Rounding Pointers – Type Safe Capabilities with C++ Meta Programming

Alexander Warg, Adam Lackorzynski
Technische Universität Dresden
Department of Computer Science
Operating Systems Group
{warg,adam}@os.inf.tu-dresden.de

ABSTRACT

Recent trends in secure operating systems indicate that an object-capability system is the security model with pre-eminent characteristics and practicality. Unlike traditional operating systems, which use a single global name space, object-capability systems name objects per protection domain. This allows a fine-grained isolation of the domains and follows the principle of least authority.

Programming in such an environment differs considerably from traditional programming models. The fine-grained access to functionality requires a programming environment that supports the programmer when using a capability system. In this paper, we present an object-oriented framework that uses the C++ programming language to offer a framework for building and using operating-system components and applications.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classification—*C++*; D.4.6 [**Operating Systems**]: Security and Protection; D.1.5 [**Software**]: Object-oriented Programming

General Terms

Security; Languages; Design

1. INTRODUCTION

Currently, operating systems seem to move to a capability-based security model. For example, the University of Cambridge, supported by Google, presented a capability extension for UNIX [17]. Other research towards highly secure systems is based on small capability-based operating-system kernels [7].

Strictly applying the object-capability model can help solving typical security flaws, such as the confused deputy problem, and supports the principle of least authority (POLA). Other work tries to support the programmers in using capability-based systems by specific programming languages or language extensions [14, 10]. Nevertheless, current operating-system software is typically written in C or C++, and constitutes a huge value with device drivers and functionality that is worth to be adopted.

The contribution of this work is a lightweight and efficient integration of the object-capability model into the C++ language, including interface types and inheritance. The work does not require any extra tools or compiler changes and aids the programmer with intuitive programming patterns.

In the next section we give an overview over the object-capability model and object-oriented programming in C++, as well as discuss related work. In Section 3 we describe the goals of this work that are deduced thereof, before we move on to discuss our implementation in Section 5. Section 6 provides a discussion about the efficiency of our solution and Section 7 concludes the work.

PLOS '11, October 23, 2011, Cascais, Portugal.

2. OBJECT-CAPABILITY MODEL

Object-oriented programming is based on the idea of objects, references to objects, and the model of carrying out operations on objects by sending *messages* on references. The object-capability [1, 12] model as a computer security model takes these ideas and asserts some requirements on them, to achieve goals, such as the principle of least authority and privilege separation.

As described in [11, 12] capability systems are superior to systems using access control lists, and provide a more intuitive abstraction for programmers.

There are two flavors of object-capability systems: purely language-based approaches [14, 10] and capability-based operating systems [13, 7, 8]. In the remainder of this work we shall focus on the capability-based operating systems and in particular systems with kernel-protected capabilities.

2.1 Capability-Based OS

Looking at implementations of current operating-systems, one can find a lot of object-oriented implementations written in languages, such as C or C++. C++ natively supports the object-oriented programming paradigm, provides good integration with C, and allows the same control over memory management as C does. Hence, we favored C++ as the implementation language for a capability-based operating system.

The central part of our architecture is a small and secure kernel [3] running in privileged processor mode. The main purpose of the kernel is to provide strong isolation between different protection domains and an highly efficient communication mechanism to support a fine-grained object-model based on protection domains. Our kernel provides a purely object-capability based API. In particular, services provided

by the small kernel are available as objects implemented in the kernel's protection domain and the kernel provides an efficient capability-based communication mechanism to implement objects in different protection domains. Results presented in [9] and [6] influenced the API of our kernel, as well as motivate the goals described in Section 3. Before we move on to the goals we shall briefly describe the properties of references to objects in terms of C++ and our object-capability kernel.

2.2 References in Detail

C++ uses pointers as references to objects. In contrast, our small kernel provides kernel-protected references to objects in different protection domains. Protection domains are either the kernel's domain or isolated user processes comprising a virtual memory space and an object-space (also called capability table). Programs refer to capabilities by an index into their local capability table. This means, they use integral values (*capability selectors*) to reference remote objects in a different protection domain.

