UNIVERSITY OF CAMBRIDGE

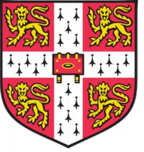
What you get is what you C: Controlling side effects in mainstream C compilers

Laurent Simon, David Chisnall, Ross Anderson

l.simon@samsung.com
https://sites.google.com/view/laurent-simon/

Approved for public release; distribution is unlimited. Sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237 (“CTSRD”) as part of the DARPA CRASH research program. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

Talk outline



- Compiler Optimizations and Side Effects
- Example: constant-time choose
- Proposed Solution and Evaluation
- Conclusion

```
int sign_data(...) {
	// prompt used for password
	char password[MAX_PWD_LEN] = {0};
	get_pasword_from_user( password );




	return OK;
}
```

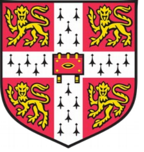
```
int sign_data(...) {
	// prompt used for password
	char password[MAX_PWD_LEN] = {0};
	get_pasword_from_user( password );

	// load private key decrypted with password
	u8 * privKey = malloc( … )
	get_private_key( privKey, password );









	return OK;
}
```

```
int sign_data(...) {
        // prompt used for password
        char password[MAX_PWD_LEN] = {0};
        get_pasword_from_user( password );

        // load private key decrypted with password
        u8 * privKey = malloc( … )
        get_private_key( privKey, password );

        // sign the data
        sign_data( privKey, ... );




        return OK;
}
```

```
int sign_data(...) {
        // prompt used for password
        char password[MAX_PWD_LEN] = {0};
        get_pasword_from_user( password );

        // load private key decrypted with password
        u8 * privKey = malloc( ... )
        get_private_key( privKey, password );

        // sign the data
        sign_data( privKey, ... );

        // zero memory
        memset(password, 0, MAX_PWD_LEN);
        memset(privKey, 0, privLen);

        free(privKey);

        return OK;
}
```

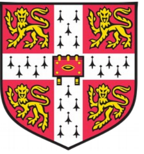
```
int sign_data(…) {
	// prompt used for password
	char password[MAX_PWD_LEN] = {0};
	get_pasword_from_user( password );

	// load private key decrypted with password
	u8 * privKey = malloc( … )
	get_private_key( privKey, password );

	// sign the data
	sign_data( privKey, ... );

	// zero memory
	memset(password, 0, MAX_PWD_LEN);
	memset(privKey, 0, privLen);

	free(privKey);

	return OK;
}
```

Question 1



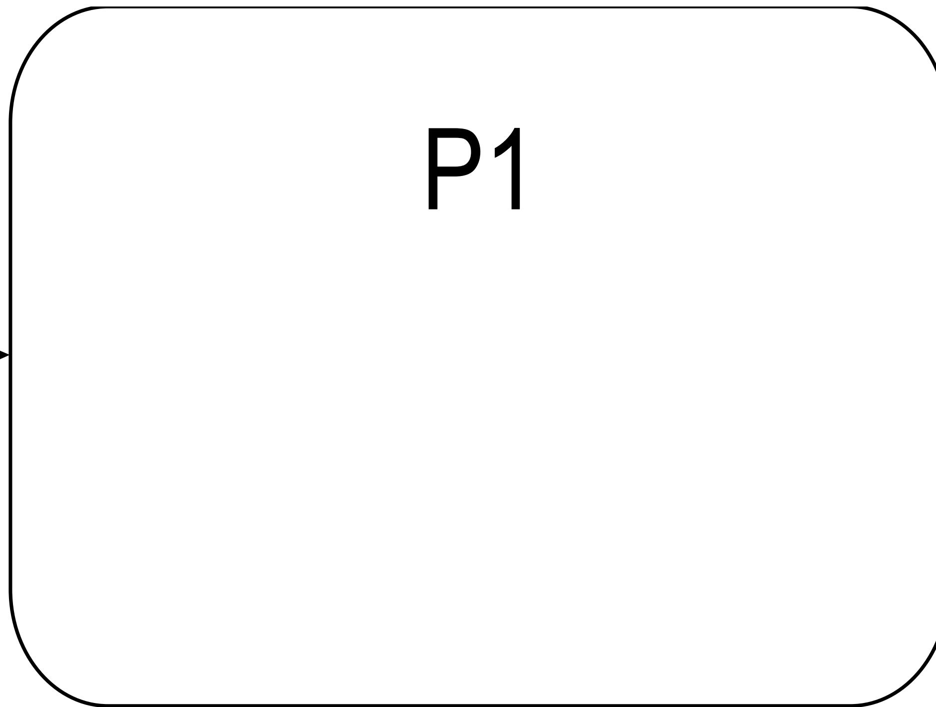
Why are C compilers allowed to remove the calls?

