2013

Ironclad C++: A Library-Augmented Type-Safe Subset of C++

Christian DeLozier
University of Pennsylvania

Richard A. Eisenberg
Bryn Mawr College, rae@cs.brynmawr.edu

Santosh Nagarakatte
Rutgers University - New Brunswick/Piscataway

Peter-Michael Osera
Grinnell College

Milo M. K. Martin
Google

See next page for additional authors

Ironclad C++

A Library-Augmented Type-Safe Subset of C++

Christian DeLozier Richard Eisenberg Santosh Nagarakatte[†] Peter-Michael Osera
Milo M. K. Martin Steve Zdancewic

Computer and Information Science Department, University of Pennsylvania
[†]Computer Science Department, Rutgers University
{delozier, eir, posera, milom, stevez}@cis.upenn.edu santosh.nagarakatte@cs.rutgers.edu

Abstract

The C++ programming language remains widely used, despite inheriting many unsafe features from C—features that often lead to failures of type or memory safety that manifest as buffer overflows, use-after-free vulnerabilities, or abstraction violations. Malicious attackers can exploit such violations to compromise application and system security.

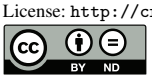
This paper introduces Ironclad C++, an approach to bringing the benefits of type and memory safety to C++. Ironclad C++ is, in essence, a library-augmented, type-safe subset of C++. All Ironclad C++ programs are valid C++ programs that can be compiled using standard, off-the-shelf C++ compilers. However, not all valid C++ programs are valid Ironclad C++ programs: a syntactic source-code validator statically prevents the use of unsafe C++ features. To enforce safety properties that are difficult to check statically, Ironclad C++ applies dynamic checks via templated "smart pointer" classes.

Using a semi-automatic refactoring tool, we have ported nearly 50K lines of code to Ironclad C++. These benchmarks incur a performance overhead of 12% on average, compared to the original unsafe C++ code.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Dynamic storage management

General Terms Languages, Performance, Reliability, Security

Keywords C++, type safety, memory safety, local pointers



OOPSLA '13, October 29, 2013, Indianapolis, IN, USA.

ACM 978-1-4503-2374-1/13/10.
http://dx.doi.org/10.1145/2509136.2509550

1. Introduction

C and C++ are widely used programming languages for implementing web browsers, native mobile applications, compilers, databases, and other infrastructure software [33]. C and C++ provide efficiency and low-level control, but these advantages come at the well-known cost of lack of memory and type safety. This unsafety allows programming errors such as buffer overflows (accessing location beyond the object or array bounds), use-after-free errors (accessing memory locations that have been freed), and erroneous type casts to cause arbitrary memory corruption and break programming abstractions. More dangerously, malicious attackers exploit such bugs to compromise system security [29].

Recognizing this problem, many approaches have been proposed to prevent memory safety violations or enforce full memory safety in C and C-like languages [2, 4, 9, 10, 12, 14, 17, 23–25, 31, 36]. Collectively, these prior works identified several key principles for bringing safety efficiently to C. However, one challenge in making C memory safe is that C provides limited language support for creating type-safe programming abstractions. In contrast, although C++ includes many unsafe features, C++ also provides advanced constructs that enable type-safe programming, such as templates and dynamically checked type-casts.

Ironclad C++ is our proposal to bring comprehensive memory and type safety to C++. Ironclad C++ is, in essence, a library-augmented [32] type-safe subset of C++. As such, an Ironclad C++ program is a valid C++ program (but not all C++ programs are valid Ironclad C++ programs). Dynamic checking is implemented in a "smart pointer" library, so no language extensions or compiler changes are required. Ironclad C++ uses a syntactic static validation tool to ensure the code conforms to the Ironclad C++ subset of C++, and the code is then compiled using an unmodified off-the-shelf C++ compiler (see Figure 1).

This paper describes Ironclad C++ and reports on our experience programming in this type-safe subset of C++, including:

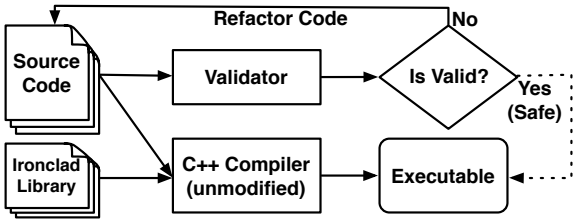


Figure 1. Workflow for coding with Ironclad C++

- a C++ smart pointer library that efficiently and comprehensively enforces strong typing and bounds safety,
- a hybrid static–dynamic checking technique that prevents use-after-free errors for stack objects, without requiring costly heap allocation,
- an opt-in, source-level heap-precise garbage collector designed to reduce memory leaks due to conservative GC,
- tools for semi-automated refactoring of existing C and C++ code into the Ironclad subset of C++ and a validator that enforces compliance with that subset, and
- experimental evaluation of the performance overheads of this approach to enforce memory safety.

To evaluate practicality and performance, we refactored several C/C++ programs—over 50K lines in total—into Ironclad C++. We performed this translation with a semi-automatic refactoring tool we created to assist in this conversion, and we report our experiences converting these programs. We also converted multiple test suites designed for evaluating memory safety bug detectors, which confirmed that Ironclad C++ does successfully detect memory access violations. Using performance-oriented benchmarks, we measured an average performance overhead of 12% for enforcing complete type and memory safety.

2. Overview and Approach

Type-safe languages such as Java and C# enforce type safety through a combination of static and dynamic enforcement mechanisms such as static typing rules, dynamically checked type casts, null dereference checking, runtime bounds checking, and safe dynamic memory management. In such languages, violations of type safety are prevented by either rejecting the program at compile time (type errors) or raising an exception during execution. The aim of Ironclad C++ is to use this same set of familiar static/dynamic enforcement mechanisms to bring type-safety to C++ while retaining much of the syntax, performance, and efficiency of C++. To provide context on how these enforcement mechanisms apply to C/C++, this section overviews some key principles of efficient type safety for C, surveys their implementations in prior work, and describes how these principles can be brought to C++ by leveraging existing language features.

Differentiating array pointers from non-array pointers. Several prior memory safety proposals have recognized the performance benefit of distinguishing between a pointer to an array (which requires bounds information) versus a pointer to non-array (a.k.a. singleton) object (which does not). Doing so automatically has typically relied on whole-program analysis at compile time or introduced language extensions. For example, CCured [25] uses a whole-program type inference at compile time to distinguish between singleton and array pointers, Cyclone [17] introduces different type decorators, and some pool allocation approaches [9] create type-homogeneous pools of singleton objects. Systems without such differentiation implicitly treat all pointers as array pointers, adding unnecessary space and time overheads for bounds checking.

Ironclad C++ captures this differentiation between singleton and array pointers without language extension or whole-program analysis during each compilation by using the well-known C++ technique of *smart pointers* [1, 2, 8]. Smart pointers leverage C++'s template and operator overloading constructs, and they have previously been used to dynamically insert safety checks [2] or perform reference counting [8] on pointer operations. Ironclad C++ requires that all bare C++ pointer types be replaced with one from a suite of smart pointers, some of which include bounds information and thus support pointer arithmetic and indexing (for array pointers) and some that avoid the bounds checking overhead (for singleton pointers). The distinction between singleton and array smart pointer types allows their overloaded operators to perform the minimum dynamic checking necessary to detect bounds violations based on the type of the pointer.

Enforcing strong typing. C's use of `void*` and unchecked type casts results in either pessimistic typing assumptions, which can significantly increase the overhead of dynamic checking [23, 27], and/or the failure to detect all memory safety violations. Disallowing unsafe casts in C reduces checking overhead, but doing so has typically required augmenting the C language in some way. For example, Cyclone found it necessary to support generics, CCured adds RTTI (run-time type information) pointers, and both support structural subtyping. However, C++ already provides alternatives to C's unsafe constructs (for example, templates and class hierarchies). Yet, to facilitate adoption, C++ inherited many of C's unsafe constructs. Ironclad C++ takes a different approach and explicitly enforces strong typing by disallowing legacy type-unsafe constructs and requiring that all pointer type-cast operations are either known to be safe statically or checked dynamically (by building upon C++'s existing `dynamic_cast` construct).

Heap-safety through conservative garbage collection. Ironclad C++'s smart pointers provide strong typing and bounds safety, but they do not prevent use-after-free errors. To avoid the overhead of reference counting [12] or use-after-free checking [2, 24, 36], Ironclad C++ facilitates

the use of conservative garbage collection [7] by targeting a key challenge of using conservative GC to enforce safety: conservative collection can lead to non-deterministic memory leaks (due to non-pointer data that "looks" like a pointer) [5, 30]. To reduce such memory leaks due to conservative garbage collection, Ironclad C++ supports *heap-precise* garbage collection. Inspired by prior work on mostly-copying and more-precise conservative garbage collection [3, 11, 15, 30], Ironclad's collector treats the roots conservatively but supports the precise identification of pointers in heap objects by employing `mark()` class methods to precisely identify pointer fields and pointer containing members.

Facilitating stack-allocation safety. Garbage collection does not prevent dangling pointers to stack objects. In recognition of this problem, CCured prevents use-after-free errors to stack objects by selectively converting escaping stack-allocated objects into heap-allocated objects (a.k.a. *heapification*), which unfortunately introduces significant performance overheads in some programs [25]. To prevent use-after-free errors for stack allocated memory without the performance penalties of heapification, this paper introduces hybrid static-dynamic checking of stack pointer lifetimes. The hybrid checking in Ironclad C++ avoids the performance overheads of heapification with simple static checking and limited dynamic checking.

Validating conformance statically. Statically validating that code conforms to the rules of Ironclad C++ is paramount for ensuring safety. Without some form of static validation, smart pointer schemes cannot alone provide a guarantee of safety because unsafe constructs can still be used outside of the use of smart pointers, and smart pointers can be used incorrectly. Our current prototype divides this responsibility between two checkers. First, we created a static code validator that checks basic syntactic properties of the program to ensure that it conforms to the Ironclad C++ subset (*e.g.*, no raw pointers). Second, after carefully precluding unsafe constructs with the validator, we then leverage the existing C++ type checker to complete the remaining checking of type safety. This use of strong static validation distinguishes Ironclad C++'s approach from other non-validated smart pointer approaches [2, 8] and `mark()` methods for precise garbage collection [3, 30].

