# Hardware and Software Support for Fine-Grained Memory Access Control and Encapsulation in C++ *

Eugen Leontie Gedare Bloom Rahul Simha
The George Washington University
eugen@gwu.edu gedare@gwu.edu simha@gwu.edu

## Abstract

Object-oriented programming (OOP) encapsulates object implementations with access specifiers like *public* and *private*. Compilers can verify that code adheres to specifiers, but verification can be broken in languages like C++ by unchecked pointers. Thus, C++ programmers are taught that "private is not secure." The lack of isolation between objects frustrates memory protection in OOP code. We propose hardware and software support to confine memory accesses in fine-grained memory regions that isolate within and between objects so that C++ programs can enforce encapsulation and prevent pointer-based exploits. Such support makes *private* secure. Although we target C++, our approach handles generic techniques like inheritance, polymorphism, dynamic dispatch, dynamic binding, and encapsulation.

***Categories and Subject Descriptors*** Security and privacy [*Security in hardware*]: Hardware security implementation

***Keywords*** hardware containers, encapsulation

## 1. Introduction

Objects are the natural boundary for protection in an OOP application. Memory protection can isolate between and within objects (inter- and intra-object protection). Although researchers have studied memory protection extensively for unsafe languages like C, OOP languages like C++ present new challenges including inheritance and composition, polymorphism, constructors/destructors, exception handling, friend classes, casting, and stack-instantiated objects. Even Java can violate language-based memory protection with reflection and the Java Native Interface.

* Work supported by NSF CNS-0934725 and AFOSR FA9550-09-1-0194.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
*SPLASH '13*, October 26–31, 2013, Indianapolis, Indiana, USA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-1995-9/13/10.
http://dx.doi.org/10.1145/2508075.2508091

Commodity memory protection works at the granularity of page and process, which is little help for object protection. For such *process-oriented memory protection*, isolating code modules requires dividing an application into processes and replacing function calls with inter-process communication, which imposes development and runtime costs.

In prior work, we introduced *containers*, hardware support for fine-grained memory protection [1]. In this paper, we discuss adopting containers to support inter- and intra-object protection in C++. Our approach aligns well with object interactions without artificial processes. Our contribution is a platform that secures direct memory access with OOP. Our solution improves on the related work by being the first to address private member runtime protection for C++.

## 2. Hardware Containers

A container is a set of bounded, contiguous ranges of instructions that is given permission to access memory regions. Containers do not overlap instruction ranges, and each container has a unique *container identifier* (cid). Code that belongs to the same container shares a cid. In OOP a method is a useful unitary container. Methods in the same library or class can be combined into a larger container for efficiency.

The *container manager (CM)* is a hardware reference monitor that enforces permissions using content-addressable memory to avoid affecting processor path delays [1]. The CM enforces memory permissions—read, write, execute, and delegate—on all memory regions. Delegate permission controls permission propagation between containers. The compiler infers permissions on arguments and automatic variables. Permissions for dynamically-allocated memory that the compiler is unable to resolve are granted programmatically by permission-granting ALLOW instructions. *ALLOW* enables permissions on dynamic memory, and the ALLOWM variant supports linked data structures [1]. A call/return to another container causes a *security context switch*, which pushes/pops the dynamic permissions onto a permission stack mirroring the call stack. This stack obviates permission naming, lookup, and revocation.

## 2.1 Containers for C++

OOP features can cause problems for the CM: permissions for private data must exist for object methods; exceptions circumvent the call stack and access object fields, locals, and global structures; inheritance and composition fragment object memory layout; virtual functions and polymorphism require dynamic execution control flow and dynamic type information; and `friend` classes require permission to access private fields. `const`, casts, and placement new are straightforward, but containers alone is insufficient for serialization.