C89/90/99/11



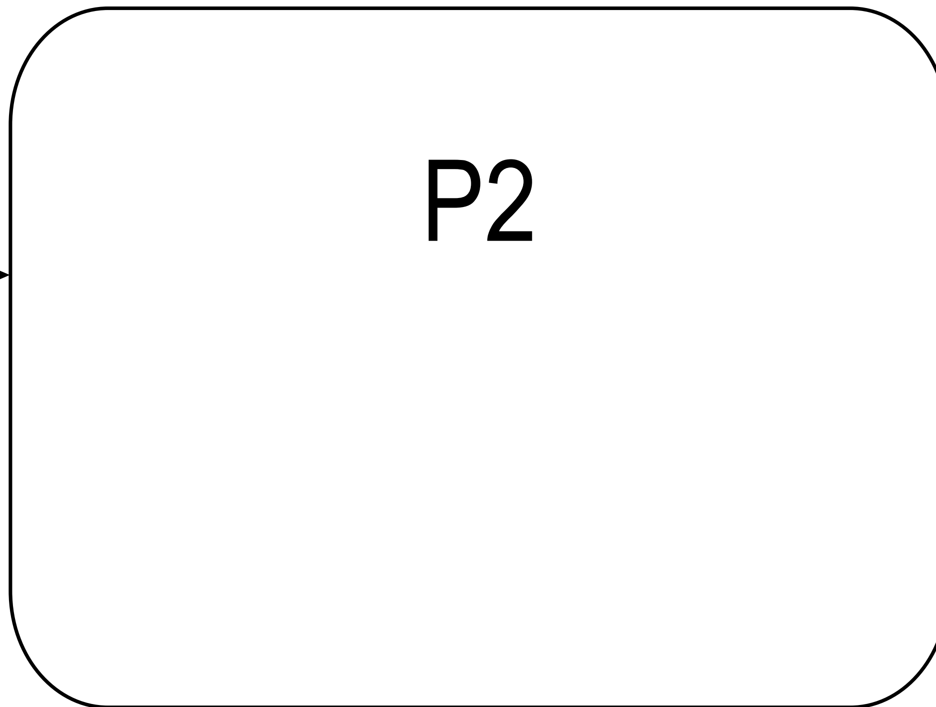
"An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced"

