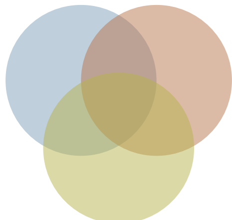
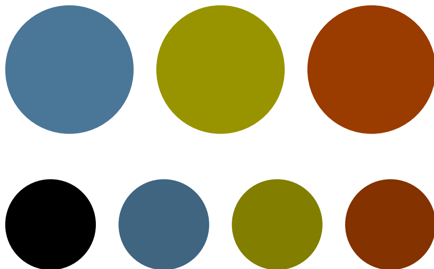
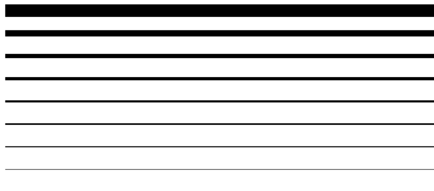


SAFE is a clean-slate effort to build a highly secure computer system, via simultaneous co-design of a new computer architecture, systems software, application software, and programming languages. The SAFE architecture has several peculiarities—pointer bounds checking, fine-grained programmable tags,...





Tempest: A Low-Level Language for a SAFE Machine

DARPA CRASH/SAFE
BAE Systems, Harvard University, Northeastern University, University of Pennsylvania

Jesse A. Tov
(joint work with Edward Amsden, Aleksey Kliger, Greg Morrisett, Luke Palmer, Greg Pfeil, Greg Sullivan, and more)

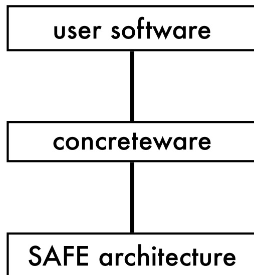
NJPLS
November 15, 2013

The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

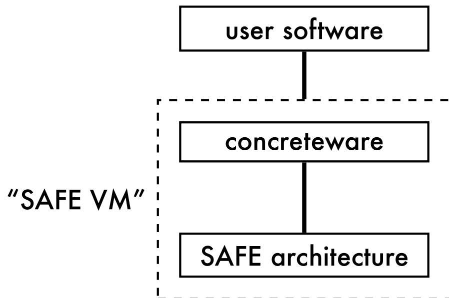
CRASH/SAFE: A Clean-Slate Design

What if we could start over?