C++ does not provide strict type safety and does not adhere to the rules for obtaining references as defined in [1]. In particular, a C++ pointer can be synthesized out of thin air, can be distributed among objects in global variables, and there are operations that do not require an explicit reference to an object, such as global functions. However, our small kernel strictly applies the rules of an object-capability system with the granularity of protection domains and objects implemented in different protection domains. Objects implemented in the kernel's domain also strictly adhere to these rules.

Considering the operations on objects, object-oriented programming provides the mechanism of abstract interfaces that declare the operations that can be performed on an object. C++ has the notion of *pure virtual functions* defined in a class declaration. This provides the same abstraction from a particular implementation of an object as the object-capability model provides with the messages on references. Object-oriented languages usually also provide the mechanism of inheritance to derive more specialized interfaces and implementations from one or more base interfaces, including a set of type conversion rules on typed references to objects.

Based on the properties of object-capability systems and object-oriented languages, and considering the work done in the area of microkernel-based systems since [9], we define the goals of a seamless integration between those two worlds in the next section.

3. GOALS

A general goal of this work is to provide a C++ representation of capabilities that does not introduce per-design overheads, such as the use of dynamically allocated proxy objects.

Efficient representation of capabilities. Capabilities shall be as efficient as C++ pointers when they are used and passed around inside a single application. This means assignment to local variables and to attributes of objects, as well as passing as parameters must not introduce extra operations compared to normal pointers.

Efficient invocation of operations on objects. The C++ representation of capabilities shall not add extra overhead, such as additional levels of indirection, to the message passing that is used for invoking operations.

C++ static typing rules. Capabilities shall behave like pointers to objects, with respect to the type safety and type-conversion rules. This means capabilities shall carry type information as typed C++ pointers do, and the C++ compiler shall enforce the same conversion and assignment rules as on pointers. This allows for extra robustness and static error catching at compile time.

Dynamic type information. Objects in our system shall carry dynamic (runtime) type information comparable to that provided by C++.

Comparable cast operations. As capabilities shall be typed, there need to be explicit conversion operations. The conversion operations working on the static type of a capability (e.g., `cap_static_cast` and `cap_reinterpret_cast`) shall have semantics and complexity compatible to their C++ versions for pointers. Conversions working with the dynamic type of an object (comparable to `dynamic_cast`) may have additional overhead, as do dynamic casts in C++.

C++ Standard Compliance. A goal of this work is to stay within the defined behavior of the C++ Standard [2] as far as possible. However, some parts are based on implementation-defined behavior of the specific compiler.

4. RELATED WORK

There are two important industry implementations of distributed object-oriented systems: COM and CORBA. While CORBA explicitly targets network transparent object models, COM is designed to support object-oriented systems locally, hence COM is more similar to our scenario. The C++ language bindings for both approaches are different. CORBA requires no direct mapping of the interface hierarchy to a C++ class hierarchy, this means no implicit conversions can be applied by the C++ compiler. COM interfaces are mapped to a C++ class hierarchy and support static C++ conversions. Nevertheless, both solutions use local proxy objects with virtual functions to represent remote objects. These proxy objects add two problems in our context: the proxy objects must be dynamically allocated and at least one additional layer of indirection is introduced.

The low-level OS framework shall avoid dynamic allocation, because the framework shall not depend on any specific memory management framework. The goal of a C++-based type checking and type conversion system requires a compatible mapping of the interface hierarchy to a C++ class hierarchy. We try to avoid any extra level of indirection to achieve highly efficient operations on objects.

5. IMPLEMENTATION

The foundation for our language mapping is provided by two C++ concepts, namely operator overloading and template meta programming. These concepts can be used to implement a pattern known as C++ smart pointers [4, 16]. Smart pointers are generic data types, which provide the syntax and semantics of normal C++ pointers and hide the internal implementation.

5.1 Basic Language Mapping

The goals described in Section 3 lead to a language mapping where interfaces of remote objects are defined using C++ classes and where capabilities are comparable to pointers to objects of those classes. The syntax for using remote objects shall be as simple as using operations on a locally implemented C++ object.