I1



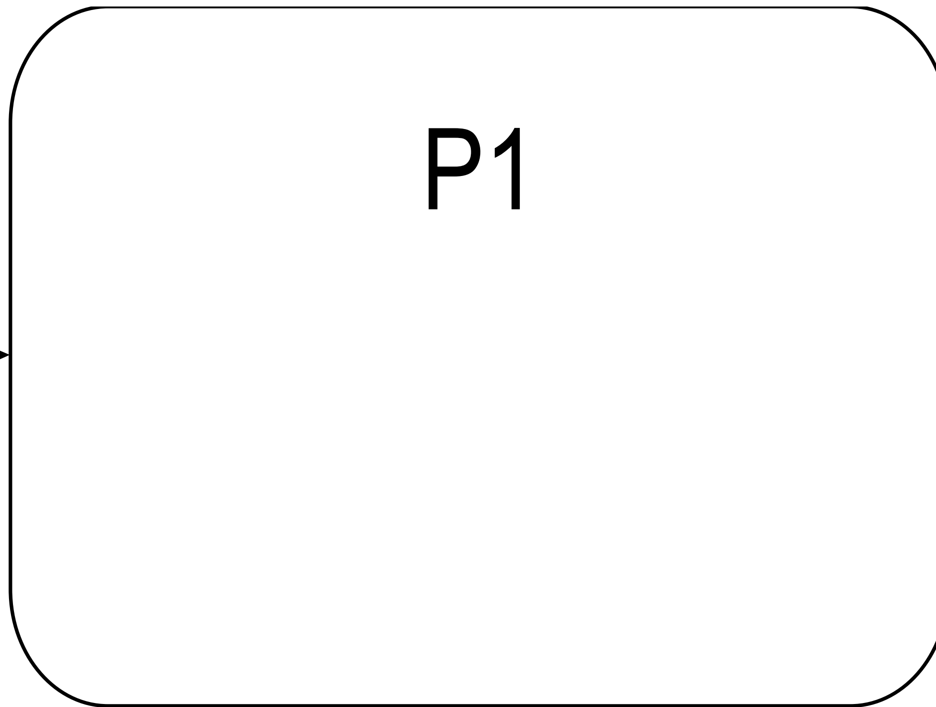
O1

I2



O2

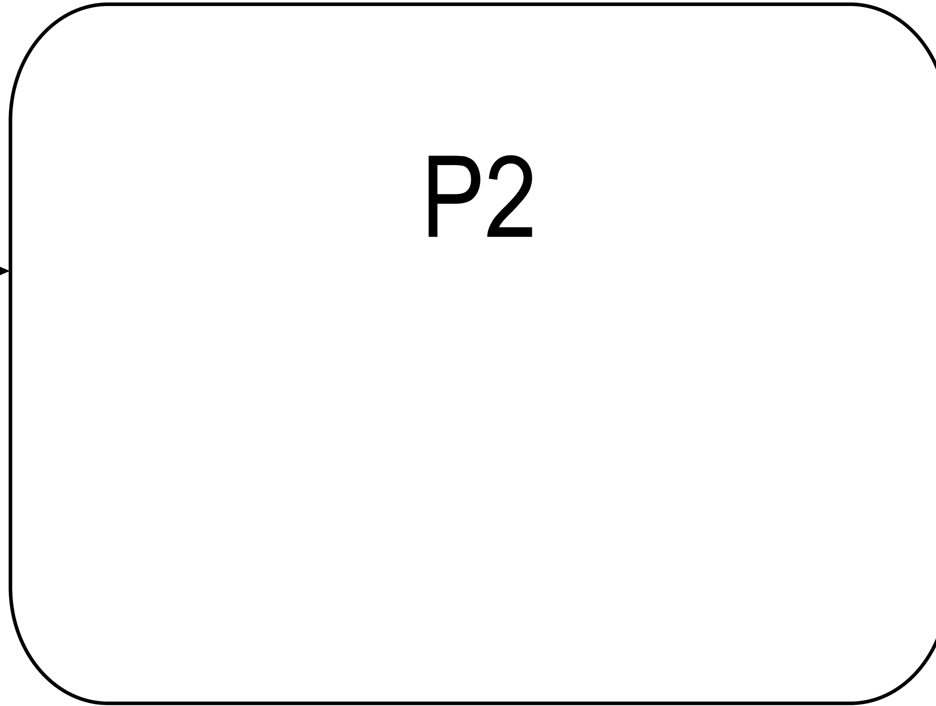
I1



O1

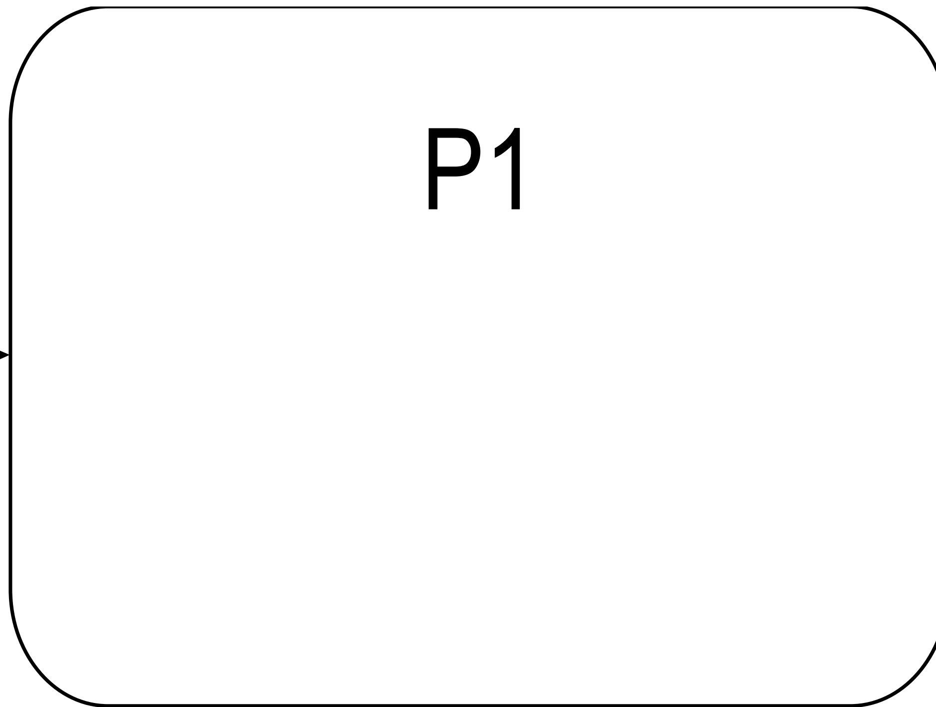
||

I2



O2

I1

