Krabcake: A Rust UB detector

Felix S. Klock (pnkfelix), Bryan Garza (bryangarza)
Amazon Web Services
Rust Platform Team

demo

source

machine code

binary run

krabcake run

source

```rust
use krabcake::ClientRequest;
println!("Hello world (from `sb_rs_port/main.rs`)!");
println!("BorrowMut is {:x}", ClientRequest::BorrowMut);
let mut val: u8 = 101;      // 0x65
let x = &mut val;
let x_alias = x as *mut u8;
let y = &mut *x;
*y = 105;                   // 0x69
// This could live in *foreign code*
unsafe { *x_alias = 103; } // 0x67
let end = *y;
println!("Goodbye world, end: {}!", end);
```

machine code

```
879f:  movb   $0x65,0x6(%rsp)
87a4:  movq   $0x4b430000,0x8(%rsp)
87ad:  lea    0x6(%rsp),%rdi
87b2:  mov    %rdi,0x10(%rsp)
87b7:  movaps 0x35842(%rip),%xmm0
87be:  movups %xmm0,0x18(%rsp)
87c3:  movaps 0x35846(%rip),%xmm1
87ca:  movups %xmm1,0x28(%rsp)
87cf:  lea    0x8(%rsp),%rax
       [...]
87e7:  mov    %rdi,%rcx
87ea:  movq   $0x4b430000,0x8(%rsp)
87f3:  mov    %rdi,0x10(%rsp)
87f8:  movups %xmm0,0x18(%rsp)
87fd:  movups %xmm1,0x28(%rsp)
8802:  lea    0x8(%rsp),%rax
       [...]
881a:  movb   $0x69,(%rdi)
881d:  movb   $0x67,(%rcx)
8820:  movzbl (%rdi),%eax
```

direct run

```
$ ./sb_rs_port/target/release/sb_rs_port
Hello world (from `sb_rs_port/main.rs`)!
BorrowMut is 4b430000
Goodbye world, end: 103!
```

krabcake run

```
$ ./bin/valgrind -q --tool=krabcake ./sb_rs_port/target/release/sb_rs_port
Hello world (from `rs_hello/src/lib.rs`)!
Hello world (from `sb_rs_port/main.rs`)!
BorrowMut is 4b430000
--974553-- kc_main.c: dispatching code 4b430000
--974553-- lib.rs: handle client request BORROW_MUT 0x1ffeffff66
--974553-- kc_main.c: dispatching code 4b430000
--974553-- lib.rs: handle client request BORROW_MUT 0x1ffeffff66
==974553== ALERT could not find tag 2 in stack for address 0x1ffeffff66
Goodbye world, end: 103!
```

Talk Outline

demo

motivation

approaches

our solution

technical details

pulling back the curtain

Motivation

Rust’s promise: *control* and *safety*

Control AND Safety

Can you really provide both?

`unsafe { ... }` is the escape hatch

How can one be confident it is used correctly?

Approaches

Stacked Borrows

(aka "SB")

$$t \in \mathit{Tag} \triangleq \mathbb{N} \qquad \mathit{Scalar} \triangleq \mathrm{Pointer}(\ell, t) \mid z \qquad \text{where } z \in \mathbb{Z}$$

$$\mathit{Item} \triangleq \mathrm{Unique}(t) \mid \ldots \qquad \mathit{Mem} \triangleq \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Scalar} \times \mathit{Stack}$$

$$\mathit{Stack} \triangleq \mathit{List}(\mathit{Item})$$

Great! A way to discuss correctness of unsafe code!

Verification?

Verification tools are great!

But: can they be broadly applied?

Verification requires developer investment

Tools usually assume foreign libraries satisfy specifications / do not break language invariants

(Kani is an exception; includes foreign code in its checking. But does not include checking of stacked borrow semantic rules; not yet.)

Miri?

Great test bed; Reference for Stacked Borrows (SB)

Directly expresses SB domains

e.g. Pointers *are* taken from (Loc x Tag)

Limited in practice: No inline asm nor arbitrary FFI

Sanitizer?

