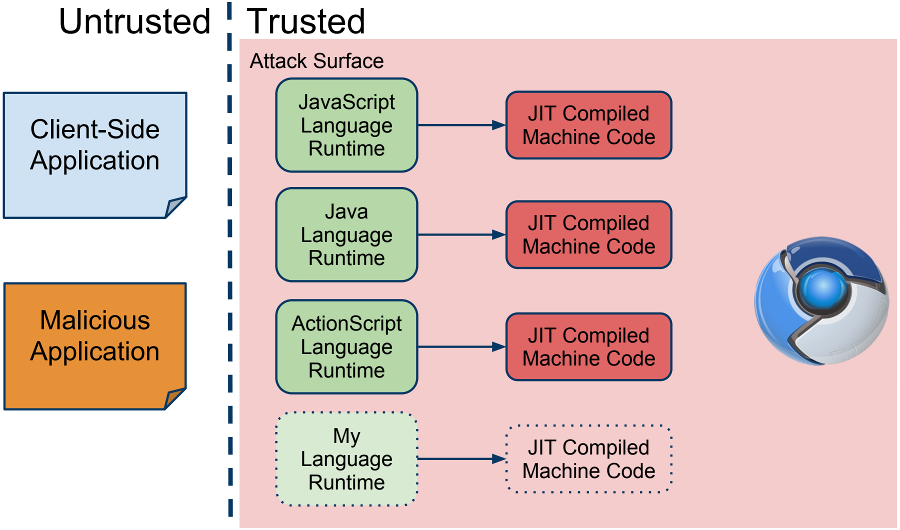
Web browser security model



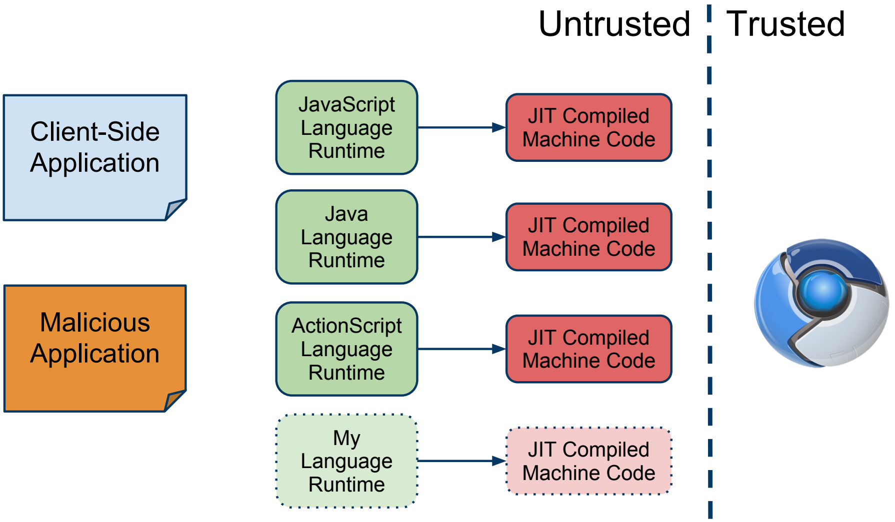
Web browser security model



Web browser security model



Web browser security model



Can we create better trust model?

- Sandbox untrusted language run-times
- Or, more generally, sandbox applications that:
  - Dynamically generate code
  - Modify the generated code (e.g. inline caches)
  - Use many threads
  - Garbage collected
  - Include large native libraries
- While maintaining performance
- Easy to verify correctness of the sandboxing

Sandboxing technology

- Software Fault Isolation (SFI)[1]
  - OS-portable
  - Low overhead
  - Fast to enter/exit
  - Easy to reason about correctness
  - Traditionally does not allow dynamic code modification

[1]Wahbe *et. al., 1993*

Sandboxing technology

- Software Fault Isolation (SFI)[1]
  - OS-portable
  - Low overhead
  - Fast to enter/exit
  - Easy to reason about correctness
  - Traditionally does not allow dynamic code modification
- We extend the Native Client SFI system to support self-modifying code