3. Bounds Checking & Strong Static Typing

The Ironclad dialect of C++ is formed by first providing safe idioms via the Ironclad C++ library (*e.g.*, the smart pointers described in Table 1) and then disallowing the use of unsafe language features (*e.g.*, disallowing the use of raw pointers). In this way, Ironclad C++ brings memory and type safety to C++ using a combination of static code validation, the standard C++ type checker, and dynamic safety checks. A code validator, described in Section 7, statically enforces that disallowed constructs are not used. This section first describes safety without considering arrays or memory deallocation (Section 3.1), it then describes support for references (Section 3.2), arrays, and pointer arithmetic (Section 3.3). Further sections discuss memory deallocation safety for the heap (Section 4) and stack (Section 5).

Type	Capabilities	Safety Checks
`ptr`	Dereference	Null Check
`aptr`	Dereference	Bounds Check
	Index	Bounds Check
	Arithmetic	No Check†
`lptr`	Dereference	Null Check
	Receive address-of object	Lifetime Check
	Receive *this* pointer	Lifetime Check
`laptr`	Dereference	Bounds Check
	Receive address-of object	Lifetime Check
	Receive *this* pointer	Lifetime Check
	Hold static-sized array	Lifetime Check
	Index	Bounds Check
	Arithmetic	No Check†

†: Arbitrary pointer arithmetic is permitted because a bounds check occurs before an `aptr` is dereferenced.

Table 1. This table describes the capabilities and safety checks performed by each pointer type. As more letters are added to the type, the capabilities increase, but the overhead also increases due to additional required checks.

3.1 Strong Static Typing with `ptr<T>`

Ironclad C++ requires replacing all raw C++ pointer types with templated smart pointer types. For referring to singleton (non-array) objects, Ironclad C++ provides the `ptr<T>` class. Because `new` returns a raw pointer, Ironclad C++ provides a replacement for performing heap allocation, `new_obj<T>(...)`, which uses `new` internally but returns a `ptr<T>` (rather than returning a `T*`). Similarly, calls to `delete` are replaced by `destruct()`, which calls the pointed-to object's destructor (but it does not necessarily deallocate the underlying memory, as described in Section 4). Accordingly, the following C++ code:

```
Rectangle* p = new Rectangle(2, 5);
...
delete p;
```

Would be rewritten in Ironclad C++ as:

```
ptr<Rectangle> p = new_obj<Rectangle>(2, 5);
...
p.destruct();
```

C++11's variadic templates allow `new_obj` to accept arbitrary arguments to pass along to the underlying object constructor.

Rule (Pointers)**.** *All pointer types are transformed to `ptr<T>` (or one of its variants, described below) provided by the Ironclad C++ library. Raw pointers are disallowed.*

Example: Strong Static Typing

```
float radius(Shape* shape) {
  Circle* circle = static_cast<Circle>(shape);
  return circle->radius;
}
```

```
float radius(ptr<Shape> shape) {
  ptr<Circle> circle = cast<Circle>(shape);
  return circle->radius;
}
```

Example: Bounds Checking

```
float* computeArea(Shape* shapes, int N) {
  float* areas = new float[N];
  for(int i = 0; i < N; ++i) {
    float r = radius(shapes);
    areas[i] = PI * (r * r);
    shapes++;
  }
  return areas;
}
```

```
aptr<float> computeArea(aptr<Shape> shapes, int N) {
  aptr<float> areas = new_array<float>(N);
  for(int i = 0; i < N; ++i) {
    float r = radius(shapes);
    areas[i] = PI * (r * r);
    shapes++;
  }
  return areas;
}
```

Figure 2. Comparison of C++ syntax (left) and Ironclad C++ syntax (right).

Supporting type casts safely. By disallowing raw pointers, Ironclad C++ also implicitly disallows both `void*` pointers and unsafe pointer-to-pointer casts. To support safe pointer-to-pointer casts, Ironclad C++ provides a `cast<T>` function template to safely cast a `ptr<S>` to a `ptr<T>`. Figure 2 shows an example of casting from a `ptr<Shape>` to a `ptr<Circle>`. The `cast<T>(...)` function is a wrapper over C++'s existing `dynamic_cast` operation, which is used to cast between members of a class hierarchy. Casts between incompatible types will be caught either: (1) during compilation when the template is instantiated (*e.g.*, when attempting a cast that can be proven invalid during type-checking) or (2) when the underlying `dynamic_cast` fails at runtime due to an incompatible type, setting the resulting pointer to `NULL`. Casting from `void*` or integers to a pointer is not supported by C++'s `dynamic_cast`, so this use of `dynamic_cast` statically enforces that a `ptr` cannot be created from an integer or `void*` pointer. Uses of `void*` pointers can generally be eliminated by refactoring the code to use inheritance or templates (*e.g.*, to implement generic containers, which is one use-case of `void*`). Note that Ironclad C++ does not restrict cast operations (type conversions) among non-pointer types, such as `int`s and `double`s, because such type conversions are well-defined and do not violate memory safety. For example, if a variable is cast from a negative `int` to an `unsigned int` and then used as an index into an array, the possibly out-of-bounds index will be caught by Ironclad C++'s dynamic checks. Thus, type-conversions do not violate memory safety in Ironclad C++.

Rule (Pointer Casts)**.** *Pointer casts must use `cast<T>(...)`, provided by the Ironclad C++ library.*

C-style unions are not allowed in Ironclad C++ because, unlike type-casts on non-pointer types, the implicit cast between types that occurs through the use of a union can lead to undefined behavior [16]. Unions are less prevalent in C++ compared to C because only POD (plain old data) types can be used in unions.

Rule (Unions)**.** *Unions are disallowed in Ironclad C++.*

Dynamic `NULL` checking. As `ptr<T>` pointers may point only to singleton objects (arrays and pointer arithmetic are handled in the next subsection), the `ptr<T>` class explicitly does not overload the operators for performing array indexing and pointer arithmetic, so the standard C++ type checker disallows such operations during compilation. However, dereferencing a `NULL` pointer is an illegal memory operation. To prevent this, the overloaded dereference operations (`*` and `->`) in `ptr` check for `NULL` prior to performing the dereference. Although many `NULL` dereferences would result in a segmentation fault, the explicit check is still necessary. Consider a program in which `p` is `NULL` and is a pointer to type `struct Foo{int a[1000000]; int x;}`. Without an explicit `NULL` check, the expression `p->x` would attempt to address at address `1000000*sizeof(int)`, which could result in arbitrary memory corruption. Checking for `NULL` before dereference also catches the dereference of a `ptr` that resulted from an invalid dynamically checked pointer cast.

3.2 C++ References

References in C++ (`T&`) are similar to pointers but differ in a few ways that allow them to be treated differently in Ironclad C++. References must be initialized when declared, as shown below.

```
Object& r = *p;   // p has type ptr<Object>
```

Once a reference has been initialized, the location that the reference points to cannot change.

In Ironclad C++, the `NULL` check performed during the dereference of a pointer ("`*p`" in the above code) prevents the creation of `NULL` references. By preventing the creation of `NULL` references, there is no need to perform a `NULL` check on the use of a reference. Thus, unlike raw pointers, which must be wrapped using smart pointers, C++ style references

are allowed in Ironclad C++. References will be further discussed in Section 5.3.

3.3 Bounds Checking with `aptr<T>`

Ironclad C++ supports static-sized arrays, dynamic-sized arrays, and pointer arithmetic by providing the `array<T,N>` and `aptr<T>` ("array pointer") classes. For static-sized arrays, the Ironclad C++ library provides a templated array class `array<T,N>`. This class overrides the index operator and checks that the requested index (an `unsigned int`) is less than `N` before returning the requested element. To create an `array<T,N>`, the size `N` of the allocated array must be known at compile time.

Rule (Static-sized Arrays)**.** *All static-sized arrays must be replaced by `array<T,N>`.*

To support dynamic-sized arrays, Ironclad C++ provides an `aptr<T>` class. The `aptr<T>` class replaces raw pointers for referring to either dynamically or statically sized arrays. To perform the necessary bounds checking, each `aptr` is a three-element fat pointer with a pointer to the base of the array, the current index, and the maximum index. A bounds check is performed on each dereference or array index operation. This bounds check will fail if the pointer is `NULL`, so a separate `NULL` check is not needed. Arbitrary pointer arithmetic is allowed, and the bounds check during dereference and array indexing are sufficient to detect invalid pointer arithmetic. To heap allocate new dynamically sized arrays, the Ironclad C++ library provides `new_array<T>(size)` function, which returns an `aptr<T>` created by calling `new`. Accordingly, the following C++ code:

```
Foo* p = new Foo[number];
...
delete[] p;
```

Would be rewritten in Ironclad C++ as:

```
aptr<Foo> p = new_array<Foo>(number);
...
p.destruct();
```

Figure 2 shows an example of creating and using an array in Ironclad C++. As shown, only the type and allocation of the array must be rewritten to conform to Ironclad C++; all other array pointer operations are performed using overloaded operators.

Rule (Array Pointers)**.** *Pointers to dynamic and static arrays must be replaced by `aptr<T>`.*

Ironclad C++ provides both `ptr<T>` and `aptr<T>` because they provide different tradeoffs: `ptr` does not provide indexing or pointer arithmetic operators, but it avoids the performance and storage overheads incurred by the bounds checking for `aptr`. The Ironclad C++ library provides an implicit conversion from `aptr<T>` to `ptr<T>`, allowing a `ptr` to point to a single element of an array. During such a conversion, if the `aptr` is invalid (not in bounds) the `ptr` is set to `NULL`.

3.4 Pointer Initialization

If pointers were allowed to be uninitialized, a pointer could contain garbage and point to arbitrary memory. Therefore, Ironclad C++ ensures that pointers are properly initialized by setting the underlying raw pointer to `NULL` in the default constructor for each smart pointer class.