A MIR-to-MIR rewrite that injects SB checks?

Doesn’t address SB’s domains for Scalar and Mem

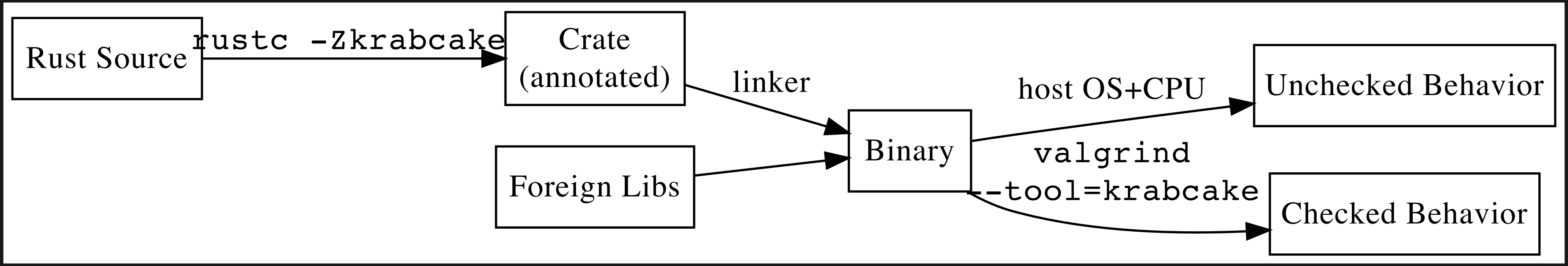
… or if it does (a la Miri), it breaks interop

The Key Problem

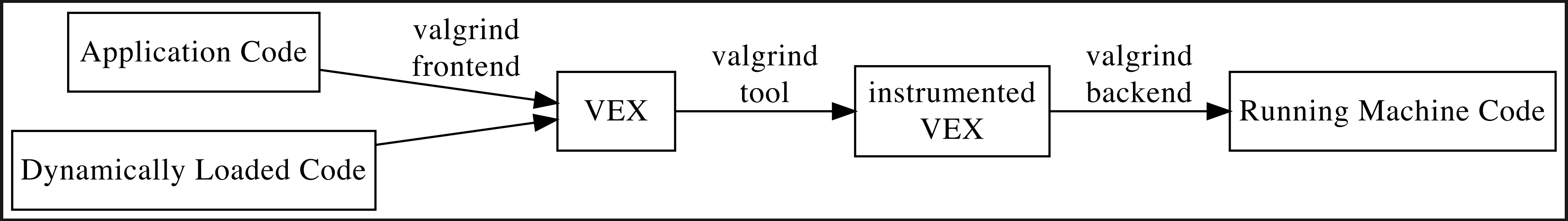
- Q: Want foreign interop *and* pointer tagging
- A1: We could sanitize *everything*
  - i.e. recompile all linked C code to inject tags
  - but … what would `sizeof(T*)` return? (How much would that break?)
  - *also*: not realistic! Cannot expect everyone to recompile world
- A2: Dynamic Binary Instrumentation! i.e. A Valgrind tool

