Evaluating tail call elimination in the face of return address protection, part 1

October 17, 2024

Tail call elimination is straightforward if the tail call is to a function with a compatible stack parameter layout as the original function, since you can just replace the parameter slots on the stack with the new parameters. (The register-based parameters you can just overwrite directly in registers.)

The obvious case where this applies is where the tail-calling and tail-called functions both have the same number of stack-based parameters. Just reuse the slots and jump to the next function.

But you can also employ tail calling even if the number of stack-based parameters does not match exactly.

One case where the tail call is possible is if the tail-called function has fewer parameters as the tail-calling function, and the calling convention is caller-clean. In that case, you can reuse the stack slots for the outbound parameters, and just leave any extra ones uninitialized. The tail-called function won't use them, but the original caller will still clean them up. (Note that this doesn't work in reverse: If the tail-called function has *more* parameters than the tail-calling function, you can't just smash the extra parameters onto the stack beyond those of the tail-calling function, because that's writing into stack space that belongs to the original caller.)

Here's an example of a tail call on x86-32 to a function with fewer stack-based parameters.

```
int __cdecl g(int c);

int __cdecl f(int a, int b)
{
    int v = helper(a, b);

    return g(a + b);

}
```

You can reuse the stack space for the tail call to `g`

```
    ; on entry, stack parameters are at [esp+4]
    ; and [esp+8]

    ; v = helper(a, b)
    push    [esp+8]
    push    [esp+8]
    call    helper

    ; reuse the "a" slot for the outbound
    ; "c" slot
    mov     [esp+4], eax

    ; tail call to g
    jmp     g
```

The caller of `f` will clean up two stack slots, and everything will return to normal. What the original caller doesn't realize is that we reused one of them for `g`, and the other still contains leftover data from `f`. Logically, you can think that we inlined all of `g` into `f`.

How does this interact with return address protection?

Since we aren't creating any imbalance in `call` or `ret` instructions, compact shadow stacks are still happy. And since the return address did not move in memory, parallel shadow stacks and return address signing are still satisfied. (For architectures that use a link register, don't forget to authenticate the link register before jumping to the tail-called function, so that the link register on entry to the tail-called function is untagged.)

Next time, we'll look at another type of tail call elimination and study how it interacts with return address protection.