In one particularly insidious corner case, the order of initialization of members of a class may allow a smart pointer to be dereferenced before its constructor has been called:

```
class Foo {
  int x;
  ptr<int> p;
  Foo() : x(*p)  // initializer list
  {
    ...  // body of constructor
  }
};
```

To ensure proper initialization, the initializer expression for any data member must not use a smart pointer data member field that was declared subsequent to the data member being initialized (in C++ the order of initialization of fields is determined by the order in which they were declared in the class declaration). To enforce the above rule, Ironclad C++ also disallows the use of the `this` pointer and method calls within any initializer list expression for a data member field that was declared prior to any smart pointer data member field. The static validator enforces these requirements.

Rule (Init.)**.** *A `ptr<T>` must be initialized before use.*

3.5 The C Standard Library

The C Standard Library contains utility functions, standard types, I/O functions, functions on C-strings (`char *`), and other functionality. To ensure safety, Ironclad C++ disallows the use of some of the available headers (*e.g.*, `<csetjmp>`) and replaces others with safe versions. A few of the C standard library headers contain functions that consume C-string parameters without checking to see if the C-string is properly null terminated or large enough to perform the operation. Even the functions in `<cstring>` that take a size parameter, such as `strncpy`, can violate memory safety if the size parameter is incorrect.

Ironclad C++ provides safe functions to replace each of these unsafe functions. These functions take `aptr<char>` parameters instead of `char*` and check that the inputs are null-terminated and within the specified bounds. Two specific functions — `memset` and `memcpy` — are unsafe in C++. Calling `memset` can accidentally overwrite virtual pointers; `memcpy` ignores any effects that copy constructors might have. Ironclad C++ replaces `memset` and `memcpy` with the functions `zero<T>` and `copy<T>`. The `zero` function iterates through the input array and sets each element to 0. The `copy` function assigns each element of the source array to the corresponding element in the destination array. To improve performance, the `zero` and `copy` functions have hand-

optimized template specializations for standard data types, such as `char`.

The `<cstdio>` header contains C-style variable argument functions that rely on the programmer to provide a correct format string. In C++11, the unsafe use of `va_list` can be replaced by the type-safe use of variadic templates. Using variadic templates, Ironclad C++ checks that the number and type of arguments provided to functions such as `printf` and `scanf` matches the arguments expected by the format string.

Rule (C Standard Library Functions)**.** *Uses of unsafe C Standard Library functions must be replaced with their corresponding safe variant (*e.g.*, `strlen(const char *)` with `safe_strlen(aptr<const char>)`).*

4. Heap-Precise Garbage Collection

Along with strong typing and bounds checking, deallocation safety (*i.e.*, no dangling pointers) is the final requirement for comprehensive type and memory safety. In many languages, including C++, pointers can refer to both heap-allocated and stack-allocated objects. In standard C++, heap-allocated objects are created using `new` and explicity deallocated using `delete`. Stack-allocated objects are automatically allocated and deallocated on function entry and exit, respectively. Ironclad C++ uses separate mechanisms for enforcing deallocation safety for heap-allocated and stack-allocated objects. For heap-allocated objects, Ironclad C++ relies on a conservative garbage collector to delay the deallocation of heap objects until a time at which it is known to be safe (*i.e.*, no references to the object remain).

4.1 Conservative Collection

For many applications, a conservative, mark-sweep garbage collector prevents dangling pointers for heap allocations with reasonable performance overheads. Unlike reference counting [12] (which requires work on every pointer assignment), dynamic allocation checking [2, 24] (which requires work on every pointer dereference), and precise garbage collection (which requires work to provide safe points), conservative garbage collection performs work only when the allocator requires more free memory than is currently available. At that time, the collector conservatively scans the program roots (*e.g.*, stack, registers, and globals) for pointers to heap allocations and pushes each identified address onto a mark stack. Collection then proceeds by (1) popping an address to an allocation off of the mark stack, (2) discarding the address if it has already been marked, (3) performing the "marking" operation by conservatively scanning the allocation region for any values that are potential pointers, (4) pushing all such values onto the mark stack, and (5) recording that the allocation has been marked. This process repeats until the mark stack is empty. Finally, the collector frees all allocations that were not recorded as having been marked [19].

```
class Circle :
  // Inherits from Shape & IroncladPreciseGC
  public Shape, public IroncladPreciseGC {
private:
  Color color;
  ptr<Point> center;
  double radius;
public:
  void mark() {
    Shape::mark();  // Marks the base class
    color.mark();   // Marks the color object
    center.mark();  // Enqueues on mark stack
    // radius is primitive, no need to mark
  }
};
```

Figure 3. Example of `mark()` methods for heap-precise collection. Calling `mark()` on the `ptr<Point>` object pushes its address onto the mark stack to ensure that the collector marks the referred to `Point` object.

Although conservative garbage collection exhibited low runtime and memory overheads on many programs (including most of the benchmarks we use for evaluations in Section 9), some programs suffer from large memory overheads due to a well-known weakness of conservative collection in which memory leaks can occur due to non-pointer data (such as floating-point numbers) that "looks" like a pointer [15, 30, 34]. Several approaches have been proposed to mitigate this problem. The Boehm-Demers-Weiser collector performs blacklisting to avoid allocating on pages that have previously been pointed to by non-pointer data [7]. It also provides an interface for providing precise pointer identification bitmap descriptors for allocated data [5]. A proposal was put forth (although not accepted) for C++11 that would have added keywords to C++ for precise identification of specified `gc_strict` classes [6]. Prior approaches have proposed methods for precisely identifying pointers in the heap either by tracking pointers on creation and destruction [11] or calling tool-generated or user-defined methods for precisely identifying an object's pointer and pointer containing members [3, 30].

4.2 Heap-Precise Collection

Building on these prior works, Ironclad C++ adopts opt-in *heap-precise* garbage collection. To enable heap-precise collection, a class (1) provides a `mark()` method that precisely identifies all of the pointers in the allocation and (2) inherits non-virtually from `IroncladPreciseGC`, which is used as a type tag to inform the collector (*i.e.*, in `new_obj<T>(...)`) to call the `mark()` method when scanning the allocation.

Like the conservative collector, heap-precise collection begins by conservatively scanning the program roots for pointers to the heap and pushing each identified address onto the mark stack. The collector then proceeds as usual by popping an address to an allocation off of the mark stack,

discarding the address if it has already been marked. To perform the "marking" operation, the collector invokes the `mark()` method on the object, which precisely identifies all pointers in the allocation and pushes them onto the mark stack. If the type of the allocation does not inherit from the `IroncladPreciseGC` class, the collector falls back to using the conservative marking strategy. Like the conservative collector, once the allocation has been processed, the allocation is recorded as having been marked. This process repeats until the mark stack is empty, and the collector frees all allocations that have not been marked.

4.3 Requirements for Precise Marking

For an object to be precisely marked, all of the objects that it directly contains (*i.e.*, not pointers to objects) and all of its base classes must define `mark()` methods because the heap-precise collector must be aware of all pointers in the physical layout of the marked object. A class's `mark()` is responsible for adding all pointers contained in the object to the mark stack, including all pointer that are data member fields, in object fields, or in an inherited-from base class. The `mark()` method accomplishes this task by calling `mark()` on all smart pointers data members, on all object data member fields, and on all inherited-from base classes. The `mark()` method of each smart pointer class pushes the address onto the mark stack. The `mark()` methods of the object's data members will in turn subsequently call `mark()` methods on their fields, resulting in all pointers contained by the allocation being added to the mark stack. Primitive non-pointer types need not be marked. Figure 3 provides an example `mark()` method for a simple class hierarchy. Calling the `mark()` method on class that does not actually contain pointers is unlikely to hurt performance because such `mark()` methods will be empty and defined in the class's header file, allowing them to be optimized away by the compiler's inlining pass. The heap-precise collector uses template specialized `mark()` methods for arrays of primitive data types. Primitive types do not contain pointers, so these methods do nothing and thus avoid the overhead of scanning an array when it clearly does not contain pointers.

In Ironclad C++, the `mark()` methods are supplied by the programmer and part of the program's permanent source code. To ensure safety, the `mark()` methods are checked for conformance with the above rules by the Ironclad C++ validator (Section 7), unlike prior systems that utilize programmer-supplied marking [3, 30]. To assist the programmer, the Ironclad C++ refactoring tool can automatically generate `mark()` methods in many cases.

4.4 Object Destruction

In C++, calling `delete` on a heap-allocated object will destruct the object and reclaim the memory in which the object had been allocated. In Ironclad C++, the garbage collector reclaims memory only once an object becomes unreachable. Therefore, Ironclad C++ divides the functionality otherwise provided by `delete` between the garage collector, which reclaims the object's memory, and the `destruct()` method (provided by `ptr` and `aptr`), which explicitly calls the object's destructor. This approach avoids the problem of calling destructor asynchronously, but it has the disadvantage that Ironclad C++ does not prevent a program from accessing an object after its destructor has been called. Such an access—although certainly a bug in the program—does not violate type safety because the garbage collector ensures the underlying memory is still valid (*i.e.*, has not be reclaimed) and thus remains properly typed.

5. Stack Deallocation Safety via `lptr`

Although garbage collection prevents all dangling pointers to objects on the heap, it does not protect against dangling pointers to stack-allocated objects. One way to prevent such errors is to forgo some of the efficiency benefits of stack allocation by limiting the use of stack allocation to non-escaping objects only (a.k.a. *heapification*). To avoid the performance penalties of heapification, Ironclad C++ provides additional templated smart pointers that, cooperatively with the static code validator and C++ type checker, uses *dynamic lifetime checking* to prevent use-after-free errors for stack allocations while avoiding heapification in almost all cases.

5.1 Heapification

A common approach for preventing use-after-free errors for stack allocations in a garbage collected system is simply to restrict stack allocations by employing heapification [25], which is the process of heap-allocating an object that was originally allocated on the stack. Heapification enforces deallocation safety by conservatively disallowing any object whose address might escape the function from being allocated on the stack. For example, without inter-procedural analysis, heapification requires heap allocation of any object whose address is passed into a function. This is a particular challenge for C++ code, because object methods and constructors are implicitly passed the address of the object (the `this` pointer), thus disallowing stack allocation of almost all objects unless inter-procedural analysis could prove otherwise. Unfortunately, heapification results in significant performance degradations in some cases (see Section 9.5).