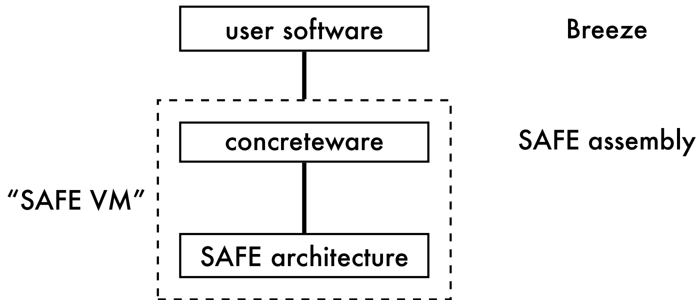
CRASH/SAFE: A Clean-Slate Co-Design



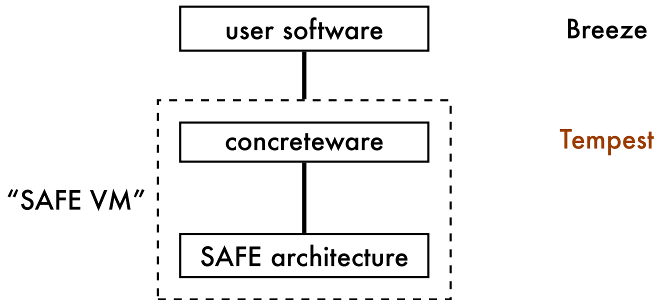
CRASH/SAFE: A Clean-Slate Co-Design



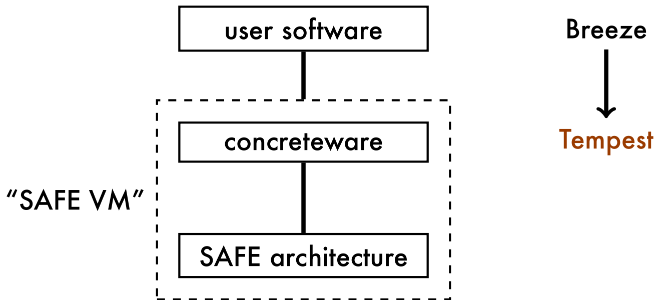
CRASH/SAFE: A Clean-Slate Co-Design



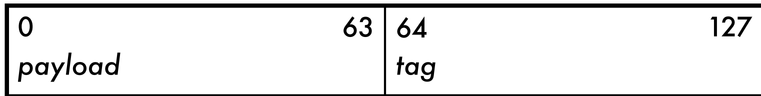
CRASH/SAFE: A Clean-Slate Co-Design



CRASH/SAFE: A Clean-Slate Co-Design



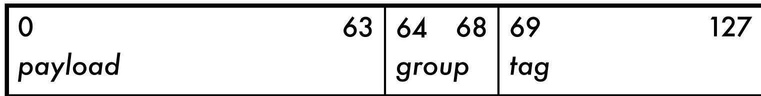
Tags, Groups, and Low-Fat Pointers



Tags, Groups, and Low-Fat Pointers

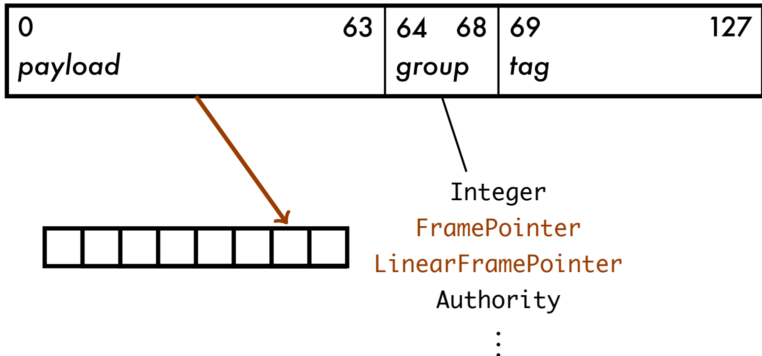
0 … 63	64 … 68	69 … 127
payload	*group*	*tag*

Tags, Groups, and Low-Fat Pointers

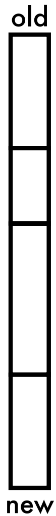



```
Integer
FramePointer
LinearFramePointer
Authority
⋮
```

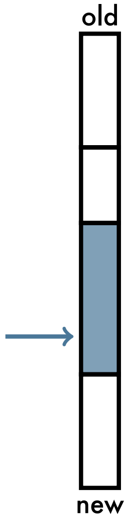
Tags, Groups, and Low-Fat Pointers



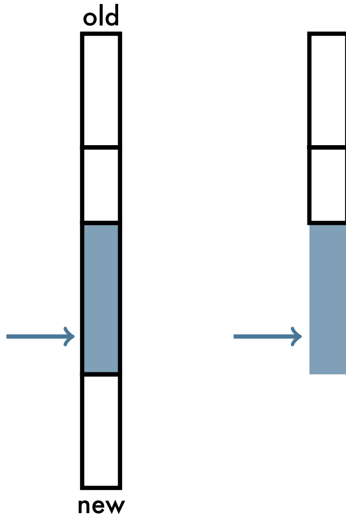
The Stack



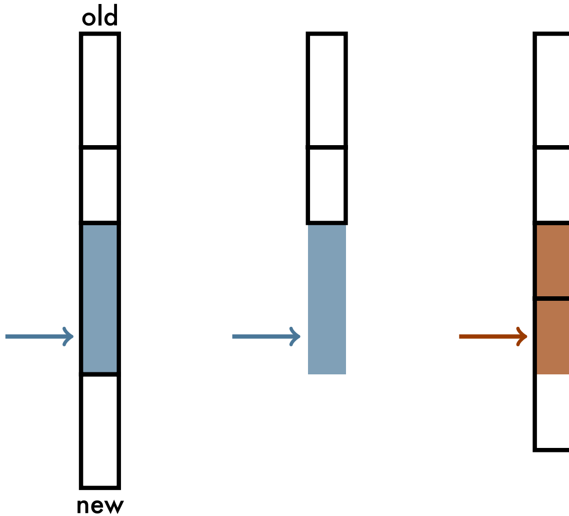
The Stack



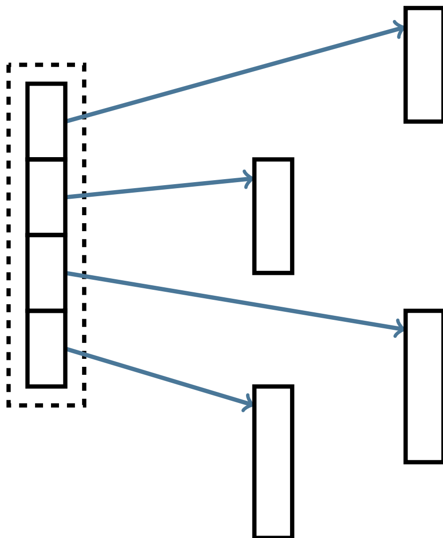
The Stack



The Stack



The Stack, Protected



What's So Funny?