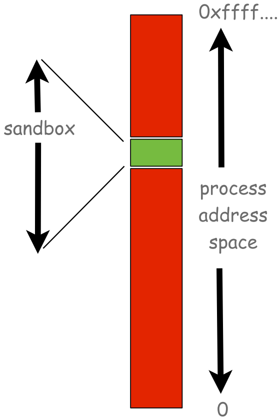
[1] Wahbe *et. al., 1993*

Outline

1. Motivation
2. Native Client background
3. Dynamic code modification
4. Experimental results
5. Conclusions

Software Fault Isolation (SFI) background

- Entire program checked once for safety at startup



Software Fault Isolation (SFI) background

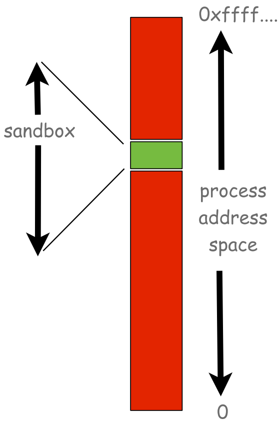
- Entire program checked once for safety at startup

Control safety

- Control cannot leave untrusted address space
- (Except through moderated interfaces)
- Only known instructions in the untrusted address space can execute

Data safety

- Writes can only change untrusted memory



Control safety (Native Client background)

- Must confine execution to instructions that have been checked
  - Prevent execution of “hidden” instructions
  - e.g., instructions overlapping at a different offset
    - Disassemble bytes 0 to 6: **81 c3 cd 80 eb 66**

      ```
      add $0x66eb80cd, %ebx
      ```

    - Disassemble bytes 2 to 6: `81 c3` **cd 80 eb 66**

      ```
      int     $0x80
      jmp     0x40052c
      ```

Control safety (Native Client background)

- Must confine execution to instructions that have been checked
  - Prevent execution of "hidden" instructions
  - e.g., instructions overlapping at a different offset
    - Disassemble bytes 0 to 6: **81 c3 cd 80 eb 66**

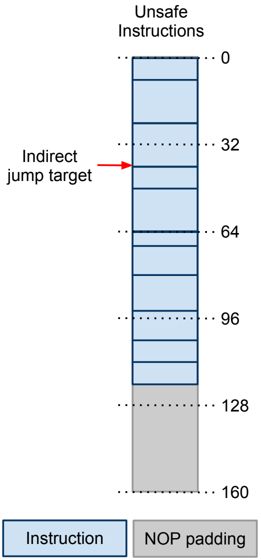
      ```
      add $0x66eb80cd, %ebx
      ```

    - Disassemble bytes 2 to 6: `81 c3` **cd 80 eb 66**

      ```
      int     $0x80
      jmp     0x40052c
      ```

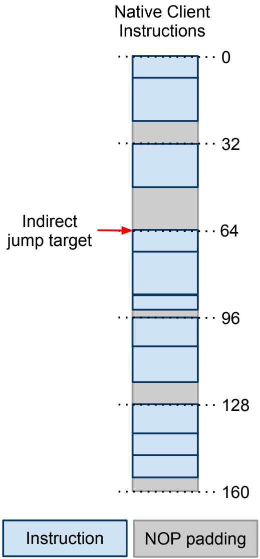
- Direct jumps
  - Can be checked statically
- Indirect jumps
  - More difficult
  - Restricted, requiring guard sequence

Instruction bundles (Native Client background)



- All 32-byte aligned addresses in code region must be safe jump targets
  - "Bundles"
  - Instructions and guard sequences can not cross bundles
  - NOP padding often required
- Indirect control flow must use guard sequence
  - Masks away lower bits
  - Forces indirect jump to go to start of a bundle

Instruction bundles (Native Client background)



- All 32-byte aligned addresses in code region must be safe jump targets
  - "Bundles"
  - Instructions and guard sequences can not cross bundles
  - NOP padding often required
- Indirect control flow must use guard sequence
  - Masks away lower bits
  - Forces indirect jump to go to start of a bundle

Native Client summary

- Data safety provided in a similar way
  - Guards and some hardware support
- Native Client enforces a small set of local constraints
- These constraints are:
  - Efficient to verify
  - Easy to reason about
- Technique does not extend directly to self-modifying code

Outline

Challenges for dynamic code

- Must incrementally validate new code
- Must incrementally validate code modification
- Must support deletion of code (or `eval` would leak memory)
- Be safe in the presence of untrusted threads:
  - Memory consistency model for instructions is weaker than for data (on x86)
  - Consistency guarantees vary between processors