5.1.1 References

We use a smart-pointer-like template class to represent capabilities in C++. The `Cap` template has full value semantics and wraps our capability selectors. In practice, declaring a reference to a remote object looks as follows: `Cap<Addressbook> a`, where `Addressbook` is the type of the object to be referenced. This is similar to the smart pointers described in [4]. The simplified skeleton of the typed capability template class is:

```
template< typename T >
class Cap
{ private: cap_idx_t _c; };
```

The data member `_c` is the only data member and has the same storage size as a native C++ pointer. C++ allows such data types to be handled as efficient as simple native integral data types with respect to assignment and parameter passing.

5.1.2 Operations

Next, we need to enable operations on remote objects, just as calling methods on a local object. The syntax for invoking operations on objects, referenced by a C++ pointer, is using the arrow operator "`->`" followed by a call operator "()" (i.e. `obj->func()`). Fortunately, C++ allows overloading of the arrow operator, as a key element for smart-pointer implementations. C++ defines that overloaded arrow operators are consecutively resolved until the resulting value has the type *pointer to object*. This means, when invoking a non-virtual member function, the function call is resolved using the static class type. All the given parameters and additionally the implicit `this` pointer are passed to the function's implementation.

Looking at the `Cap` class, it seems unclear how to provide an arrow operator that returns a pointer, because we have only the capability selector available. The solution is to use the C++ reinterpret cast to convert the capability selector to a pointer of type `T`, and return the pointer as result of the arrow operator:

```
T *operator -> () const
{ return reinterpret_cast<T*>(_c); }
```

The returned pointer does not hold a valid address of a C++ object, rather it carries the capability selector that is required for the kernel call. The invalid pointer implies that we have no instances of our stub class `T`, and hence `T` should be an *empty class type*, without any data members or virtual functions. To prevent instantiation of stub classes we add a protected constructor to those classes. A method call to our interface translates to a statically resolved call to a function deduced by the type `T`, the given identifier for the function's name, and the parameter types. This interface stub implementation is called with the capability selector as `this` pointer and is responsible for marshalling (see [5]) the parameters into a message, converting the `this` pointer back to a capability selector, and passing the message to the remote object using the kernel mechanism.

5.1.3 Inheritance and Type Conversion

Because we study interfaces of remote objects, we have to consider abstract interfaces only. Inheritance of interfaces provides a mechanism to take a base interface (a set of operations) and add more operations to the derived interface. Another flavor is multiple inheritance that enables the combination of multiple base interfaces into a common interface.

C++ provides inheritance as part of the language and deploys the inheritance relations in the form of implicit and explicit type conversion rules. For example, we can derive the class `Addressbook` from the class `Table`:

```
class Table
{ public: int get_num_rows(); };

class Addressbook : public Table
{ public: int find_by_name(char const *n); };
```

This means, a pointer to `Addressbook` can be implicitly (automatically) converted into a pointer to type `Table`. The other direction, from `Table` to `Addressbook`, has to be done using explicit cast expressions. C++ cast expressions exist in different flavors and may fail at compile time or at run time in the case types are not convertible.

The type system of C++ is used to determine the available operations on objects of a certain class, as well as to provide static and dynamic type checking. These advantages of the C++ type system shall also apply to the interfaces of our remote objects. As we already define the interface of a remote object using C++ stub classes, we can also use the C++ inheritance features to add inheritance relations to the types of our remote objects.

The inheritance relation shown in the previous example allows to call the `get_num_rows` method on a capability of type `Addressbook`, as C++ implicitly converts the `this` pointer for the function call to type `Table*`. To provide pointer-like assignment semantics to the `Cap` class we have to add a set of template conversion constructors that statically enforce the same conversion rules as applied for pointers.

For the case of single inheritance this usually works straight forward, however the C++ standard allows static cast operations to change the value of the pointer, even though they might be implicit. This *pointer offsetting* results from Section 4.10 §3 of the ISO C++ standard [2]. In our case, where we abuse the pointer to contain our capability selector this property may lead to a change in the capability selector and render the selector invalid or referring to a different object. Using multiple inheritance, this property of type conversions definitely becomes a problem, because the C++ standard also contains a rule that two different object of the same type must never have equal addresses.