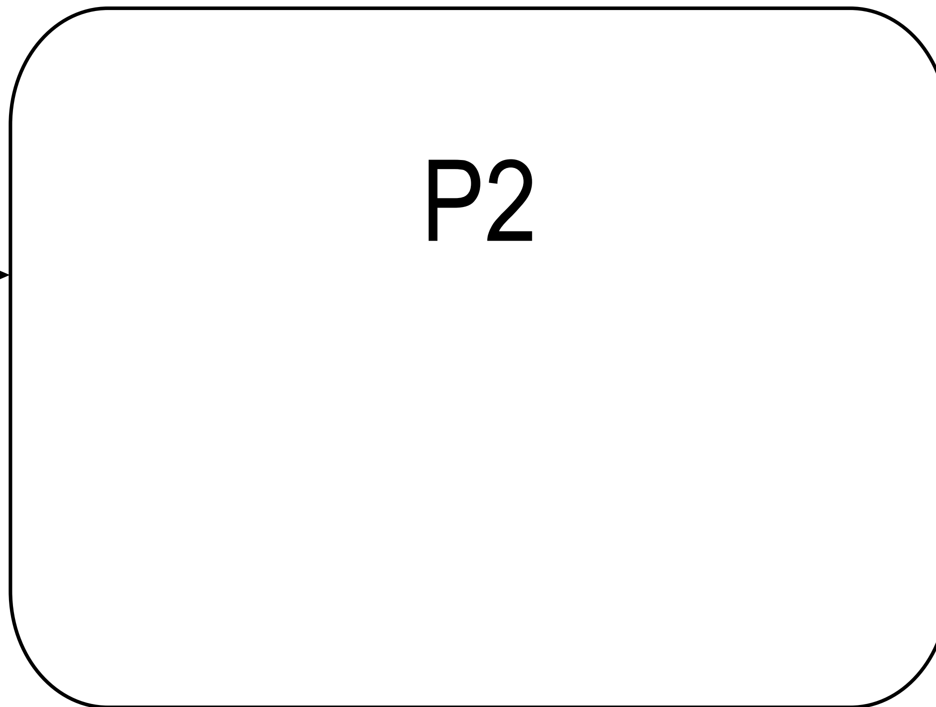


O1

||

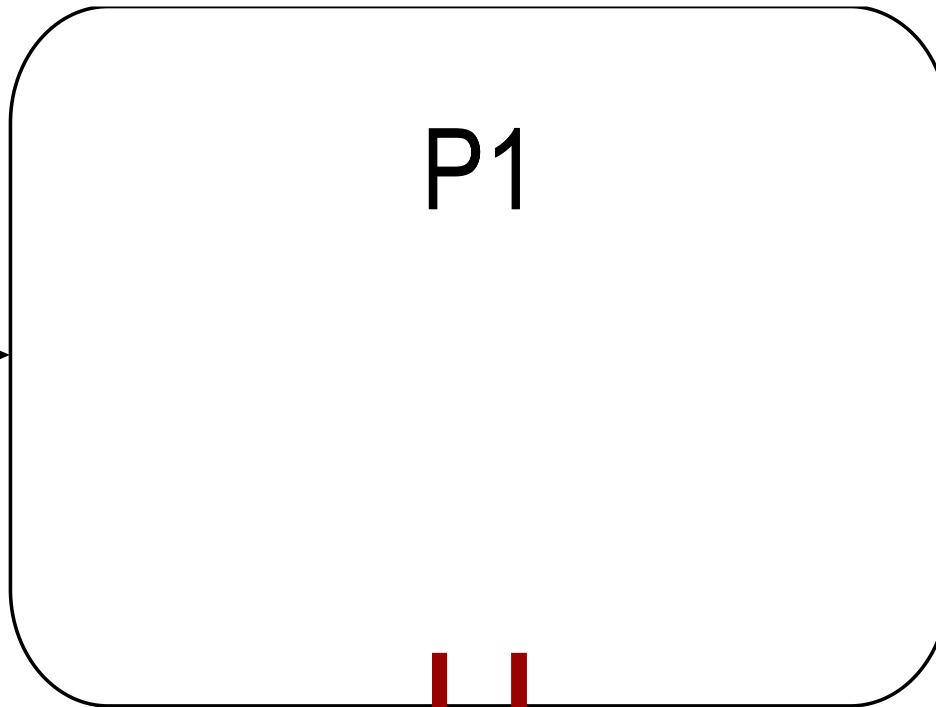
||

I2



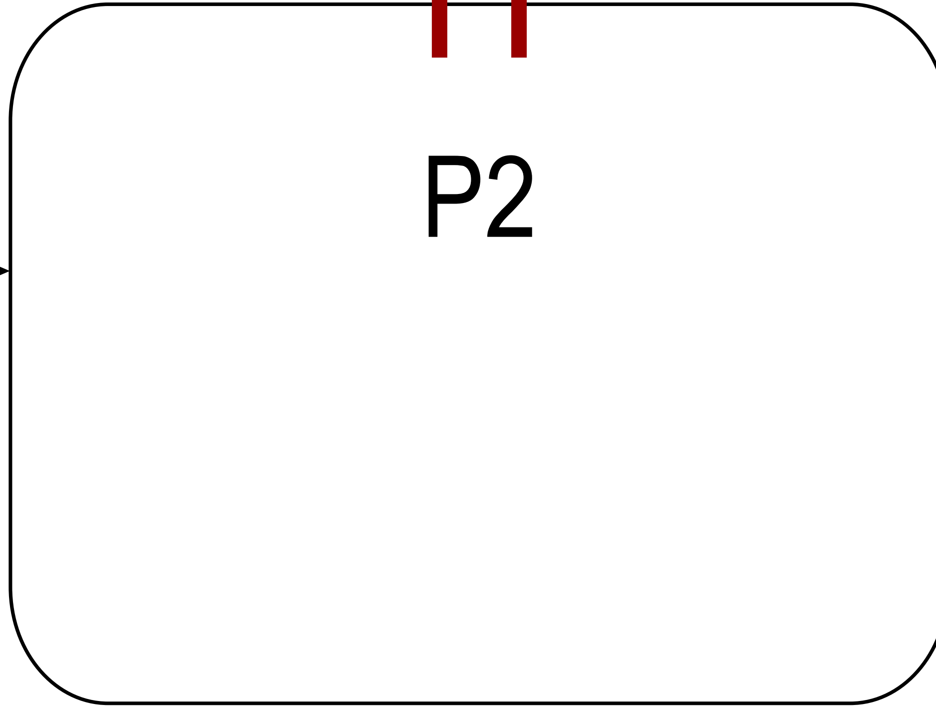
O2

I1



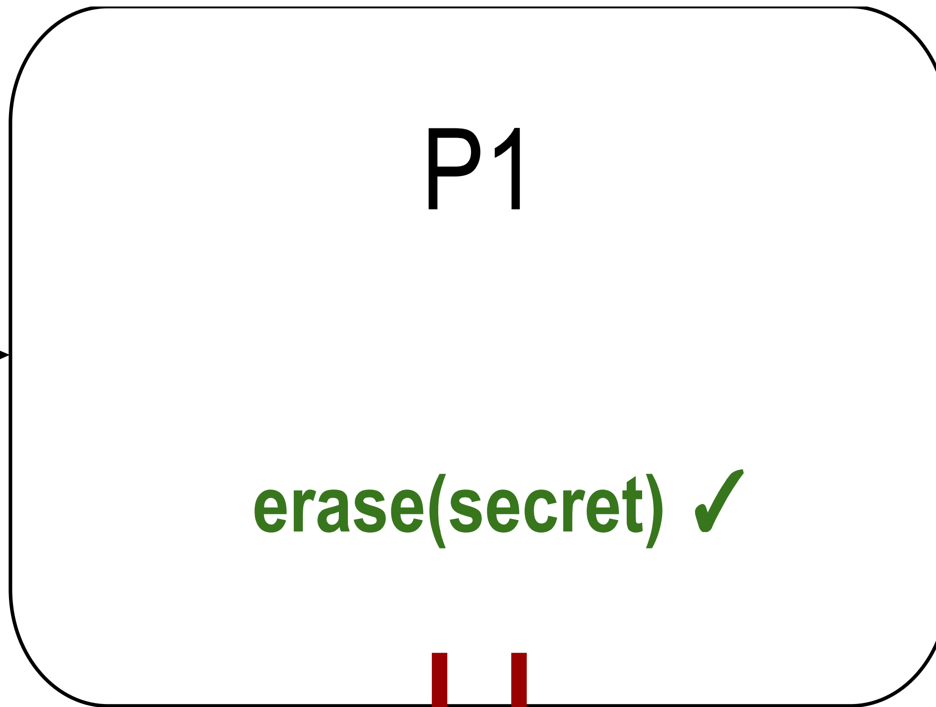