New Native Client interfaces

Create Dynamic Code

```
int nacl_dyncode_create(void* target,
                        void* src,
                        size_t size);
```

Modify Dynamic Code

```
int nacl_dyncode_modify(void* target,
                        void* src,
                        size_t size);
```

Delete Dynamic Code

```
int nacl_dyncode_delete(void* target,
                        size_t size);
```

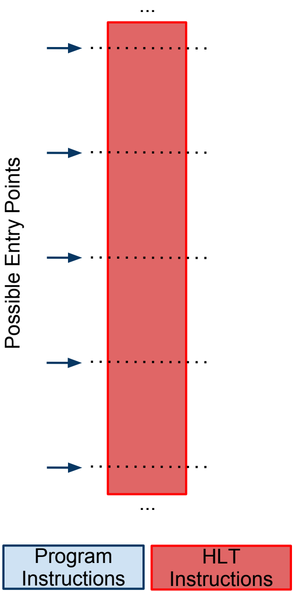
Dynamic code regions

- Dynamic code region: a block of code inserted and deleted as a unit
- Operate on entire regions:
  - `nacl_dyncode_create`
  - `nacl_dyncode_delete`
- Operates on instruction(s) inside a region:
  - `nacl_dyncode_modify`

Dynamic code regions

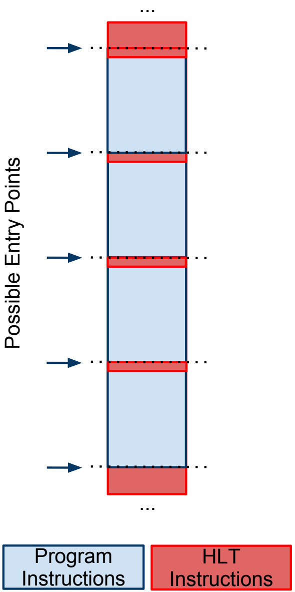
- Dynamic code region: a block of code inserted and deleted as a unit
- Operate on entire regions:
  - `nacl_dyncode_create`
  - `nacl_dyncode_delete`
- Operates on instruction(s) inside a region:
  - `nacl_dyncode_modify`
- Unaligned direct jumps only allowed within dynamic regions
  - External entry points at bundle boundaries

Lifecycle of a dynamic region



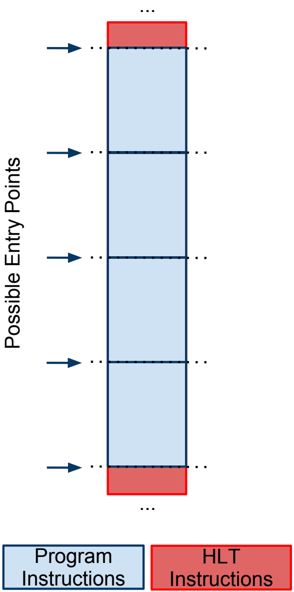
- nacl_dyncode_create
  - Validates new code
  - Two-phase update so that change appears atomic

Lifecycle of a dynamic region



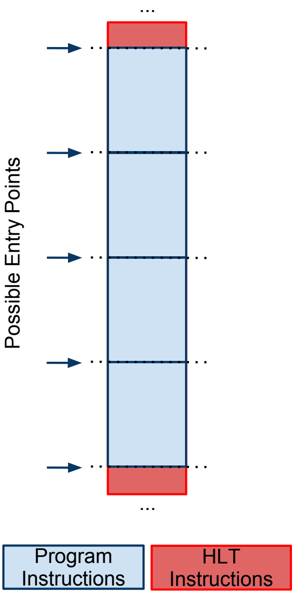
- nacl_dyncode_create
  - Validates new code
  - Two-phase update so that change appears atomic

Lifecycle of a dynamic region



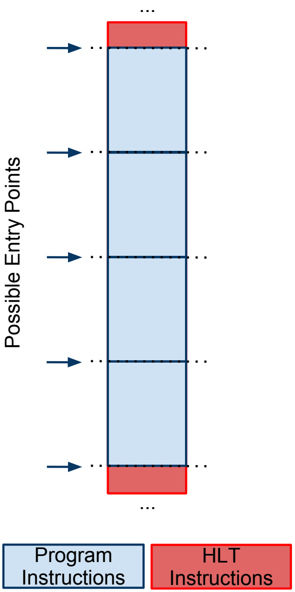
- nacl_dyncode_create
  - Validates new code
  - Two-phase update so that change appears atomic