5.2 Dynamic Lifetime Checking

Ironclad C++ reduces the need for heapification by focusing on preventing the two causes of dangling pointers to stack allocations: stack addresses that escape to the heap or globals and stack addresses that escape to longer-living stack frames. To prevent dangling pointers to the stack, Ironclad C++ identifies, tracks, and prevents the unsafe escape of stack addresses using two additional templated pointer classes, `lptr<T>` and `laptr<T>`, called *local pointers*. Prior work on preventing use-after-free errors has introduced some notion of a local pointer [10, 21], but these efforts have been

focused on purely static enforcement through sophisticated program analyses.

To prevent stack addresses from escaping to the heap or globals (example shown in Figure 4(a)), Ironclad C++ combines static restrictions on where stack addresses may exist in memory with a dynamic check on assignment between local pointers and non-local pointers (`ptr` and `aptr`). Ironclad C++ does not place any restrictions on where `ptr` and `aptr` can be allocated; `ptr` and `aptr` can both exist on the heap or as globals. Therefore, it is unsafe for a `ptr` or `aptr` to hold the address of a stack allocation.

In C++, the address of a stack allocation can be initially accessed by using the address-of operator (`&`), using the `this` pointer of a stack allocated object, or by implicitly converting from a stack allocated array to a pointer. Although operations such as address-of may also produce a non-stack address (*i.e.* if applied to a global variable), Ironclad C++ requires that the result of any operation that may produce a stack address be held by an `lptr` or `laptr`. More concretely, Ironclad C++ requires that stack addresses can only be held by `lptr` and `laptr`.

Rule (Stack Pointers)**.** *Any pointer to stack object must be held by an `lptr` or `laptr`.*

Given that local pointers can hold stack addresses, and stack addresses should not be allowed to escape to the heap or globals, it follows that local pointers cannot be stored in the heap or globals. That is, local pointers are limited to use as local variables and function parameters.

Rule (Local Pointer Location)**.** *A local pointer may only exist on the stack.*

Although it is unsafe for a non-local pointer to point to a stack allocation, a local pointer can safely point to the heap or globals. Further, the address held by a local pointer can be safely assigned into a `ptr` or `aptr` if the address refers to the heap or globals. When assigning to a `ptr` or `aptr` from an `lptr` or `laptr`, the assignment operator performs a dynamic check to ensure the address being written into the `ptr` or `aptr` is actually a heap or global address; if the address is a stack address, the check fails and an exception is raised. No such check is required when assigning a `ptr` into a `ptr` (or `aptr` into an `aptr`).

Dangling pointers to stack-allocated objects can occur even without placing a pointer to a stack object in the heap or globals. As illustrated in Figure 4(b), a stack address escaping to longer living stack frames can also cause a dangling pointer error. To prevent dangling pointers from one stack frame to another, local pointers record a lower bound on stack addresses to which they may point.[1] Local pointers are allowed to point only to stack allocations at the same level or above in the call stack. Local pointers ensure that each assignment into or out of the local pointer will not create a dangling pointer.

Overall, Ironclad C++ enforces the following invariant to ensure deallocation safety with regard to stack allocations.

Invariant (Pointer lifetime)**.** *The lifetime of a pointer may not exceed the lifetime of the value that it points to.*

To more concretely explain the dynamic checks applied by local pointers, we give a case-by-case analysis of the non-trivial assignments between `lptr` and `ptr`.

Case: Assign from `ptr<T>` into `lptr<T>`

In this case, the address being assigned into the `lptr` points to the heap or globals. Therefore, the address can be safely assigned into the `lptr`.

Case: Assign from `lptr<T>` into `ptr<T>`

To assign from an `lptr` into a `ptr`, the address currently held by the `lptr` must point to a heap or global value. A dynamic check is performed to ensure that the address held by the `lptr` points to a heap or global value. If the `lptr` points to a stack value, the check fails.

Case: Assign from `lptr<T>` into `lptr<T>`

In this case, the address held by the source `lptr` is assigned into the destination `lptr`. If the source `lptr` currently points to a heap or global value, execution proceeds as in the first case. If not, the destination `lptr` must check that the address held by the source `lptr` is not below the minimum address allowed to be held by the destination `lptr`, which is defined by the destination `lptr`'s lower bound.

To ensure the correct use of local pointers, Ironclad C++ places a few restrictions on where local pointers may be used. First, a function may not return a local pointer. Second, as noted earlier, a local pointer may not be allocated on the heap. From the second restriction, it follows that a local pointer may not be declared in a struct or class because Ironclad C++ does not restrict in which memory space an object may be allocated.

Rule (Local Pointer Return)**.** *A local pointer may not be returned from a function.*

With the dynamic lifetime checks described above and these few restrictions placed on the static use of local pointers, Ironclad C++ provides deallocation safety for stack objects without the need for heapification in most situations. For example, stack-allocated arrays can be passed to nested functions without requiring heapification. For the code we examined, the single example requiring heapification occurred in our conversion of the benchmark `leveldb`. The relevant code is shown in Figure 5. Here, the address of a stack value is stored in the field of a heap object, which is disallowed by the rules for local pointers. Even though the code does not actually create a dangling reference, Iron-

[1] The use of "lower" here assumes that a stack grows down through its memory region.

(A) Stack Address Escapes to Global Scope

```
int* global = NULL;

void f()
{
  g();
  cout << *global;  // dangling pointer dereference
}

void g()
{
  int y = 1;
  int* q = &y;
  global = q;  // Puts stack pointer into a global
}
```

```
ptr<int> global = NULL;

void f()
{
  g();
  cout << *global;  // Prevented by check in g()
}

void g()
{
  int y = 1;
  lptr<int> q = &y;  // check passes
  global = q;        // "ptr = lptr" check fails
}
```

(B) Stack Address Escapes to Longer Living Stack Frame

```
void f()
{
  int* p = NULL;
  g(&p);
  cout << *p;    // dangling pointer dereference
}

void g(int** ptr_to_p)
{
  int y = 1;
  int* q = &y;
  *ptr_to_p = q;   // Sets p to point to y
}
```

```
void f()
{
  lptr<int> p = NULL;
  g(&p);
  cout << *p;       // Prevented by check in g()
}

void g(lptr<lptr<int>> ptr_to_p)
{
  int y = 1;
  lptr<int> q = &y;  // check passes
  *ptr_to_p = q;     // "lptr = lptr" check fails
}
```

Figure 4. Examples of unsafe operations that are prevented by local pointers.

```
void Acquire(Logger* logger) {
  obj->logger = logger;
}
void Release() {
  obj->logger = NULL;
}
void f() {
  Logger logger;
  Acquire(&logger);
  ...
  Release();
}
```

Figure 5. Case in `leveldb` under which dynamic lifetime checking could not avoid heapification

clad C++ could not provide this guarantee. Therefore, the programmer must heap-allocate the object to ensure safety.

5.3 References

C++ references behave similarly to local pointers. However, references cannot be reassigned after initialization and therefore do not need to track a lower bound like local pointers. If a reference points to valid memory when it is initialized, it will point to valid memory for its lifetime.

Unlike local pointers, Ironclad C++ allows references to be used as function return values, mainly to support common code idioms, including chaining function or method calls on an object (*e.g.*, `std::cout`) but restricts the expressions that can be returned as references. In general, any value with a lifetime that will persist through the function or method call may be returned safely. The result of the dereference of an `aptr` or a `ptr` can be returned as a reference because the referred location must be in the heap or globals. Ironclad C++ limits the expressions that may be returned by reference to the dereference of an `aptr` or a `ptr`, reference function parameters, the dereference of the `this` pointer, and class members (of the class that the method was called on). Intuitively, these expressions are allowed because the location they point to must have a lifetime that is at least as long as the lifetime of the return value.

Rule (Reference Return Values)**.** *A reference return value may only be initialized from the dereference of an `aptr` or a `ptr`, from a reference function parameter, from the dereference of the `this` pointer, or from a class member.*

Although we did not identify any such cases in our benchmarks, it is possible that a valid program will not conform to the above static restrictions on reference return values. For any such cases, Ironclad C++ provides the `ref<T>` class,

which is used as a return value. The `ref<T>` class provides nothing other than an implicit conversion to a `T&`, which performs a dynamic check, similar to the local pointer dynamic check, to ensure that the location held by the `ref<T>` refers to valid memory.

Ironclad C++ prohibits the use of reference class members due to the possible unsafety from initializer lists. Otherwise, a class member with reference type of an object on the heap could be initialized to point to a stack location through the use of a constructor initializer list. Reference class members are rare in C++ code and generally discouraged because the fact that they cannot be reseated makes them inflexible. For example, an assignment operator cannot properly assign a new location to a reference. We encountered only a single case in `astar` in which we refactored a reference class member to be a `ptr` instead.

Rule (Reference Class Members)**.** *Reference class members are disallowed.*

6. Formalizing the Pointer Lifetime Invariant

To ensure that this collection of rules satisfies the pointer lifetime invariant, we prove that the invariant is maintained during execution using a formalism of a core fragment of Ironclad C++ called *Core Ironclad*. In the name of simplicity, Core Ironclad omits most language features of C++ that do not directly impinge on this part of the system. For example, inheritance, templates, and overloading do not interact with pointer lifetimes, so they are left out. References are also excluded in the interest of simplicity as the interesting interactions with references arise in conjunction with other language features that are not in the formalism, *e.g.* overloading. What is left is a small, C++-like core calculus with just enough features to cover the interesting parts of Ironclad's pointer lifetime checking system.

For a complete account of Core Ironclad, including a complete language description and full proofs of type safety, please see our technical report [28].