O1

I2



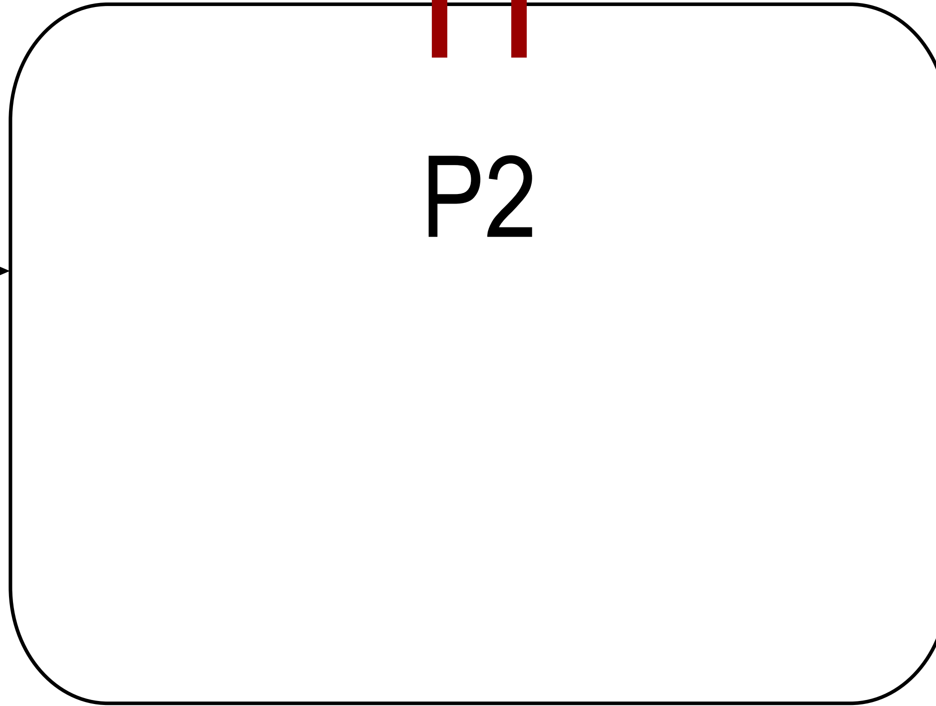
O2

I1



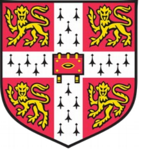
O1

I2



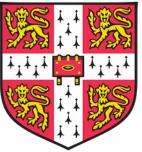
O2

Takeaway message (1)



