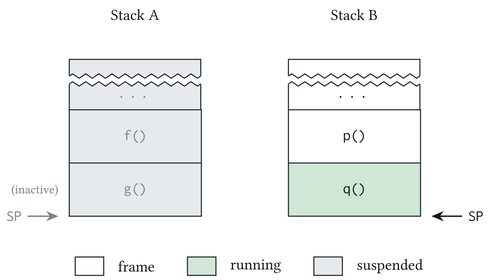
Verified Stack Switching (CompCertFX)

Barış Bayazıt, Oghenevwogaga Ebresafe, Ningning Xie

Computer Science
UNIVERSITY OF TORONTO

The Problem: Stack Switching is Unverified

Can you pause a function, let other code run, and later resume from that line?



Used by:

- goroutines
- coroutines
- green threads
- language runtimes
- fibers
- the OS
- effect handlers
- … many more!

Stack-related bugs were found in many compilers (e.g., Go's arm64 compiler).

CompCertFX **is the first formally verified, realistic compiler** with stack switching.
No realistic compiler has ever been **proved** to implement stack switching correctly.

Background: Effect Handlers

```
try {
  throw Fail;
  ... // never runs
} catch Fail {
  printf("Error...");
};
```

```
handle {
  int r = perform Fail();  // switch away
  ... // runs after resume
} with {
  Fail(k) → {
      printf("Error recovery...");
      resume k(3);  // switch back
  }
};
```

- Effect handlers generalize exceptions and can express generators.
- To **`perform`** means **to suspend** the computation being handled.
- To **`resume`** means **to continue** it and return to the clause later.

Background: CompCert C Compiler

CompCert is the **only production C compiler with a machine-checked correctness proof** from source to assembly.

The Frontend

SEMANTICS What each operation means

$$\frac{}{\langle F, \mathsf{Shandle}(x, H, F_c), k, \rho, re\rangle \rightsquigarrow \mathsf{Call}\langle F_c, [\,], \mathsf{Khandle}(H, \mathsf{Kcall}(x, F, \rho, re, k)), re\rangle} \quad \textsc{Handle}$$

$$\frac{\text{fresh } r \qquad \mathsf{find\_handler}(\ell, k) = (k_r, H, k_o) \qquad H[\ell] = F_\ell}{\langle F, \mathsf{Sperform}(x, \ell, \bar{v}), k, \rho, re\rangle \rightsquigarrow \mathsf{Call}\langle F_\ell, \bar{v} +\!\!+ [r], k_o, re\,[r \mapsto \mathsf{Kcall}(x, F, \rho, re, k_r)]\rangle} \quad \textsc{Perform}$$

$$\frac{re[r] = k_r \qquad re' = re\,[r \mapsto \bot]}{\langle F, \mathsf{Sresume}(x, r, v), k, \rho, re\rangle \rightsquigarrow \mathsf{Return}\langle v, k_r +\!\!+ \mathsf{Kcall}(x, F, \rho, re', k), re\rangle} \quad \textsc{Resume}$$

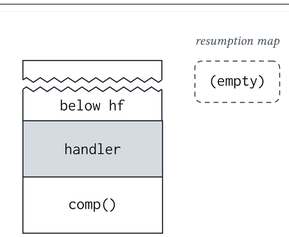
CONTINUATIONS The "rest" of the program

Kseq	`x = 1;`	→	`Kseq( x = 1, <rest> )`
Kcall	`main() { g(); }`	→	`Kcall( main, <rest> )`
Khandle	`handle { g(); }`	→	`Khandle( H, <rest> )`

The Backend

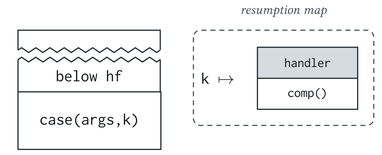
1 HANDLE Mark the boundary

```
void f3() {
  handle comp() with {
    Effect(x, k) → case(x, k)
  };
}
```



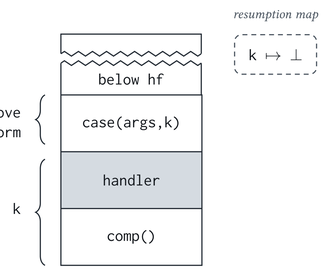
2 PERFORM Save the computation

```
void comp() {
  ...
  perform Effect(...);
  // resumption starts here
}
```



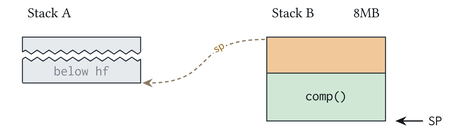
3 RESUME Reattach the segment

```
void case(..., k) {
  ...
  resume k(...);
  // want to come back after
  // resume finishes
}
```

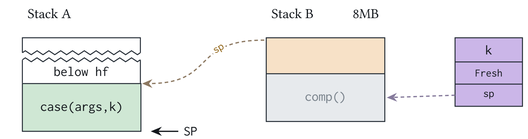


Stack Switching on the Machine

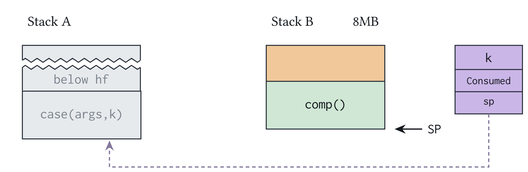
1 HANDLE
Allocate Stack B

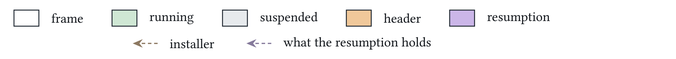


2 PERFORM
Save Stack B

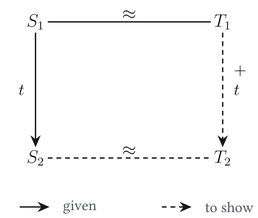


3 RESUME
Restore Stack B
k is consumed





Result: Verified Stack Switching



1. Start from related states: $S_1 \approx T_1$.
2. If the source steps $S_1 \xrightarrow{t} S_2$,
3. the target must take matching steps to some T_2, produce the **same external events**, and finish related: $S_2 \approx T_2$.

t records externally visible events, such as system calls.

Why is this difficult?

- **Extended semantics:** existing handler designs do not fit CompCert!
- **End-to-end:** must modify the full pipeline of 10 IRs + translations!
- **Correctness proofs:** every IR needs its own simulation proof.
- **Minimal intervention:** we cannot touch existing C semantics.