To overcome this problem we use the property that our capability selectors, as defined by the kernel's API, do not use the full value range of a C++ pointer. In particular, the selector values are integers that are a multiple of 4096, which means the least significant twelve bits of the integer representation are zero. Assuming, that the pointer changes engendered by the type conversions are in a range of one to eight bytes (particular value depends on the compiler) per interface in an inheritance hierarchy, we can use a simple bit masking operation to align our capability selectors back to multiples of 4096. The stub code has to do this masking when an abused `this` pointer is transformed back to a capability selector. A more detailed view on the stub methods and the translation to messages shall follow in Section 5.3. The conversion we use to put our capability selectors into the `this` pointer are robust according to the C++ standard [2] Section 5.2.10, in particular the note in §4 provides confidence for our bit masking.

5.1.4 Explicit Type Conversion

As described before, we use C++ classes to represent the interfaces of remote objects, and we use the inheritance features of C++ to compose and refine interfaces. The implementation supports implicit type conversions that are either

done during dereference using the arrow "`->`" operator or during assignment or initialization using template conversion constructors or assignment operators. The conversion constructors and assignment operators must assert the validity of a conversion according to C++ pointer rules.

Explicit conversions in C++ are done using cast expressions, for example `static_cast<T>(v)`. These expression do not work for our capabilities, because the classes representing the capabilities as smart pointers have no inheritance relation on their own. For example, `Addressbook` is derived from `Table`, however, `Cap<Addressbook>` is unrelated to `Cap<Table>`.

Hence, support for type conversion has to be implemented manually as a set of template functions, for example `Cap<T> cap_static_cast<T>(Cap<Tr> v)`. The implementation of these cast functions has to ensure that a pointer to type `Tr` is convertible to a pointer to type `T` with the corresponding C++ cast expression. An exception of this simple and straight forward approach for type conversion is the dynamic cast expression that we shall elaborate on in the following section.

5.2 Dynamic Types

C++ provides the dynamic cast, a robust type conversion based on the dynamic type of objects. The dynamic type of an object is the most derived type `D` of an object instance, even if currently referenced by a pointer to a base class `B` of `D`. C++ requires the types `B` and `D` to be polymorphic types (i.e., classes equipped with some virtual functions). The dynamic cast is robust in the sense that it yields a `NULL` pointer in the case where the dynamic type of the object is not of type `D` or a type derived from `D`.

Projecting this to remote objects in our object-capability system is no longer trivial. As our interface classes have to be *empty classes* as described in Section 5.1.2, they cannot be polymorphic and thus cannot carry dynamic C++ type information. Furthermore, the dynamic type of a remote object is known solely by the object itself and therefore acting on the dynamic type must be done sending messages to the object, otherwise we would break the rules for object capabilities.

Nevertheless, the dynamic type of a remote object could be useful, especially in the case when capabilities to the generic type `Kobject` are received and a robust conversion to some "useful" type is considered. To provide such dynamic type information we developed a `Meta` interface that shall be implemented by every remote object in our system. The `Meta` interface provides operations to support conversion of a capability of type `B` to a capability of type `T` and applies the conversion rules and information located in the remote objects implementation.

5.3 Operations on Remote Objects

So far, we considered the interfaces and their inheritance and left out the task of transforming C++ method calls to object-capability based message passing. An inherent part of this task is to assign an ID (opcode) to each operation of an interface. For simple interfaces without inheritance this can be as trivial as incrementally assigning integers starting from zero to each operation of the interface.

Introducing single inheritance, the task to assign opcodes to operations becomes slightly more complicated. Opcodes for operations of a derived interface must not collide with opcodes assigned to operations of the base interface, however, opcodes used by the base interface must stay compatible with the implicit type conversions that are possibly done during C++ method invocation. One solution is to also incrementally assign integer values starting at the opcode of the last operation in the base interface plus one.

