Evaluating tail call elimination in the face of return address protection, part 2

devblogs.microsoft.com/oldnewthing/20241018-00

October 18, 2024

Last time, we looked at tail call elimination that reuse the activation frame.

You may be tempted to expand the scope of tail calls to functions with different stack parameter layouts by editing the stack. If the tail-calling and tail-called functions both follow the callee-clean convention, the original caller expects the stack to be clean, and as long as we arrange for that to be the case at the end of the tail-called function, nobody will be any wiser, right?

```
int __stdcall g(int, int);

int __stdcall f(int v)
{
    ⟦ do some stuff ⟧

    return g(v, 42);

}
```

You might try to use a tail call to `g` by inserting the additinal parameter under the existing one.

```
; on entry, stack parameter is at [esp+4]

⟦ do some stuff ⟧

; pop our return address
pop     eax

; pop our parameter v
pop     ecx

; push the extra parameter
push    42

; push v back
push    ecx

; restore return address
push    eax

; tail call to g
jmp     g
```

While this technique keeps the original caller happy, you still have a problem: By manipulating the return address on the stack, you risk messing up return address protection.

Let's evaluate these mechanisms in the light of the various return address protection mechanisms we know about.

For AArch64 return address signing, the signature is a hash of the return address itself, the value of the stack pointer, and some secret data in the kernel. If you copy the return address to a new location, that invalidates the hash (because the stack pointer is now different), and that means that your rewritten activation frame will result in a fatal error when the function `g` tries to use the (now-invalid) return address. As noted before, ou can fix this by authenticating the link register before jumping to the tail-called function, so that the link register on entry to the tail-called function is untagged. However, activation frame rewriting is not really an issue on AArch64 since all the major calling conventions are caller-clean.

For x86-32 and x86-64 with compact shadow stacks or hardware shadow stacks (which are compact), the return address is tracked by call depth rather than stack position, so copying a return address to a new location does not invalidate the shadow stack.

For x86-32 and x86-64 with parallel shadow stacks, copying the return address to a new location will not copy the shadow, so when the tail-called function tries to return, it will not see a match between the CPU stack and the shadow stack and raise a fatal error.

So activation frame rewriting looks like a bad idea on all architectures. Which is probably a good thing, since they also are not easily expressible by unwind codes (if at all). Even if you managed to keep the shadow stacks happy, the system exception handler and stack walkers

won’t like them.

Bonus chatter: As I recall, callee-clean as a calling convention exists in Windows only in x86-32. All other architectures on Windows use exclusively caller-clean calling conventions. (Another case where x86-32 is the weirdo.)