Solution: Krabcake

Krabcake Overview



Valgrind

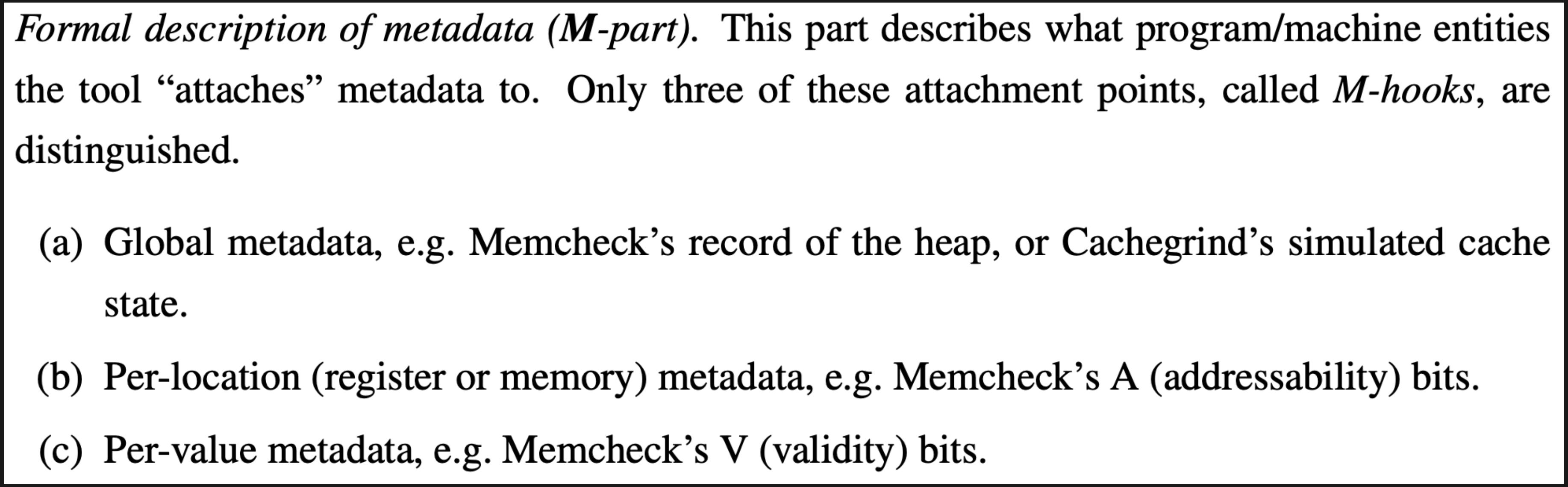

Application Code
Dynamically Loaded Code
valgrind frontend
VEX
valgrind tool
instrumented VEX
valgrind backend
Running Machine Code

Krabcake: Technical Details

From Nicholas Nethercote

(one of Valgrind’s two main authors)

“You should read chapter 6 of my thesis”



*Formal description of metadata (**M**-part).* This part describes what program/machine entities the tool “attaches” metadata to. Only three of these attachment points, called *M-hooks*, are distinguished.

(a) Global metadata, e.g. Memcheck’s record of the heap, or Cachegrind’s simulated cache state.

(b) Per-location (register or memory) metadata, e.g. Memcheck’s A (addressability) bits.

(c) Per-value metadata, e.g. Memcheck’s V (validity) bits.

- Per-location – that’s the SB Stacks…
- Per-value – that’s the SB *Tags* !!!

Shadow Memory

During the VEX to VEX rewrite, inject new operations that build and maintain shadow state for memory addresses, registers, and the intermediate SSA temporaries of the VEX IR.

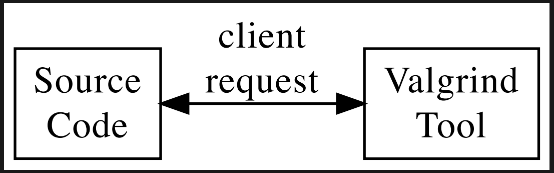
(Probably hardest implementation step.)

A Rust Gotcha for Valgrind

- Q: How can Valgrind implement the SB rules?
- At machine code level, {`&mut`, `&`, `&raw`} are not distinguishable.
- A1: Valgrind cannot. Not without help.
- A2: `rustc -Zkrabcake <input>` *annotates* the machine code to make them distinguishable.

Annotated machine code?

Yes, using the Valgrind “client request” mechanism.



Trapdoors for code, inserted prior to Valgrind instrumentation. They are specially interpreted during instrumentation, become communication channels

Can annotate each `&mut`- and `&`-borrow so that the Valgrind tool can distinguish them from `&raw`-borrows !

Isn’t that sanitizing?

We do require `rustc` assistance.

It’s implementation (and maintenance!) should be lightweight.

We don’t sanitize the foreign code. All annotations are injected solely on the Rust side.