- *Groups* distinguish pointers from integers
- Pointers are bounds checked
- No stack until GC is available

What’s So Funny?

- *Groups* distinguish pointers from integers
- Pointers are bounds checked
- No stack until GC is available
- Many special-purpose instructions

What's So Funny?

- *Groups* distinguish pointers from integers
- Pointers are bounds checked
- No stack until GC is available
- Many special-purpose instructions
- Several operations involve register masks

What’s So Funny?

- *Groups* distinguish pointers from integers
- Pointers are bounds checked
- No stack until GC is available
- Many special-purpose instructions
- Several operations involve register masks
- Linear pointers
- Secure closures

Tempest Design Criteria

- Access to all architecture features

Tempest Design Criteria

- Access to all architecture features
  - Can write GC, scheduler, etc.

Tempest Design Criteria

- Access to all architecture features
  - Can write GC, scheduler, etc.
- Suitable target for high-level Breeze compiler

Tempest Design Criteria

- Access to all architecture features
  - Can write GC, scheduler, etc.
- Suitable target for high-level Breeze compiler
  - Can use GC

Tempest Design Criteria

- Access to all architecture features
  - Can write GC, scheduler, etc.
- Suitable target for high-level Breeze compiler
  - Can use GC
- Suitable for humans

Why Not *X*? – Related Work

- C
- LLVM, Cyclone, Habit,...
- C−−

Tempest is Low-Level

- “C for SAFE”—plenty of rope
- Or: SAFE assembly with register allocation
- Types are based on SAFE groups
- No runtime library*
- No memory allocation*

* Unless you need it

Tempest is High-Level

- Per-procedure calling conventions and inlining
- Proper tail calls
- Types include structures, unions, arrays, pointers, “newtypes”
- Can take advantage of precise GC

A First Tempest Program

```
fun fibCC fib (i : Int) : Int = {
  var a = 0;
  var b = 1;
  while i > 0 do {
    (a, b) := (b, a + b);
    i      := i - 1;
  };
  a;
};
```

A First Tempest Program

```
fun fibCC fib (i : Int) : Int = {
  var a = 0;
  var b = 1;
  while i > 0 do {
    (a, b) := (b, a + b);
    i      := i - 1;
  };
  a;
};

type fibCC = cconv { 1 2 3 -> 1 2 3; 0..8 : AVAIL };
```

A First Tempest Program

```
fun fibCC fib (i : Int) : Int = {
  var a = 0;
  var b = 1;
  while i > 0 do {
    (a, b) := (b, a + b);
    i      := i - 1;
  };
  a;
};

type fibCC = cconv { 1 2 3 -> 1 2 3; 0..8 : AVAIL };
```

`fib : fibCC(Int -> Int)`

A First Tempest Program

```
fun fibCC fib (i : Int) : Int = {
  var a = 0;
  var b = 1;
  while i > 0 do {
    (a, b) := (b, a + b);
    i      := i - 1;
  };
  a;
};

fun inline (-) (a, b : Int) : Int = {
  var result : Int;
  asm sub $a $b $result;
  result;
}
```

A First Tempest Program

```
fun fibCC fib (i : Int) : Int = {
  var a = 0;
  var b = 1;
  while i > 0 do {
    (a, b) := (b, a + b);
    i      := i - 1;
  };
  a;
};

fun inline (-) (a, b : Int) : Int =
  asm (result : Int) sub $a $b $result;
```

Fibonacci, Compiled

```
.atomtag Bottom_Tag
.frame fib
fib:
    mvrr r1 r3
    lcfp r1 __1
    cpmr r1 r1
    lcfp r2 __2
    cpmr r2 r2
L0:
    bg r3 L1
    grtn
L1:
    cprr r1 r0
    cprr r2 r1
    add r1 r0 r2
    lcfp r0 __2
    cpmr r0 r0
    sub r3 r0 r3
    jmp L0
__1:
    .data 0x00000000 Integer
__2:
    .data 0x00000001 Integer
.endframe
```