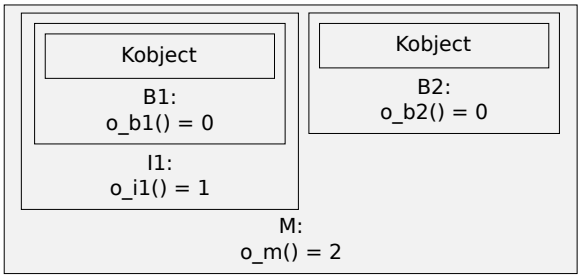

Figure 1: Typical memory layout of class `M` with multiple inheritance. You can see two single inherited hierarchies: [Kobject←B1←I1] and [Kobject←B2]. Both are combined by deriving `M` from `I1` and `B2`. The interfaces have some example operations `o_b1`, `o_b2`, `o_i1`, and `o_m`. The number after each operation shows the assigned opcodes. The arrangement of the boxes from left to right depict the pointer offsets for the sub-objects of `M`.

Going forward to multiple inheritance, to compose a new interface out of multiple independently defined base interfaces, raises a general problem. The opcodes of the independent interfaces are assigned individually and are likely to collide. One way to resolve the collisions is to add static, unique identifiers to each interface. However, this requires a central unit, comparable to IANA, that assigns these interface IDs (IIDs). Another possibility is a dynamic assignment of IIDs at runtime and a negotiation protocol for clients to acquire the information about the IIDs used by a particular remote object.

The downside of the use of dynamically assigned IIDs is that they have to be stored alongside the capability selector in the `Cap` class. And even more tricky, the IID has to be used within the stub code for remote object invocation that runs with the restriction of getting the capability selector via the abused `this` pointer.

As our capability selectors use only a part of the value range used by C++ pointers we could use the remainder to store the interface ID, theoretically as much as 4096 for each individual object.

In practice this creates a collision between the pointer offsetting, mentioned in Section 5.1.3, and the use of the least significant bits of a capability selector for carrying vital data. And additionally, there arises the question about interface IDs and implicit casts to pointers to a base class when a method to a base class is invoked.

5.3.1 Combining C++ Semantics and Interface IDs

The aforementioned collision between pointer offsetting and interface IDs lead us to the idea that the pointer offsetting could be used for automatically calculating interface IDs within an inheritance hierarchy.

Taking a closer look on the pointer offsetting results in the following observations. As our interface classes must be empty classes (regarding data members) this usually results in a zero offset for single inheritance, which is absolutely acceptable when we use distinct opcodes for operations in derived interfaces. Multiple inheritance is, in particular, used to combine interfaces that may be also used stand alone, such interfaces typically inherit from the remote object base class `Kobject`. And this in turn leads to a pointer offsetting

other than zero, because different sub-objects of a C++ object that have the same type (Kobject) must have a different address even if they are empty, see Figure 1.

Based on this knowledge we can use the C++ compiler to assign interface IDs to the different interfaces that are combined with multiple inheritance. The implementation of a remote object has to use the values deduced from the pointer offsets as interface IDs and dispatch incoming messages to the implementation of the according sub-object. On the client side, the interface IDs manifest themselves in the least significant bits of the abused `this` pointer and need to be added to the message that is sent to a remote object. In the example in Figure 1, sub-hierarchy [Kobject←B1←I1] has zero offset and [Kobject←B2] has one byte offset (compiler implementation defined). This means, calling `M::o_b2()` needs to use one as IID, whereas calling `M::o_i1()` has to use a zero IID, otherwise the implementation of `M` could not distinguish between `M::o_b1()` and `M::o_b2()`.

Care has to be taken to represent the interface IDs in the implementation of the conversion constructors, assignment operators, and cast functions for the capability template class. The `Meta` interface has to provide information about the interface ID of a requested type, in order to implement correct semantics for the dynamic cast for capabilities.

6. DISCUSSION

To evaluate the results of our mapping of object capabilities to C++ primitives we discuss the overheads that are induced by our implementation of object capabilities in C++. We will neither discuss the performance of the message-passing mechanism provided by the kernel nor the costs induced by the parameter marshalling code. Instead, we refer you to results from [15] and [5], respectively. Nevertheless, we consider the performance of the message passing as a critical component and try to avoid any unnecessary overheads.

