ROUNDING POINTERS
TYPE SAFE CAPABILITIES WITH C++ META PROGRAMMING

Alexander Warg, Adam Lackorzynski
MOTIVATION
OBJECT CAPABILITIES

Protected / Secure references
Local naming for components and services
Fine-grained access control
Well known security properties

C++
Templates as powerful concept
Full control over memory management
Object oriented
CAPABILITIES ARE INTEGERS

No dynamically allocated C++ objects
Treat capabilities as values (not objects)
Use static C++ type checks
No extra indirection for operations
Interfaces are C++ classes

```cpp
class Remote_iface : public Kobject { ... };
```

Capabilities are smart pointers

```cpp
Cap<Remote_iface> c = ...
c->method(args);
```

Composition of interfaces via C++ inheritance

```cpp
class Specific_iface : public Remote_iface { ... };
```

Abstract interfaces $\rightarrow$ dynamic type information desirable
EDGE OF THE C++ STANDARD

Overloaded '->' operator of Cap must return a pointer

→ 'this' pointer carries the capability (integer)

Implicit conversions on 'this' pointer

→ blurry capabilities

→ invocations on non-existing objects
META PROTOCOL

Dynamic type information from objects (IPC)
Enables 'cap_dynamic_cast'

BLURRY CAPABILITIES ARE USEFUL

The blur provides type information for composed objects
Dynamic type information provides initial offset
INTUITIVE, POINTER-LIKE C++ INTEGRATION
including explicit and implicit type conversion

STATIC AND DYNAMIC C++ TYPE CHECKING

EFFICIENT REPRESENTATION
template class wrapping a native integer

EFFICIENT METHOD INVOCATION
capability as 'this' pointer

ADHERE TO C++ STANDARD
capability as 'this' pointer depends on sane implementation
```cpp
l4_cap_idx_t Kobject::this_to_cap() const
{
    return reinterpret_cast<l4_cap_idx_t>(this) & BLUR_MASK;
}

int Remote_iface::method(int arg)
{
    L4::Ipc::Io_stream s(l4_utcb());
    s << L4::Opcode(Remote_iface::OP_METHOD) << arg;
    return s.call(this_to_cap());
}
```
template< typename T, typename S >
Cap<T> cap_static_cast(Cap<S> const &s)
{
    // let the compiler check the conversion
    (void)static_cast<T*>(reinterpret_cast<S*>(0));
    return Cap<T>{s.cap_idx()};
}
template< typename T, typename S >
Cap<T> cap_dynamic_cast(Cap<S> const &s)
{
    if (meta_supports(s, T::INTERFACE_ID))
        return Cap<T>(s.cap_idx());
    else
        return Cap<T>::INVALID;
}