Fibonacci, Compiled

```
.atomtag Bottom_Tag
.frame fib
fib:
    mvrr r1 r3        i in r3
    lcfp r1 __1
    cpmr r1 r1
    lcfp r2 __2
    cpmr r2 r2
L0:
    bg r3 L1
    grtn
L1:
    cprr r1 r0
    cprr r2 r1
    add r1 r0 r2
    lcfp r0 __2
    cpmr r0 r0
    sub r3 r0 r3
    jmp L0
__1:
    .data 0x00000000 Integer
__2:
    .data 0x00000001 Integer
.endframe
```

Fibonacci, Compiled

```
.atomtag Bottom_Tag
.frame fib
fib:
    mvrr r1 r3
    lcfp r1 __1
    cpmr r1 r1
    lcfp r2 __2
    cpmr r2 r2
L0:
    bg r3 L1
    grtn
```

$i \text{ in } r3$

$a := 0$

$b := 1$

```
L1:
    cprr r1 r0
    cprr r2 r1
    add r1 r0 r2
    lcfp r0 __2
    cpmr r0 r0
    sub r3 r0 r3
    jmp L0
__1:
    .data 0x00000000 Integer
__2:
    .data 0x00000001 Integer
.endframe
```

Fibonacci, Compiled

```
.atomtag Bottom_Tag
.frame fib
fib:
    mvrr r1 r3        i in r3
    lcfp r1 __1
    cpmr r1 r1        a := 0
    lcfp r2 __2
    cpmr r2 r2        b := 1
L0:
    bg r3 L1          test i > 0
    grtn              return a
```

```
L1:
    cprr r1 r0
    cprr r2 r1
    add r1 r0 r2
    lcfp r0 __2
    cpmr r0 r0
    sub r3 r0 r3
    jmp L0
__1:
    .data 0x00000000 Integer
__2:
    .data 0x00000001 Integer
.endframe
```

Fibonacci, Compiled

```
.atomtag Bottom_Tag
.frame fib
fib:
    mvrr r1 r3          i in r3
    lcfp r1 __1
    cpmr r1 r1          a := 0
    lcfp r2 __2
    cpmr r2 r2          b := 1
L0:
    bg r3 L1            test i > 0
    grtn                return a
L1:
    cprr r1 r0          temp := a
    cprr r2 r1          a := b
    add r1 r0 r2        b := a + temp
    lcfp r0 __2
    cpmr r0 r0
    sub r3 r0 r3
    jmp L0
__1:
    .data 0x00000000 Integer
__2:
    .data 0x00000001 Integer
.endframe
```

Fibonacci, Compiled

```
.atomtag Bottom_Tag
.frame fib
fib:
    mvrr r1 r3        i in r3
    lcfp r1 __1
    cpmr r1 r1        a := 0
    lcfp r2 __2
    cpmr r2 r2        b := 1
L0:
    bg r3 L1          test i > 0
    grtn              return a
L1:
    cprr r1 r0        temp := a
    cprr r2 r1        a := b
    add r1 r0 r2      b := a + temp
    lcfp r0 __2
    cpmr r0 r0        temp := 1
    sub r3 r0 r3      i := i - temp
    jmp L0            to loop header
__1:
    .data 0x00000000 Integer
__2:
    .data 0x00000001 Integer
.endframe
```

Using the Allocator

```
type Node = { value : Int; left, right : Tree };
type Tree = { null : Int | node : FP(Node) };
```

Using the Allocator

```
type Node = { value : Int; left, right : Tree };
type Tree = { null : Int | node : FP(Node) };

type cc = { 1..15 -> 1..6; 0..30 : AVAIL, 31 : ALLOC };

fun cc sumTree (tree : Tree) : Int =
  if isInt(tree)
    then 0
    else !tree.node.value +
         sumTree (!tree.node.left) +
         sumTree (!tree.node.right);
```

More on Tempest Types

`Int`	: U
`FP(t)`	: U

More on Tempest Types

`Int`	: U
`FP(t)`	: U
`LFP(t)`	: L

More on Tempest Types

`Int`	: U
`FP(t)`	: U
`LFP(t)`	: L
`{ a : Int; b : FP(Int); c : LFP(Int) }`	: U*U*L
`Int[8]`	: U*U*U*U*U*U*U*U

More on Tempest Types

`Int`	: U
`FP(t)`	: U
`LFP(t)`	: L
`{ a : Int; b : FP(Int); c : LFP(Int) }`	: U*U*L
`Int[8]`	: U[8]