6.1 Locations and the Store

Figure 6 gives the syntax of Core Ironclad. The language consists of statements s and expressions e, evaluated in a context Δ that contains the program's set of class definitions. The store Σ contains both the stack and the heap which maps locations ℓ to values v. The only values that need to be formalized to verify the pointer lifetime invariant are pointers ptr and lptr and objects of class type C. Within the store, a value is tagged as being either a ptr or an lptr. These contain a *pointer-value*, pv, which is either a valid location or null or the "head" of an object of type C.

Locations have the form $x^n@y_1..y_m$, where x^n is a base location with name x and store level n, and $y_1..y_m$ is a path in the style of Rossie and Friedman [18]. The heap is located at store index 0 and the stack grows with increasing indices starting at index 1. The store height index n disambiguates between two variables that have the same name but exist in different stack frames.

The type-checking and evaluation relations appear in Figure 7. All of these judgments keep track of the current store height to ensure that variable lookup picks out the appropriate values. Ψ is the store typing which maps locations to types τ.

Objects are represented in the store using paths. For example, consider the following class definitions:

$$\mathsf{struct}\, C\{\, \mathsf{ptr}\langle C\rangle\, a;\,\};$$
$$\mathsf{struct}\, B\{\, C\, c;\,\};$$

A declaration `B x;` in the `main` function creates a `B` object that lives at base location x^1 with value tag B. The fields of this object are represented as locations with paths rooted at x^1. For example, the location $x^1@c$ refers to the `C` sub-object within `x`, and $x^1@c.a$ refers to the pointer within that sub-object.

A curious feature of Core Ironclad is that all expressions evaluate to locations. The value denoted by an expression is stored at the location the expression evaluates to. This greatly simplifies the evaluation semantics in several places. Notably, assignment simply copies the location denoted by the right-hand side into the location denoted by the left-hand side. As the only primitive values that we have are pointers, this amounts to making the left-hand pointer refer to the same location referred to by the right-hand side. We discuss these rules in more detail in Section 6.3.

6.2 Method Calls

Core Ironclad embeds the active call frame within the term language using an internal frame expression $\{s;\, \mathsf{return}\, e\}$ rather than using continuations or a separate stack syntax. This is similar in style to Core Cyclone [13]. For example, the (abbreviated) rule for method calls

$$\frac{\vdots \qquad\qquad\qquad\qquad\qquad\qquad \text{EVAL\_EXP\_METH}}{\Sigma' = \ldots[\mathit{this}^{n+1}@\ \mapsto \mathsf{lptr}\,(\ell_1)]}$$
$$\frac{}{(\Sigma, \ell_1.f(x_2{}^{n_2}@\pi_2)) \xrightarrow[\mathsf{exp}]{n}_{\Delta} (\Sigma', \{s;\, \mathsf{return}\, e\})}$$

replaces a method invocation with an appropriate frame expression. While the this pointer cannot be wrapped in a smart pointer in the implementation, the Ironclad validator ensures that the `this` pointer behaves like a `lptr`. Consequently, Core Ironclad treats the this pointer as a lptr rather than a third pointer type distinct from ptr and lptr. The remaining premises (not shown) look up the appropriate method body to invoke and set up the arguments and local variable declarations in the store. Because the method body is evaluated at any index n, we typecheck the method at index 0 (*i.e.*, has no dependence on prior stack frames) and prove a lemma that shows we can lift that result to the required index n.

When the statement of a frame expression steps, the stack count must be one higher than that of the frame expression

Types	τ	$::=$	$\mathsf{ptr}\langle\tau\rangle \mid \mathsf{lptr}\langle\tau\rangle \mid C$
Surface exprs	e	$::=$	$x \mid \mathsf{null} \mid e.x \mid e_1.f(e_2)$
			$\mid \mathsf{new}\,C() \mid \&e \mid *e$
Internal exprs	e	$::=$	$\ell \mid \{s; \mathsf{return}\,e\} \mid \mathsf{error}$
Statements	s	$::=$	$e_1 = e_2 \mid s_1; s_2$
			$\mid \mathsf{skip} \mid \mathsf{error}$
Class decls	cls	$::=$	$\mathsf{struct}\,C\,\{\overline{\tau_i\,x_i;}^{\,i}\,meths\};$
Methods	$meth$	$::=$	$\tau_1\,f(\tau_2\,x)\{\overline{\tau_i\,x_i}^{\,i}; s; \mathsf{return}\,e\}$

Expression variables	x, y		
Locations	ℓ	$::=$	$x^n@y_1..y_m$
Pointer values	pv	$::=$	$\ell \mid \mathsf{bad\_ptr}$
Values	v	$::=$	$\mathsf{ptr}\,(pv) \mid \mathsf{lptr}\,(pv) \mid C$
Store	Σ	$::=$	$\cdot \mid \Sigma[\ell \mapsto v]$
Store typing	Ψ	$::=$	$\cdot \mid \Psi[\ell : \tau]$
Class context	Δ	$::=$	$\{cls_1..cls_m\}$
Programs	$prog$	$::=$	$\Delta; \mathsf{void\ main}()\ \{s\}$

Figure 6. Core Ironclad Syntax

	expressions	statements
typing	$\Psi \vdash^{\Delta;n}_{\mathsf{exp}} e : \tau$	$\Psi \vdash^{\Delta;n}_{\mathsf{stmt}} s\,\mathsf{ok}$
evaluation	$(\Sigma, e) \longrightarrow^{n}_{\mathsf{exp}\,\Delta} (\Sigma', e')$	$(\Sigma, s) \longrightarrow^{n}_{\mathsf{stmt}\,\Delta} (\Sigma', s')$

Figure 7. Typing judgments and evaluation relations

to reflect the new stack frame:

$$\textsc{eval\_exp\_body\_cong1}\quad \frac{(\Sigma, s) \longrightarrow^{n+1}_{\mathsf{stmt}\,\Delta} (\Sigma', s')}{(\Sigma, \{s; \mathsf{return}\,e\}) \longrightarrow^{n}_{\mathsf{exp}\,\Delta} (\Sigma', \{s'; \mathsf{return}\,e\})}$$

Finally, when a frame expression returns, the frame expression is replaced with the location of the return value.

$$\textsc{eval\_exp\_body\_ret}\quad \frac{\begin{array}{l} x^n\ \textbf{fresh for}\ \Sigma \\ \Sigma_2 = \mathsf{copy\_store}(\Sigma, \ell, x^n) \\ \Sigma' = \Sigma\Sigma_2 \backslash (n+1) \end{array}}{(\Sigma, \{\mathsf{skip}; \mathsf{return}\,\ell\}) \longrightarrow^{n}_{\mathsf{exp}\,\Delta} (\Sigma', x^n@\,)}$$

The premises copy the return value ℓ into a fresh base location x^n in the caller's frame (taking care to copy additional locations if the returned value is an object) and pop the stack. The result of the method call then becomes that fresh location. Note that no dynamic check (say, to make sure that the value stored in x^n is valid after the function returns) is needed here because the type system enforces that the returned value cannot be a lptr in this rule:

$$\textsc{type\_exp\_body}\quad \frac{\begin{array}{l} \forall \tau'.\tau \neq \mathsf{lptr}\langle\tau'\rangle \\ \Psi \vdash^{\Delta;n+1}_{\mathsf{stmt}} s\,\mathsf{ok} \qquad \Psi \vdash^{\Delta;n+1}_{\mathsf{exp}} e : \tau \end{array}}{\Psi \vdash^{\Delta;n}_{\mathsf{exp}} \{s; \mathsf{return}\,e\} : \tau}$$

6.3 Pointer Semantics

With respect to the pointer lifetime invariant, the most interesting rules concern the assignment of pointers as well as the constraints we place on their values in the store. The simplest case is when we assign between two ptr values where we simply overwrite the left-hand pointer with the right-hand pointer in the store.

$$\textsc{eval\_stmt\_assign\_ptr\_ptr}\quad \frac{\begin{array}{ll} \Sigma(\ell_1) = \mathsf{ptr}\,(pv_1) & \Sigma(\ell_2) = \mathsf{ptr}\,(pv_2) \\ \Sigma' = \Sigma\,[\ell_1 \mapsto \mathsf{ptr}\,(pv_2)] & \end{array}}{(\Sigma, \ell_1 = \ell_2) \longrightarrow^{n}_{\mathsf{stmt}\,\Delta} (\Sigma', \mathsf{skip})}$$

When assigning a (non-null) lptr to a ptr, we verify that the lptr does indeed point to the heap by checking that the store index of the location referred to by the lptr is 0.

$$\textsc{eval\_stmt\_assign\_ptr\_lptr}\quad \frac{\begin{array}{ll} \Sigma(\ell_1) = \mathsf{ptr}\,(pv_1) & \Sigma(\ell_2) = \mathsf{lptr}\,(x^0@\pi) \\ \Sigma' = \Sigma\,[\ell_1 \mapsto \mathsf{ptr}\,(x^0@\pi)] & \end{array}}{(\Sigma, \ell_1 = \ell_2) \longrightarrow^{n}_{\mathsf{stmt}\,\Delta} (\Sigma', \mathsf{skip})}$$

When the lptr does not point to the heap, we raise an error.

$$\textsc{eval\_stmt\_assign\_ptr\_lptr\_err}\quad \frac{\begin{array}{ll} \Sigma(\ell_1) = \mathsf{ptr}\,(pv_1) & n' \neq 0 \\ \Sigma(\ell_2) = \mathsf{lptr}\,({x^{n'}}@\pi_2) & \end{array}}{(\Sigma, \ell_1 = \ell_2) \longrightarrow^{n}_{\mathsf{stmt}\,\Delta} (\Sigma, \mathsf{error})}$$

Finally, when assigning (non-null) lptrs, the dynamic check ensures that the lptr being assigned to out-lives the location it receives by comparing the appropriate store indices.

$$\textsc{eval\_stmt\_assign\_lptr\_lptr}\quad \frac{\begin{array}{ll} \Sigma({x_1}^{n_1}@\pi_1) = \mathsf{lptr}\,(pv_1) & \\ \Sigma(\ell_2) = \mathsf{lptr}\,({x_2}^{n_2}@\pi_2) & \\ \Sigma' = \Sigma\,[{x_1}^{n_1}@\pi_1 \mapsto \mathsf{lptr}\,({x_2}^{n_2}@\pi_2)] & n_2 \leq n_1 \end{array}}{(\Sigma, {x_1}^{n_1}@\pi_1 = \ell_2) \longrightarrow^{n}_{\mathsf{stmt}\,\Delta} (\Sigma', \mathsf{skip})}$$

If the lptr does not out-live its new location, we raise an error like the ptr-lptr case above.