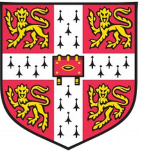
I. C standard is not suited to express security guarantees relying on controlling side effects of code

Side Effects in Cryptography



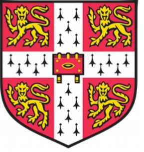
- Side channels:
  - Examples: timing, power, energy, EM
- Hardening techniques:
  - Bit scattering
  - Fault injection (e.g., rowhammer)
- See paper for more

Question 2



How do programmers attempt to control side effects today; and are they successful?

Talk outline

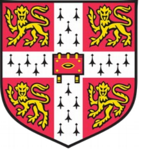


- Compiler Optimizations and Side Effects
- Example: constant-time choose
- Proposed Solution and Evaluation
- Conclusion

```
u32 choose(bool condition, u32 TrueVal, u32 FalseVal)
{
	if ( condition )
		return TrueVal;
	else
		return FalseVal;
}
```

Goal: for all inputs, execution time is the same
=> resistant to timing side channels

Constant-time requirements



1. No branching on sensitive data

```
u32 choose(bool condition, u32 TrueVal, u32 FalseVal)
{
	if ( condition )
		return TrueVal; // instruction evicted from I-caches
	else
		return FalseVal; // instruction loaded in I-cache
}
```

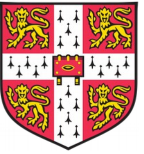
```
u32 choose(bool condition, u32 TrueVal, u32 FalseVal)
{
	if ( condition )
		return TrueVal; // instruction evicted from I-caches
	else
		return FalseVal; // instruction loaded in I-cache
}
```

```
u32 choose(bool condition, u32 TrueVal, u32 FalseVal)
{
        if ( condition )
                return TrueVal; // instruction evicted from I-caches
        else
                return FalseVal; // instruction loaded in I-cache
}
```

```
u32 choose(bool condition, u32 TrueVal, u32 FalseVal)
{
        if ( condition )
                return TrueVal; // instruction evicted from I-caches
        else
                return FalseVal; // instruction loaded in I-cache
}
```

```
u32 choose(bool condition, u32 TrueVal, u32 FalseVal)
{
        if ( condition )
                return TrueVal; // instruction evicted from I-caches
        else
                return FalseVal; // instruction loaded in I-cache
}
```

Constant-time requirements



1. No branching on sensitive data

2. Same data access pattern for all inputs

```
u32 choose(bool condition, u32 TrueVal, u32 FalseVal)
{
        if ( condition )
                return TrueVal; // TrueVal evicted from D-caches
        else
                return FalseVal; // FalseVal loaded in D-cache
}
```

```
u32 choose(bool condition, u32 TrueVal, u32 FalseVal)
{
        if ( condition )
                return TrueVal; // TrueVal evicted from D-caches
        else
                return FalseVal; // FalseVal loaded in D-cache
}
```

```
u32 choose(bool condition, u32 TrueVal, u32 FalseVal)
{
	if ( condition )
		return TrueVal; // TrueVal evicted from D-caches
	else
		return FalseVal; // FalseVal loaded in D-cache
}
```

```
u32 choose(bool condition, u32 TrueVal, u32 FalseVal)
{
	if ( condition )
		return TrueVal; // TrueVal evicted from D-caches
	else
		return FalseVal; // FalseVal loaded in D-cache
}
```

```
u32 choose(bool condition, u32 TrueVal, u32 FalseVal)
{
	if ( condition )
		return TrueVal; // TrueVal evicted from D-caches
	else
		return FalseVal; // FalseVal loaded in D-cache
}
```

```
static
u32 ct_choose(bool condition, u32 TrueVal, u32 FalseVal)
{
        s32 b = 0 - condition;
        return ( TrueVal & b )  |  ( FalseVal &  ~b) ;
}
```

```
static
u32 ct_choose(bool condition, u32 TrueVal, u32 FalseVal)
{
        s32 b = 0 - 0; // b = 0, ~b = 0xFFFFFFFF
        return ( TrueVal & 0 ) | ( FalseVal & 0xFFFFFFFF) ;
}
```

```
static
u32 ct_choose(bool condition, u32 TrueVal, u32 FalseVal)
{
	s32 b = 0 - 0; // b = 0, ~b = 0xFFFFFFFF
	return ( TrueVal & 0 ) | ( FalseVal & 0xFFFFFFFF) ;
}
```

```
static
u32 ct_choose(bool condition, u32 TrueVal, u32 FalseVal)
{
        s32 b = 0 - 0; // b = 0, ~b = 0xFFFFFFFF
        return ( TrueVal & 0 ) | ( FalseVal & 0xFFFFFFFF) ;
}
```

```
static
u32 ct_choose(bool condition, u32 TrueVal, u32 FalseVal)
{
        s32 b = 0 - 0; // b = 0, ~b = 0xFFFFFFFF
        return ( TrueVal & 0 )  | ( FalseVal & 0xFFFFFFFF) ;
}
```

```
static
u32 ct_choose(bool condition, u32 TrueVal, u32 FalseVal)
{
	s32 b = 0 - 0; // b = 0, ~b = 0xFFFFFFFF
	return ( TrueVal & 0 ) | ( FalseVal & 0xFFFFFFFF) ;
}
```