Lifecycle of a dynamic region



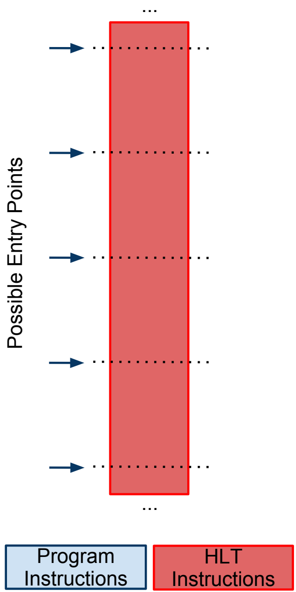
- nacl_dyncode_create
  - Validates new code
  - Two-phase update so that change appears atomic
- nacl_dyncode_modify
  - Possibly called many times
  - More details next

Lifecycle of a dynamic region



- nacl_dyncode_create
  - Validates new code
  - Two-phase update so that change appears atomic
- nacl_dyncode_modify
  - Possibly called many times
  - More details next
- nacl_dyncode_delete
  - Must wait for all threads to leave
  - Wind-down before reuse

Lifecycle of a dynamic region



- nacl_dyncode_create
  - Validates new code
  - Two-phase update so that change appears atomic
- nacl_dyncode_modify
  - Possibly called many times
  - More details next
- nacl_dyncode_delete
  - Must wait for all threads to leave
  - Wind-down before reuse

Modifying dynamic code

- New constraints/validator for replacing code
- New technique for copying replacement code safely

Modifying dynamic code

- New constraints/validator for replacing code
- New technique for copying replacement code safely

New constraints for replacing code *OLD* with code *NEW*

1. NEW must satisfy all NaCl safety constraints
2. NEW and OLD must have the same location and size
3. NEW and OLD must contain identical instruction boundaries
4. No pseudo instructions (guards) are added, changed, or removed

Danger of copying replacement code

Thread 1: in nacl_dyncode_modify

Running:

```
memcpy(A, B, 5);
```

A	PUSH	00	00	00	03
B	JUMP	00	00	00	00

Danger of copying replacement code

Thread 1: in nacl_dyncode_modify

Running:

```
memcpy(A, B, 5);
```

A	PUSH	00	00	00	03
B	JUMP	00	00	00	00

Thread 2: in untrusted code

Executes:

JUMP	00	00	00	03

... and just broke out of the sandbox!

Memory consistency for x86 instructions

- Different than data consistency model
- Requires research to discover
  - Careful reading of documentation
  - Discussions with Intel
  - Tests with micro-benchmarks
- Aligned 8-byte (AMD) or 16-byte (Intel) writes are atomic
- Changes become visible to other processes in an undefined order
- `mfence` doesn't work for instructions!
- Can run the latest instructions by executing a serializing instruction (e.g., `cpuid`)
- We base our algorithm on `SerializeAllProcessors`
  - Forces serializing instruction on each processor
  - Implementation described in the paper

Copying replacement code safely

Pseudo code

```
for (each pair of changed instructions OLD, NEW) {
  if (DiffIsAlignedQuadWord(OLD, NEW)) {
    /* common fast path */
    update OLD with a single aligned movq store;
  } else {
    OLD[0] = 0xf4; /* HLT instruction */

    SerializeAllProcessors();

    OLD[1:n] = NEW[1:n];

    SerializeAllProcessors();

    OLD[0] = NEW[0];
  }
}
```

Outline

Representative language runtimes

- **V8: JavaScript runtime**
  - JIT compiles JavaScript to machine code
  - Heavy use of self-modifying inline caches for performance
    - ($\approx 10x$ performance difference if inline caches are disabled)
- **Mono: C# (and other .NET languages) runtime**
  - JIT compiles Common Intermediate Language (CIL) to machine code
  - Often mixes constant data and code

- Both 32-bit and 64-bit x86 versions of each
  - Code generation backends are different
    - e.g., V8 uses different large integer boxing
  - Native Client requirements are different
    - Memory accesses require guards in 64-bit

Porting effort

- Porting effort relatively straightforward
- Primarily in back-end code generation

	LoC total	LoC added/changed
V8 (32-bit)	190526	1972 (1.04%)
V8 (64-bit)	189969	5005 (2.63%)
Mono (32-bit)	386300	2469 (0.64%)
Mono (64-bit)	388123	3240 (0.83%)