Section 5.3 describes the steps for carrying out operations on remote objects. We use the implicit `this` pointer provided by C++ to store the capability selector. The `this` pointer is usually stored in a register of the CPU. The conversion from a pointer to an integer value of equal size is usually reduced to a no-op. This means we have to add an extra bit masking instruction to mask the lower twelve bits of the `this` pointer. Additionally, we have to take the lower twelve bits of the `this` pointer and put them into the message to be transferred, which is in the case of our kernel done in a CPU register as well.

Implicit type conversion applied by the C++ compiler is based on the type conversion of C++ pointers and does not induce any additional overhead compared to C++ pointers. Also, explicit type conversions based on static types are equivalent to C++ pointer conversions. However, dynamic types of objects require additional operations on remote objects and hence add an overhead compared to C++ dynamic type conversions. In particular, we have to pay the penalty of message passing to the remote object to gain access to the dynamic type information necessary for the conversion.

7. CONCLUSION

As a result of this work we implemented a lightweight and efficient integration of object capabilities into a C++ framework, as a form of smart pointers. The resulting framework supports static C++-based typing of capabilities, including explicit and implicit type conversion. We also described a mechanism for providing dynamic type information for objects in our capability-based OS and use it to provide `cap_dynamic_cast<>` similar to `dynamic_cast<>`.

The implementation is mostly based on well defined behavior of the C++ language. The noteworthy exception to this is to use the `this` pointer as storage for our capability selectors. However, under the assumption of a reasonable compiler, we tried to achieve a robust implementation by carefully considering the pointer offsetting applied for type conversion.

8. REFERENCES

[1] Object-Capability Model. `http://en.wikipedia.org/w/index.php?title=Object-capability_model&oldid=422387511`.

[2] *ISO/IEC 14882: Programming languages — C++*, 1 ed. International Organization for Standardization, Geneva, Switzerland, 1998.

[3] Fiasco.OC: The Fiasco microkernel. `http://os.inf.tu-dresden.de/Fiasco/`, 2011.

[4] Alexandrescu, A. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, Feb. 2001.

[5] Feske, N. A Case Study on the Cost and Benefit of Dynamic RPC Marshalling for Low-Level Systems Components. *SIGOPS OSR Special Issue on Secure Small-Kernel Systems* (2007).

[6] Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S., and Wolter, J. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)* (Saint-Malo, France, Oct. 1997), pp. 66–77.

[7] Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: Formal verification of an operating system kernel. *Communications of the ACM 53*, 6 (Jun 2010), 107–115.

[8] Lackorzynski, A., and Warg, A. Taming Subsystems: Capabilities as Universal Resource Access Control in L4. In *IIES '09: Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems* (Nuremberg, Germany, 2009), ACM, pp. 25–30.

[9] Liedtke, J. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)* (Copper Mountain Resort, CO, Dec. 1995), pp. 237–250.

[10] Mettler, A., Wagner, D., and Close, T. Joe-E: A Security-Oriented Subset of Java.

[11] Miller, M. S. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[12] Miller, M. S., Yee, K.-P., and Shapiro, J. Capability Myths Demolished. Tech. rep., Combex, Inc.; University of California, Berkley; John Hopkins University, 2003.

[13] Shapiro, J. S. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, Apr. 1999.

[14] Spiessens, F., and Roy, P. V. The oz-e project: Design guidelines for a secure multiparadigm programming language. In *Multiparadigm Programming in Mozart/Oz*, pp. 21–40.

[15] Steinberg, U., and Kauer, B. NOVA: a microhypervisor-based secure virtualization architecture. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems* (New York, NY, USA, 2010), ACM, pp. 209–222.

[16] Sutter, H. The New C++: Smart(er) Pointers. `http://www.drdobbs.com/184403837#`, Aug. 2002.

[17] Watson, R. N. M., Anderson, J., Laurie, B., and Kennaway, K. Capsicum: practical capabilities for UNIX. In *Proceedings of the 19th USENIX Security Symposium* (Aug. 2010).