condition = false, return FalseVal

```
static
u32 ct_choose(bool condition, u32 TrueVal, u32 FalseVal)
{
	s32 b = 0 - 0; // b = 0, ~b = 0xFFFFFFFF
	return ( TrueVal & 0 ) | ( FalseVal & 0xFFFFFFFF) ;
}
```

$clang-3.0 -O[0,1,2,3]

```
u32 ct_choose(bool condition, u32 TrueVal, u32 FalseVal)
{
	s32 b = 0 - condition;
	return ( TrueVal & b ) | ( FalseVal &  ~b) ;
}
```

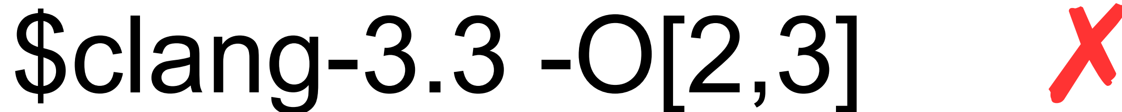
$clang-3.0 -O[1,2,3] ✗

```
[static]
u32 ct_choose(u32 condition, u32 TrueVal, u32 FalseVal)
{
        u32 b = −(u32) ((condition|−condition)>>31);
        return ( TrueVal & b )  |  ( FalseVal & ~b) ;
}
```

```
[static]
u32 ct_choose(u32 condition, u32 TrueVal, u32 FalseVal)
{
        u32 b = −(u32) ((condition|−condition)>>31);
        return ( TrueVal & b )  |  ( FalseVal & ~b) ;
}
```

$clang-3.0 -O[0,1,2,3]

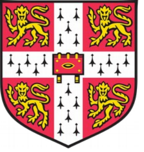
```
[static]
u32 ct_choose(u32 condition, u32 TrueVal, u32 FalseVal)
{
        u32 b = −(u32) ((condition|−condition)>>31);
        return ( TrueVal & b )  |  ( FalseVal & ~b) ;
}
```



```
[static]
u32 ct_choose(u32 condition, u32 TrueVal, u32 FalseVal)
{
        u32 b = −(u32) ((condition|−condition)>>31);
        return ( TrueVal & b )  |  ( FalseVal & ~b) ;
}
```

Observation: newer versions of compilers may be less reliable than older versions for controlling side effects

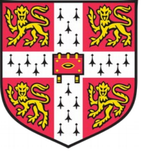
Takeaway message (2)



I. C abstract standard is not suited to express security guarantees relying on controlling side effects of code

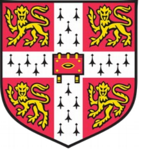
II. Developers are left fighting the compiler through obfuscation to control side effects. This must stop: we must make C compilers our allies, not our enemies.

Talk outline



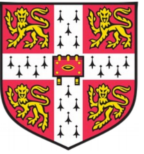
- Compiler Optimizations and Side Effects
- Example: constant-time choose
- Proposed Solution and Evaluation
- Conclusion

Proposed Solution



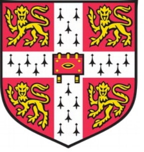
- Adding support into the compilers

Proposed Solution



- Adding support into the compilers
- Expose support to developers explicitly
  - Examples: pragma, annotations, flags, attributes, new functions, etc

Proposed Solution

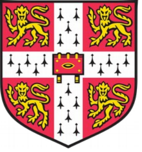


- Adding support into the compilers
- Expose support to developers explicitly

    - Examples: pragma, annotations, flags, attributes, new functions, etc

    - Better communication has improved performance (e.g., SIMD attributes, *restrict* keyword), so will it help control side effects