Interesting porting challenges

- Mixed code and data
  - V8: debug, relocation, and other metadata
    - We split the code and metadata
  - Mono: some immediate values
    - We decorated immediates to look like instructions
    - Insert a PUSH opcode

Interesting porting challenges

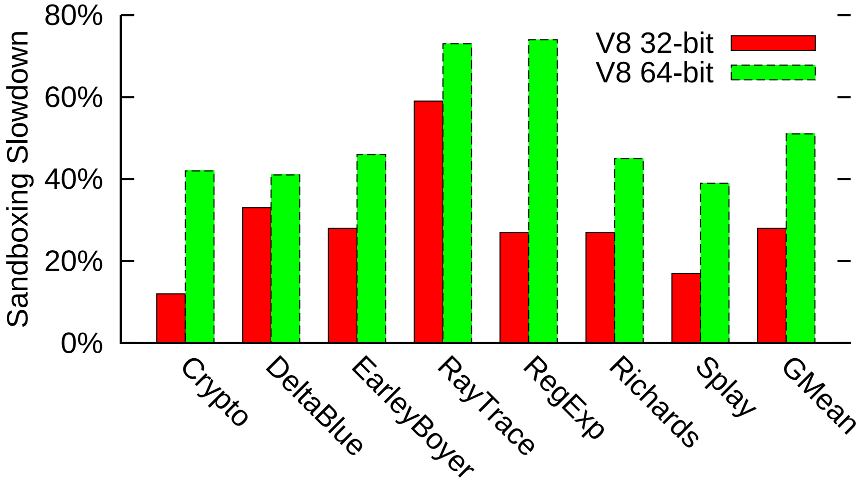
- Mixed code and data
    - V8: debug, relocation, and other metadata
        - We split the code and metadata
    - Mono: some immediate values
        - We decorated immediates to look like instructions
        - Insert a PUSH opcode
- ILP32 data model on 64-bit
    - Pointers are 32-bits on heap, 64-bits on stack
    - Registers different size than pointers
    - Must differentiate stack and heap pointers

Overhead sources breakdown (V8 benchmark suite)

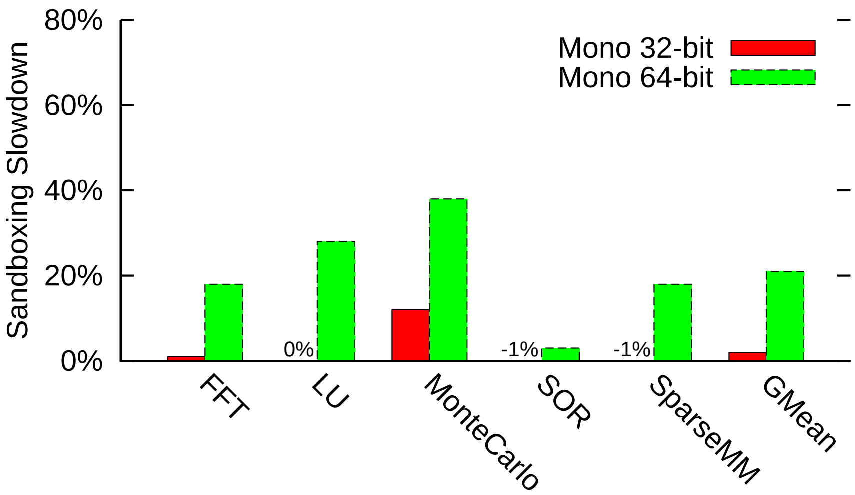
- Estimated by incrementally disabling features
  - Not additive
- Percentage of total overhead

Source of overhead	**32-bit**	**64-bit**
NOP padding	23%	37%
Software guards	42%	46%
Runtime validation	2%	5%

Overheads for V8 (V8 benchmark suite)



Overheads for Mono (SciMark benchmark suite)



More results in our paper

- Other benchmark suites
- Overheads on different microarchitectures
- Comparison to native-C and ahead-of-time compilation
- New “Crankshaft” V8 optimizing backend
- Other optimizations

Outline

Thanks!

Take away

A step towards safely bringing the language-freedom and performance of desktop applications to the web.

- Questions?
- Open source: `http://code.google.com/p/nativeclient/`