To support `private` data we add ownership transfer of memory regions with ALLOWD, and persistent private access rights with ALLOWP. ALLOWD revokes access from one container and delegates access to another. ALLOWP creates a dynamic permission that a container with a specified cid can use—cid 0 gives permission for any container. Allocators execute ALLOWD to pass ownership rights over private memory to class constructors, which use ALLOWP on private data with the cid of each member method, and with cid 0 for public data. Adversarial code could use a derived class to access `protected` members, so the compiler and CM treat protected like private, and instances of the base class are not affected by (malicious) derived objects. Inheritance and composition affect performance of delegating permissions because compilers split object memory layouts so that a child (front-end) class cannot access private members of ancestor (back-end) classes. Splitting the memory range fragments the object representation and increases the overhead of permission management, because the CM needs to manage permissions for each private region of an object.

We have also identified solutions to problematic areas of C++ but not yet implemented our solutions. For *exception handling*, we propose to compartmentalize global state, remove trusted access from exception handling code, and subject exception handling to the same memory protection as other code. We identify three necessary modifications: (1) each container needs methods for stack unwinding and compensation code; (2) global tables need to be divided into per-container tables; (3) global lists must be placed in containers to confine operations done on these lists. *Virtual functions and polymorphism* require inserting extra ALLOW instructions to handle dynamic binding, and dynamic permissions may be needed for fields in a derived class that are not present in the base class. Such permissions may be derived by extending virtual tables with object permission layout information similar to runtime type identification (RTTI). The same kind of RTTI can be applied to disambiguate *multiple inheritance*. Containers can support *friend classes* if constructors create private permissions for all friends, which is not a problem because classes already declare friends. The compiler can extend the *const attribute* to CM enforcement by using read-only permission on `const` data, but `const` casts will not change the read-only permission. Other *cast operators* can work directly. *Placement new* allows a programmer to place an object at a predetermined memory location, which bypasses heap allocation, but otherwise uses the same permission delegation as `new`—execute ALLOWD to relinquish ownership of object memory. *Serialization* enables object storage and transmission and requires additional trusted code in the C++ runtime and kernel I/O.

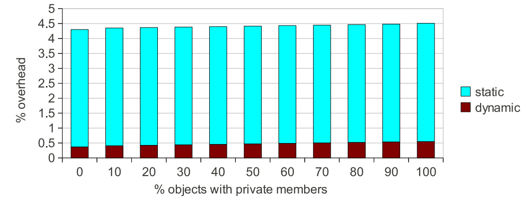


**Figure 1.** Overhead for heap-allocated objects.

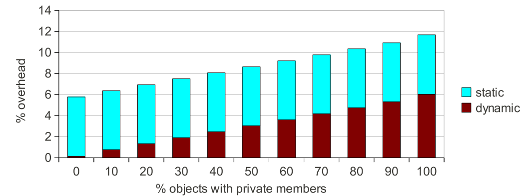


**Figure 2.** Overhead for stack-allocated objects.

# 3. Experiments and Results

We implemented our approach using GEMS/Opal [2] and evaluate it using C++ microbenchmarks designed to stress performance. Our microbenchmark mimics C program workloads and adds C++ object creation, access of private/public members, and object destruction. We have not been able to measure real applications due to time constraints. Results show the percent overhead of execution time using our approach compared with the baseline: smaller is better.

Figures 1 and 2 show the costs for object creation and destruction on the heap and stack respectively for 1000 objects and a varying percent of objects using private data. Memory allocation, rather than permissions management, dominates the performance for heap objects. For stack objects, overhead reaches 11.68% when all objects have private data.

# 4. Conclusion

We demonstrated that fine-grained memory protection can support OOP languages like C++. Future work can implement and evaluate more OOP features and real applications.

# References


[1] E. Leontie, G. Bloom, B. Narahari, and R. Simha. No principal too small: Memory access control for fine-grained protection domains. *15th Euromicro Conference on Digital System Design(DSD)*, September 2012.

[2] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33 (4):92–99, 2005. ISSN 0163-5964.