In addition to standard typing rules for statements and expressions, Core Ironclad enforces a consistency judgment over the store through the store typing. Two of these binding consistency rules for pointers capture the pointer lifetime invariant.

The first rule concerns ptrs and requires that the location pointed to by the ptr is on the heap (at index 0).

$$\frac{\Sigma(x^n@\pi) = \mathsf{ptr}\,(x'^{n'}@\pi') \qquad n' = 0 \quad \Psi(x'^{n'}@\pi') = \tau}{\Psi;\Sigma \vdash_{\mathsf{st1}} x^n@\pi : \mathsf{ptr}\langle\tau\rangle\,\mathsf{ok}}\ \text{CONS\_BINDING\_PTR}$$

The second rule concerns lptrs and requires that lptrs only exist at base locations without paths (not embedded within an object) and that the location pointed to by a particular lptr is in the same stack frame or a lower one.

$$\frac{\Sigma(x^n@\,) = \mathsf{lptr}\,(x'^{n'}@\pi') \qquad n' \leq n \quad \Psi(x'^{n'}@\pi') = \tau}{\Psi;\Sigma \vdash_{\mathsf{st1}} x^n@\,: \mathsf{lptr}\langle\tau\rangle\,\mathsf{ok}}\ \text{CONS\_BINDING\_LPTR}$$

6.4 The Pointer Lifetime Invariant

Invariant (Pointer lifetime)**.** *For all bindings of the form* $[{x_1}^{n_1}@\pi_1 \mapsto \mathsf{ptr}\,(pv)]$ *and* $[{x_1}^{n_1}@\pi_1 \mapsto \mathsf{lptr}\,(pv)]$ *in* Σ*, if* $pv = {x_2}^{n_2}@\pi_2$ *(*i.e.*, is non-null) then* $n_2 \leq n_1$*.*

We can now present our theorem that states that the pointer lifetime invariant is preserved by execution. We make use of the summary judgment $\mathsf{wf}\,(\Delta,\Psi,\Sigma,k)$, which asserts that the class and method context Δ, the store typing Ψ, and the store Σ are all consistent with one another and contain only bindings at or below stack height k. These checks also include the pointer lifetime invariant. Therefore, $\mathsf{wf}\,(\Delta,\Psi,\Sigma,k)$ implies that the pointer lifetime invariant holds for Σ.

Theorem (Pointer lifetime invariant is preserved)**.** *If* $\mathsf{wf}\,(\Delta,\Psi,\Sigma,k)$*,* $\Psi \vdash^{\Delta;n}_{stmt} s\,\mathsf{ok}$*, and* $(\Sigma,s) \longrightarrow^{n}_{stmt\,\Delta}(\Sigma',s')$*, where* k *is the maximum stack height used within the statement* s*, then the pointer lifetime invariant holds in the store* Σ'*.*

This theorem is a straightforward corollary of a standard type preservation lemma. That is, preservation tells us that the wf judgment is preserved by evaluation.

Lemma 1 (Preservation)**.**

1. *If* $\mathsf{wf}\,(\Delta,\Psi,\Sigma,|s|+n)$*,* $\Psi \vdash^{\Delta;n}_{stmt} s\,\mathsf{ok}$*, and* $(\Sigma,s) \longrightarrow^{n}_{stmt\,\Delta}(\Sigma',s')$*, then there exists* Ψ' *such that* $\mathsf{wf}\,(\Delta,\Psi',\Sigma',|s|+n)$*,* $\Psi \vdash^{\Delta;n}_{stmt} s'\,\mathsf{ok}$*, and* $\Psi \subseteq_n \Psi'$*.*
2. *If* $\mathsf{wf}\,(\Delta,\Psi,\Sigma,|e|+n)$*,* $\Psi \vdash^{\Delta;n}_{exp} e : \tau$*, and* $(\Sigma,e) \longrightarrow^{n}_{exp\,\Delta}(\Sigma',e')$*, then there exists* Ψ' *such that* $\mathsf{wf}\,(\Delta,\Psi',\Sigma',|e|+n)$*,* $\Psi' \vdash^{\Delta;n}_{exp} e' : \tau$*, and* $\Psi \subseteq_n \Psi'$*.*

Core Ironclad also supports a standard progress lemma, completing the proof of type safety.

Lemma 2 (Progress)**.**

1. *If* $\Psi \vdash^{\Delta;n}_{stmt} s\,\mathsf{ok}$ *and* $\Psi \vdash_{st} \Sigma\,\mathsf{ok}$ *then* s *is* skip*,* error*, or* $(\Sigma,s) \longrightarrow^{n}_{stmt\,\Delta}(\Sigma',s')$*.*
2. *If* $\Psi \vdash^{\Delta;n}_{exp} e : \tau$ *and* $\Psi \vdash_{st} \Sigma\,\mathsf{ok}$ *then* e *is* ℓ*,* error*, or* $(\Sigma,e) \longrightarrow^{n}_{exp\,\Delta}(\Sigma',e')$*.*

In Core Ironclad, well-formed terms take a step or step to error. error terms arise only when an attempted pointer assignment breaks the pointer lifetime invariant. This corresponds to the dynamic checks outlined in Section 6.3.

We prove preservation at stack height $|s|+n$ (respectively $|e|+n$) where n is the height of the stack up to statement s (respectively e) and $|s|$ (respectively $|e|$) is the number of additional stack frames embedded in the subject of the evaluation. The extra condition $\Psi \subseteq_n \Psi'$ says that new store typing Ψ' has all the bindings of the old store typing Ψ up to stack height n. The complete details of the proofs of these theorems can be found in our companion technical report.

7. Validator for Ironclad C++

Although many of the rules of Ironclad C++ are simple enough for a well-intentioned programmer to follow unaided, ensuring safety requires the use of a static syntactic validation tool to certify the code's conformance to the Ironclad C++ subset. The role of the validator is to ensure that any violations of memory/type safety will be caught by either: (1) the standard C++ static type checker or (2) the dynamic type cast and bounds checks performed by the smart pointers. For example, the validator ensures that no raw pointer or union types remain. For heap-precise garbage collection, the validator ensures that user-defined `mark()` methods correctly identify all pointers, pointer-containing members, and inherited base classes of each precisely marked class. To prevent dangling pointers to stack objects via dynamic lifetime checking, the validator checks that all uses of address-of, the `this` pointer, and conversions from stack arrays to pointers immediately enter an `lptr` or `laptr`. The syntactic validator examines the initializer lists for each class constructor to ensure that pointer class members are safely initialized. The validator ensures that references are not used as class members. Finally, the validator ensures that expressions returned by reference match one of the following constructs: dereference of an `aptr` or `ptr`, a reference parameter, the dereference of the `this` pointer, or a class member.

Once the code passes the static syntactic validator, the C++ type-checker then statically enforces the remaining type safety properties. For example, unsafe casts and `void*` will not type-check in validated code because Ironclad C++ smart pointers explicitly do not support them. Similarly, array indexing and pointer arithmetic operators are not defined on `ptr<T>` objects, ensuring the disallowed use of such operators will be caught during compilation.

Our implementation of the static validator builds upon LLVM's Clang compiler front-end [20]. Static validation is performed on the AST after both preprocessing and template instantiation have been performed by the Clang front-end. The Ironclad C++ validator applies simple, local checks to

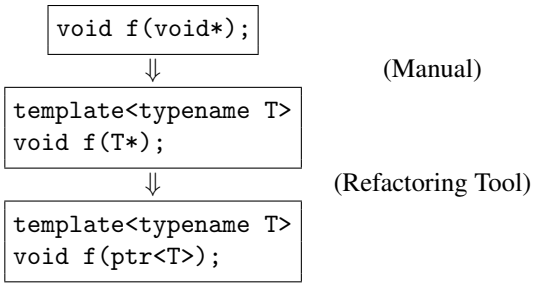


Figure 8. Common refactoring to remove `void*` pointers. First we manually add templates. Then the refactoring tool adds `ptr`s.

type declarations and expressions. None of the checks used by the static validator require complicated analysis.

Although currently implemented as a stand-alone checking tool, an alternative implementation would be to integrate the static validation into a compiler such that the validator can be invoked with a command line flag during compilation (much as GCC's `-std=` flag ensures the code conforms to a specific language standard).

8. Experiences Refactoring to Ironclad C++

To evaluate the usability and applicability of Ironclad C++, we refactored multiple performance benchmarks (from the SPEC and Parsec suites) and an open-source key-store database written in C++ to Ironclad C++. The open-source database, `leveldb`, was developed at Google and uses custom data structures, including a skip list and a LRU cache. Table 2 characterizes the C++ language features used by these applications and details the nature of the code changes performed to refactor to Ironclad C++. Overall, we were able to successfully refactor 50K lines of C/C++ code to Ironclad C++. We performed a series of manual and automated refactoring steps to transform these programs, and the majority of the code transformations were performed by our semi-automatic refactoring tool (Section 8.3).

8.1 Step 1: Moving from C to C++

Ironclad C++ requires the use of a C++ compiler to compile all our benchmarks. C++, unlike C, does not allow a `void*` to be implicitly converted to a `T*`. Hence, three C benchmarks from the SPEC benchmark suite could not be compiled using a C++ compiler without a few manual modifications. We manually added explicit casts to the offending expressions. In `sjeng`, we changed the name of a variable named "`this`", which is a C++ keyword. Once all of the programs could be compiled using a C++ compiler, we further modified them to use C++'s `new` function for allocation rather than C's `malloc`, which requires unsafe casts from `void*`. To match `new`, we also replaced `free` with `delete`.