Proposed Solution

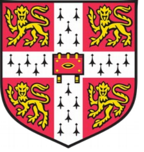


- Adding support into the compilers
- Expose support to developers explicitly

  - Examples: pragma, annotations, flags, attributes, new functions, etc

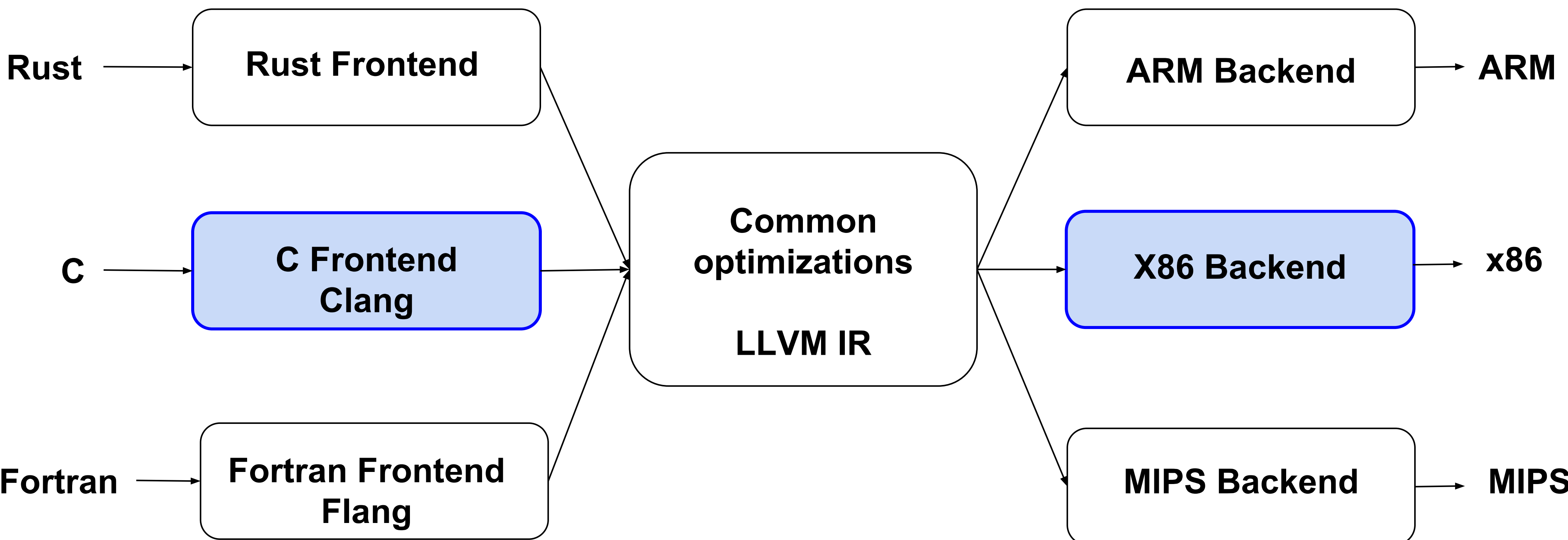
  - Better communication has improved performance (e.g., SIMD attributes, *restrict* keyword), so will it help control side effects
- EuroLLVM 2018: general support for extensions that better express programmer intent

Implementation

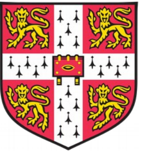


- Two steps towards our goal:
  - Secret erasure for stack and registers: see paper
  - Constant-time choose()

- Clang/LLVM framework

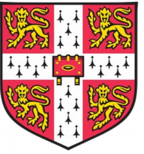

Rust
Rust Frontend
C
C Frontend
Clang
Fortran
Fortran Frontend
Flang
Common
optimizations
LLVM IR
ARM Backend
ARM
X86 Backend
x86
MIPS Backend
MIPS

Constant-time choose()



```
Type __builtin_ct_choose(
                        bool cond,
                        Type TrueVal,
                        Type FalseVal);
```

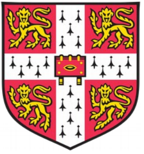
Constant-time choose()



```
Type __builtin_ct_choose(
                    bool cond,
                    Type TrueVal,
                    Type FalseVal);
```

OpenSSL defines 37 functions

Runtime overhead



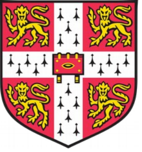
Runtime overhead (# CPU cycles)

1
0.75
0.5
0.25
0

X25519
Montgomery ladder

OpenSSL choose
__builtin_ct_choose

Takeaway message (3)

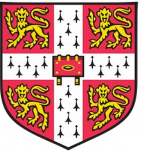


I. C abstract standard is not suited to express security guarantees relying on controlling side effects of code

II. Developers are left fighting the compiler through obfuscation to control side effects. This must stop: we must make C compilers our allies, not our enemies.

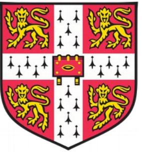
III. Explicit compiler support will empower developers

Conclusion



- C standard not appropriate to control side effects
- Arms race between compiler writers and developers/cryptographers must stop
- Compiler support and expose it to developers
- Ton of work, with real impact
- Long journey: compiler, developers, OS, hardware (e.g., power side effects)

Thanks!



Questions?

Reference implementations:

https://github.com/lmrs2/ct_choose

https://github.com/lmrs2/zerostack

Laurent Simon
l.simon@samsung.com
https://sites.google.com/view/laurent-simon/