More on Tempest Types

```
Int                                              : U
FP(t)                                            : U
LFP(t)                                           : L
{ a : Int; b : FP(Int); c : LFP(Int) }           : U*U*L
Int[8]                                           : U[8]
((Int, Label)[6], Int)                           : U[13]
```

More on Tempest Types

`Int`	: U
`FP(t)`	: U
`LFP(t)`	: L
`{ a : Int; b : FP(Int); c : LFP(Int) }`	: U*U*L
`Int[8]`	: U[8]
`((Int, Label)[6], Int)`	: U[13]
`Int[]`	: MEMORY
`{ size : Int; chars : Char[] }`	: MEMORY

More on Tempest Types

`Int`	: U
`FP(t)`	: U
`LFP(t)`	: L
`{ a : Int; b : FP(Int); c : LFP(Int) }`	: U*U*L
`Int[8]`	: U[8]
`((Int, Label)[6], Int)`	: U[13]
`Int[]`	: MEMORY
`{ size : Int; chars : Char[] }`	: MEMORY
`FP({ size : Int; chars : Char[] })`	: U

Function Types and Subtyping

```
  ccall { 1..6 -> 1..6; 1..10 : AVAIL } (Int, Int -> Int)
= ccall { 1..2 -> 1   ; 1..10 : AVAIL } (Int, Int -> Int)
```

Function Types and Subtyping

```
  ccall { 1..6 -> 1..6; 1..10 : AVAIL } (Int, Int -> Int)
= ccall { 1..2 -> 1   ; 1..10 : AVAIL } (Int, Int -> Int)

  ccall { 1..2 -> 1; 1..10 : AVAIL } (Int, Int -> Int)
≤ ccall { 1..2 -> 1; 1..15 : AVAIL } (Int, Int -> Int)
```

Function Types and Subtyping

```
  ccall { 1..6 -> 1..6; 1..10 : AVAIL } (Int, Int -> Int)
= ccall { 1..2 -> 1   ; 1..10 : AVAIL } (Int, Int -> Int)

  ccall { 1..2 -> 1; 1..10 : AVAIL } (Int, Int -> Int)
≤ ccall { 1..2 -> 1; 1..15 : AVAIL } (Int, Int -> Int)

  { a : t; b : Int }
≤ { a : u; b : Int }     if t ≤ u
```

Function Types and Subtyping

```
  ccall { 1..6 -> 1..6; 1..10 : AVAIL } (Int, Int -> Int)
= ccall { 1..2 -> 1   ; 1..10 : AVAIL } (Int, Int -> Int)

  ccall { 1..2 -> 1; 1..10 : AVAIL } (Int, Int -> Int)
≤ ccall { 1..2 -> 1; 1..15 : AVAIL } (Int, Int -> Int)

  { a : t; b : Int }
≤ { a : u; b : Int }     if t ≤ u

  { a : Int; b : Int }
≰ { a : Int }
```

Function Types and Subtyping

```
  ccall { 1..6 -> 1..6; 1..10 : AVAIL } (Int, Int -> Int)
= ccall { 1..2 -> 1   ; 1..10 : AVAIL } (Int, Int -> Int)

  ccall { 1..2 -> 1; 1..10 : AVAIL } (Int, Int -> Int)
≤ ccall { 1..2 -> 1; 1..15 : AVAIL } (Int, Int -> Int)

  { a : t; b : Int }
≤ { a : u; b : Int }     if t ≤ u

  { a : Int; b : Int }
≰ { a : Int }

  FP({ a : Int; b : Int })
≤ FP({ a : Int })
```

Implementation and Use

- $\sim$12k lines of Haskell
- About five users (plus three Tempest compiler developers)
- $\sim$20k lines of Tempest have been written (tests, concreteware, and a web server)

What's Next

- Polymorphism (parametric, ad hoc)
- Interprocedural calling convention inference?

What's Next

- Polymorphism (parametric, ad hoc)
- Interprocedural calling convention inference?
- A debugger

What’s Next

- Polymorphism (parametric, ad hoc)
- Interprocedural calling convention inference?
- A debugger
- A model—what properties does it have?

Why Should We Care?

- What are user-defined calling conventions good for?
  - Target for calling convention specialization
  - Maybe even type-preserving

Why Should We Care?

- What are user-defined calling conventions good for?
  - Target for calling convention specialization
  - Maybe even type-preserving
- Recipe for bootstrapping a low-level language:
  - Inline assembly
  - Register allocation and procedure calls
  - Structural types over ISA-derived base types

Thank you