8.2 Step 2: Increasing Type-safety

After refactoring the code to compile with a C++ compiler and use C++ allocation and deallocation functions, we performed a few additional code modifications to prepare the code for automated refactoring. As noted in Section 3.1, `void*` pointers are not permitted by Ironclad C++. The process of replacing `void*` pointers with type-safe constructs is not generally automatable because it may require recognizing and extracting an inheritance hierarchy or adding template parameters for more than one type. Rather than attempting to perform this refactoring automatically, we instead chose to manually replace occurrences of `void*`, as shown in Figure 8. A few additional benchmark-specific code modifications were necessary, but these modifications were typically simple and did not require any deep understanding of the algorithms or the data structures used in the benchmarks. For example, `lbm` used an errant cast from a `double***` to a `double**` that was fixed by correcting the pointer types to allow compilation without an unsafe cast.

8.3 Step 3: Automated Refactoring

Once we manually modified the benchmarks to use type-safe features, we applied a custom automated refactoring tool to the code. This refactoring tool performs simple automated code modifications, including modifying pointer type declarations (`T*` to `ptr<T>`), allocation and deallocation sites (`new T(...)` to `new_obj<T>(...)` and `delete p` to `p.destruct()`), and type-casts (`(T*)p` to `cast<T>(p)`). As shown in Table 2, the majority of the code modifications necessary to refactor C and C++ code to Ironclad C++ are performed by the refactoring tool. The refactoring tool is meant to be used only once to aid the initial transformation. The refactoring tool is built upon LLVM's Clang compiler front-end.

The refactoring tool could simply use `aptr<T>` as a replacement for every `T*`, but doing so results in unnecessary dynamic checks whenever a more-efficient `ptr<T>` would suffice. Thus, the refactoring tool implements a best-effort analysis similar to the whole program pointer type inference in CCured [25] to determine whether to replace a `T*` with an `aptr<T>` or a `ptr<T>`. This whole program analysis is run once, during refactoring, and it is not required for future validation or further manual refactoring. The refactoring tool analysis accounts for the use of address-of, `this`, and assignments from stack allocated arrays to determine which pointers require dynamic lifetime checking (`lptr` and `laptr`). The refactoring tool can also optionally generate `mark()` methods for heap-precise garbage collection.

8.4 Step 4: Post-Refactoring Modifications

Considering that C++ is a large language with many corner cases, the refactoring tool does not automatically handle every possible code modification. We also performed a few manual code changes following refactoring. Given that

Benchmark	Lang	Class	Ptrs	Refs	LoC	Manual Code Changes				Automated Code Changes			
						C++	Alloc	Pre	Post	Ptr Types	SysFunc	Alloc	Casts
blackscholes	C	1	20%	N/A	405	0	5	4	2	5	5	4	0
bzip2	C	2	47%	N/A	5731	16	41	28	14	224	21	30	30
lbm	C	1	57%	N/A	904	1	3	20	6	49	7	2	4
sjeng	C	2	46%	N/A	10544	45	24	164	158	310	82	30	0
astar	C++	25	7%	35%	4280	0	60	2	7	72	15	76	2
canneal	C++	3	27%	29%	2817	0	0	2	9	72	3	2	1
fluidanimate	C++	4	7%	7%	2785	0	1	0	2	85	7	44	1
leveldb	C++	66	49%	24%	16188	0	0	160	149	1028	69	195	33
namd	C++	15	46%	7%	3886	0	0	0	44	265	11	69	10
streamcluster	C++	5	37%	0%	1767	0	25	2	2	63	17	9	21
swaptions	C++	1	39%	0%	1095	0	11	1	12	63	3	0	9

Table 2. Characterization of the evaluated programs. From left to right, benchmark name, source language, number of classes/structs, % pointer declarations, % reference declarations, lines of code, manually refactored lines of code, and automatically refactored lines of code.

refactoring is intended to be performed only once, the number of lines modified post-refactoring was relatively small in most cases. For example, our pointer type inference implementation sometimes missed a nested increment operation on an array pointer and inferred that the pointer was therefore a singleton. This error is easily caught through the type-checking done by the C++ compiler (`ptr<T>` does not overload the increment operator). A more mature refactoring tool would avoid these minor refactoring errors.

Two notable outliers required more manual refactoring than the rest of the refactored programs (`sjeng` and `leveldb`). We describe the code modifications below.

sjeng In `sjeng`, there were 154 uses of `f(&A[0])` to pass a pointer to the first element of a stack allocated array as an argument to a function `f`. In Ironclad C++, the `lptr<T>` constructed from the result of `&A[0]` contains no information about the size of `A` and therefore assumes that is has size 1. We modified the code to use `f(A)` instead, which retains the correct array size information. Where `&A[i]` is used with some non-zero index `i`, the `A.offset(i)` method provided by the array and array pointer classes was used to create a new array pointer with the correct offset and size.

In addition, the `gen` function stores a pointer to a stack allocated array in a global variable, which is not permitted in Ironclad C++. This global variable is set on each entry to the `gen` function and used by other functions that were called from `gen`. In place of the unsafe use of the global variable, we modified the code to simply pass the pointer parameter from `gen` to the rest of the functions that required it. This change was conceptually straightforward but required modifying 137 lines of code.

leveldb `leveldb` required a larger number of manual code modifications compared to the benchmarks with fewer total lines of code. In particular, the `Slice` class in `leveldb` contains a constructor that accepts a `const char *`. Refactoring this constructor to Ironclad C++ converts the parameter to type `aptr<const char>`. However, in cases where a string literal was originally used to call a function with a `Slice` parameter, the Ironclad C++ code required more than the one implicit user-defined conversion allowed by the C++ standard [16]. Thus, we added an explicit conversion from string literals to `aptr<const char>` at each call site.

8.5 Step 5: Performance Tuning

Finally, we identified modifications to three of the benchmark programs as examples of performance tuning that can reduce the performance penalty of providing memory safety. These optimizations serve as starting points for other source-level optimizations that the Ironclad C++ API could provide.

bzip2 In `bzip2`, the `generateMTFValues` function creates a temporary pointer to a stack allocated array in a doubly nested loop. In the Ironclad C++ version of this code, the temporary pointer becomes an `laptr`, which must initialize its `lowerBound` on each iteration of the outer loop. Further, the temporary pointer was used in pointer arithmetic, which is relatively expensive (compared to the `array` index operator, which does not need to check a lower bound) for Ironclad C++ `aptr` and `laptr` types. To improve the performance of this code, we replaced the temporary pointer with an integer index and used array indexing off of the original stack allocated array instead of pointer arithmetic on the temporary. This optimization reduced the performance penalty from 53% to 35%.

streamcluster The `streamcluster` benchmark spends the majority of its runtime in the distance function, which computes the pointwise distance between two vectors. Due to the use of a tight for-loop with repeated indexing into the input vectors, our initial Ironclad C++ version of this benchmark suffered from unnecessarily high bounds checking overheads of 70%. To reduce this overhead, we replaced the loop with a call to a `reduce` function, provided by the Ironclad C++ library, that simply bounds checks the

start and end indices of the reduction on both input arrays, and then runs the computation at unchecked speeds. With this optimization, the `streamcluster` benchmark executes with no measurable overhead compared to the original.

swaptions `Swaptions` spends most of its runtime performing operations on vectors and matrices of floating pointer numbers. For matrix structures, `swaptions` uses an array-of-array-pointers to approximate a two-dimensional array (*i.e.*, `aptr<aptr<double>>`). These structures are inefficient in two ways. First, creating the structure requires multiple memory allocations. Second, due to the additional metadata used by `aptr`, each two-dimensional index operation (*i.e.*, `A[i][j]`) must first load the address, current index, and size stored in the first level array and then load the double stored in the second level array. We replaced the `aptr<aptr<double>>` structures with a `matrix<double>`. The `matrix` class provides a proper two-dimensional array by overloading `operator() (unsigned int x, unsigned int y)`. In this way, the `matrix` class performs two bounds checks (one on each index) and then returns the data, avoiding the additional indirection required by the `aptr<aptr<double>>` structure. This optimization reduced the performance penalty from 67% to 45%.

8.6 Libraries

The C++ STL The C++ Standard Template Library (STL) provides common containers and algorithms. The underlying implementations of these containers use unchecked pointer operations and are not safe by default. Only a few of our benchmarks used the STL (`canneal` and `leveldb`), so instead of refactoring the entire STL (approximately 100k lines of code) to conform to Ironclad C++, we modified key parts of the STL to emulate the checking that a fully refactored version would perform. We performed four major modifications on the containers used in our benchmarks. First, we changed the default allocator to be the `gc_allocator`. Second, we inserted bounds checks on all array operations, including the indexing operators for `string` and `vector`. Third, we modified methods that accepted or returned raw pointers to instead use Ironclad C++ smart pointers. Finally, we modified the iterators to each container to avoid accessing invalid memory. For `string` and `vector`, this was as simple as replacing the raw pointer iterator with an `aptr`. For `map` and `set`, we modified the `tree_iterator` to avoid iterating past the root or end nodes of the tree.

Atomics In C++11, atomic operations on primitive data types and pointers were introduced as part of the STL. To support atomic operations on pointers, Ironclad C++ provides template specializations for the `atomic<T>` type for `ptr` and `aptr`; `lptr` and `laptr` do not require support for atomic operations because pointers to them cannot escape a thread's stack. The template specialization for `ptr` mirrors the standard atomic pointer implementation because `ptr` contains a single pointer field. Unlike `ptr`, the template specialization of `atomic` for `aptr` requires the use of a lock or transaction to atomically update the three fields of the `aptr`. Using these `atomic` pointers, any program that was data-race free prior to using Ironclad C++ remains data-race free.

External Libraries Ideally, library source-code would also be refactored to Ironclad C++, but we acknowledge that it may not always be feasible. In such cases, Ironclad C++ protects what it can, while begrudgingly allowing the program to call unsafe library code through methods on the smart pointer classes that allow the underlying raw pointer to be passed to a library call. This functionality is provided for the case where refactoring is not possible, similarly to how Java provides the JNI for access to unsafe C/C++ code. This behavior is optional and can be disabled.