Pulling Back the Curtain

One White Lie

There is no `rustc -Z krabcake` flag. Not yet.

Wanted proof-of-concept valgrind tool first.

The Real Code

```rust
use krabcake::ClientRequest;
println!("Hello world (from `sb_rs_port/main.rs`)!");
println!("BorrowMut is {:x}", ClientRequest::BorrowMut);
let mut val: u8 = 101;

let x = kc_borrow_mut!(val); // aka `&mut val`
let x_alias = x as *mut u8;
let y = kc_borrow_mut!(*x);  // aka `&mut *x`

*y = 105;
unsafe { *x_alias = 103; }

let end = *y;
```

What's `kc_borrow_mut!`?

```
macro_rules! kc_borrow_mut {
  ( $data:expr ) => {{
    let place = &mut $data;
    let raw_ptr = valgrind_do_client_request_expr!(
      place as *mut u8,
      crate::krabcake::ClientRequest::BorrowMut,
      place as *mut u8,
      0x91, 0x92, 0x93, 0x94); // (these are ignored)
    // When rustc machinery is in place, `kc_borrow_mut!(PLACE)` will
    // be replaced with `&mut PLACE`. Therefore, we go ahead and
    // convert the `&raw` place above into an `&mut`, so that the
    // appropriate type is inferred for the expression.
    if true {
      unsafe { &mut *raw_ptr }
    } else {
      // return original `&mut` on false branch, forcing lifetimes on
      // `&mut` above to match lifetimes assigned to original place.
      place
    }
  }};
}
```

What’s `valgrind_do_client_request_expr!`?

```rust
#[macro_export]
macro_rules! valgrind_do_client_request_expr {
    ( $zzq_default:expr, $request_code:expr,
      $arg1:expr, $arg2:expr, $arg3:expr, $arg4:expr, $arg5:expr ) => {
        {
            let zzq_args = crate::Data {
                request_code: $request_code as u64,
                arg1: $arg1,
                arg2: $arg2,
                arg3: $arg3,
                arg4: $arg4,
                arg5: $arg5,
            };
            let mut zzq_result = $zzq_default;
            #[allow(unused_unsafe)]
            unsafe {
                ::std::arch::asm!(
                    "rol rdi, 3",
                    "rol rdi, 13",
                    "rol rdi, 61",
                    "rol rdi, 51",
                    "xchg rbx, rbx",
                    inout("di") zzq_result,
                    in("ax") &zzq_args,
                );
            }
            zzq_result
        }
    }
}
```

Also

The tool is not finished yet

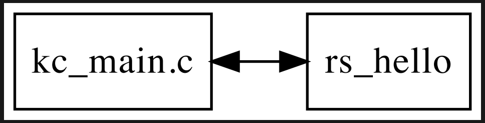
(e.g. have not implemented pointer arithmetic in the shadow memory system)

I know you’re dissapointed by all that bad news

But I have good news

The tool … is in Rust

(mostly)



`rs_hello` is a `#![no_std]` staticlib crate.

`kc_main.c` instruments VEX to add calls into `rs_hello`, building and manipulating shadow state (the tags and stacks).

Low barrier for contributions from Rust community!

Conclusion

Unsafe code developers need validation tools

Verification is great, if available

Lighter weight tools can be applied to arbitrary projects with little developer investment

Krabcake wants to fill that niche

If you are interested, reach out!

Thanks!

```
$ ./bin/valgrind -q --tool=krabcake ./sb_rs_port/target/release/sb_rs_port
Hello world (from `rs_hello/src/lib.rs`)!
Hello world (from `sb_rs_port/main.rs`)!
BorrowMut is 4b430000
--974553-- kc_main.c: dispatching code 4b430000
--974553-- lib.rs: handle client request BORROW_MUT 0x1ffeffff66
--974553-- kc_main.c: dispatching code 4b430000
--974553-- lib.rs: handle client request BORROW_MUT 0x1ffeffff66
==974553== ALERT could not find tag 2 in stack for address 0x1ffeffff66
Goodbye world, end: 103!
```