8.7 Bug Detection Effectiveness

As a coarse sanity check on the implementation of the Ironclad C++ library, we tested Ironclad C++ on multiple suites of known bugs, including selected programs from BugBench (`gzip`, `man`, `ncompress`, and `polymorph`) [22], thirty array-out-of-bounds vulnerability test cases from the NIST Juliet Suite [26], and the Wilander test suite [35]. As expected, Ironclad C++ safely aborted on all buggy inputs. We note that these tests are not definitive proof of Ironclad C++'s soundness or correctness, of course.

9. Experimental Evaluation

The previous section established the feasibility of bringing full type and memory safety to C++ at the cost of refactoring programs to conform to the Ironclad C++ rules, but it did not evaluate the performance and memory usage cost of enforcing such safety at runtime. This section describes our prototype implementation and presents runtime overhead results, including experiments to isolate the overhead added by the various aspects of Ironclad C++. In addition, we present results that indicated the overheads from garbage collection are low, that heap-precise collection reduces memory consumption versus purely conservative collection, and dynamic lifetime checking is faster than heapification.

9.1 Implementation and Experimental Methods

We use the programs refactored and optimized in the previous section (from the SPEC benchmark suite, the Parsec benchmark suite, and an open-source database—`leveldb`) to evaluate the runtime overheads of enforcing safety. The benchmarks were compiled using LLVM/Clang version 3.2 C++ compiler, and our test system contains an Intel 2.66Ghz Core2 processor. The Ironclad C++ library includes implementations of the various smart pointer classes and safe versions of various C standard library functions. We modified the `libcxx` STL implementation to provide safe iterators, bounds-checked array access operations, and interfaces that accept and return smart pointers instead of raw point-

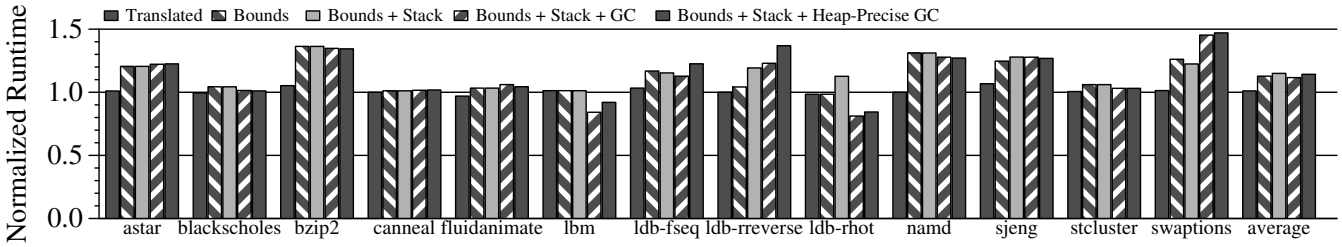


Figure 9. Normalized runtimes for refactored Ironclad C++ code with (adding checking from left to right): no checking, bounds checked arrays, safe stack allocations, safe heap allocations, and heap-precise garbage collection.

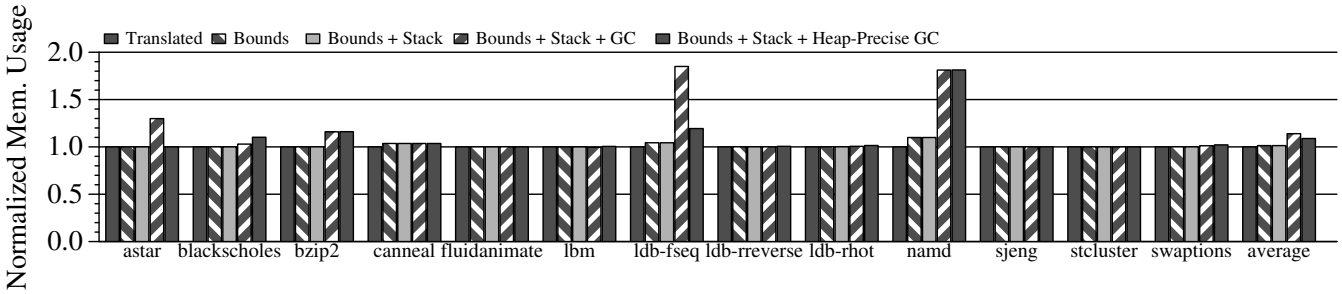


Figure 10. Normalized memory usage for Ironclad C++ code with (from left to right) bounds checking metadata, bounds and stack checking metadata, both metadata with conservative GC, and both metadata with heap-precise GC.

ers. To reduce overhead, the smart pointer implementation uses Clang's `always_inline` function attribute to ensure the compiler inlines the dereference and indexing operators. We used Valgrind's Massif tool to measure memory usage overheads.

We used the Boehm-Demers-Weiser conservative garbage collector both as the baseline garbage collector and as the basis for the heap-precise garbage collector [7]. To implement the heap-precise collector, we extended the marking implementation of the Boehm-Demers-Weiser collector to call the `mark()` method of allocations initialized for precise collection rather than pushing the allocation's address onto the conservative collector's mark stack. Any addresses passed back to the collector from `mark()` are added to the mark stack. Each potential address passes through the existing checks employed by the Boehm-Demers-Weiser collector for duplicate marking and blacklisting.

9.2 Overall Performance

The overall performance overhead for bringing type and memory safety to the refactored programs is just 12% on average. Figure 9 shows these results, and it also includes results for multiple configurations to show the impact of each aspect of Ironclad C++. The left-most bar in each group ("Translated") shows the normalized execution time of the fully refactored, strongly typed benchmarks when dynamic bound checking, dynamic lifetime checking, and the garbage collector are all disabled. These results indicate there is negligible overhead from replacing raw pointers with smart pointers. The second bar from the left in each group of Figure 9 ("Bounds") shows bounds checking is by far the most significant contributor to the overall performance overhead.

9.3 Overheads of Garbage Collection

Figure 9 shows that the runtime overhead of garbage collection is negligible in our benchmarks. Although perhaps surprising, our benchmarks are not typical of programs used in garbage collection studies, which are generally selected for their frequent allocation behavior. For example, several of our benchmarks allocate memory only during initialization and do not deallocate memory—resulting in extremely rare collection invocation (less than once per second for many benchmarks). The benchmark that collects most frequently (six hundred times per second), `swaptions`, incurs an additional 23% performance penalty due to garbage collection.

The garbage collector increases memory usage by 14% on average and up to 85% for `leveldb-fillseq` (Figure 10) when compared to explicit deallocation with the same underlying memory allocator. One caveat is that the allocator underlying the conservative collector uses more space on average than Clang's default memory allocator (by 29%) even when operating in explicit memory deallocation mode; if this overhead is included, the total memory overhead of GC rises to 43% (not shown on Figure 10).

9.4 Benefits of Heap-Precise Collection

Figure 10 also shows the impact of Ironclad's heap-precise extension to the garbage collector. In most cases, the heap-precise collector provides no appreciable reduction of memory usage, but in two cases — `astar` and `leveldb-fillseq` — the pure-conservative collector suffers due to imprecise identification of heap pointers. When applying heap-precise collection, these programs' memory usage is reduced by 28% and 66%, respectively,

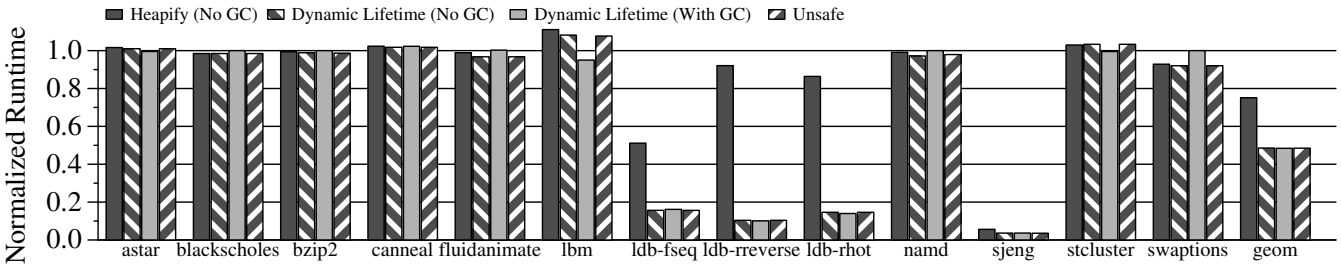


Figure 11. Runtime normalized to using heapification with GC. Bars from left to right are heapification with unsafe deallocation, dynamic lifetime checking (safe stack allocations), dynamic lifetime checking with GC (safe allocations), and unsafe deallocation.

compared to the unmodified conservative collector. The memory overheads of `bzip2` and `namd` do not improve under heap-precise collection due to stack-allocated arrays (which are not tagged) with elements that are misidentified as pointers. The overall average memory overhead of GC vs. explicit memory allocation drops from 14% to 9% with the addition of heap-precise collection.

We observe a 2% performance penalty for heap-precise garbage collection. Upon investigation, we found this penalty is not due to slower tracing time using `mark()` methods during garbage collection; the penalty is a result of changes in data layout introduced by our implementation of heap-precise collection (which wraps each allocation with a templated wrapper class with a virtual table pointer for calling the correct `mark()` method). We confirmed this hypothesis by including the same header when using the completely conservative collector, which yielded the same runtime overheads.

9.5 Benefits of Dynamic Lifetime Checking

Dynamic lifetime checking incurs less than 1% overhead over the baseline of providing no safety checking for stack deallocation. We originally observed overheads of 17% in `bzip2`, but these overheads were due to the creation of an `laptr` temporary in a tight loop. The optimization described in Section 8.5 eliminated the overhead from dynamic lifetime checking in `bzip2`.

In Figure 11, we compare dynamic lifetime checking to the other notable alternative for stack allocation safety: heapification. On average, heapification is 2× slower than dynamic lifetime checking. We observed two situations in which heapification lead to increased performance overheads: a stack-allocated array escaping to a function (occurs in `sjeng`) and calling a method on a stack-allocated object (occurs in `leveldb`). As a result, dynamic lifetime checking is faster than heapification by 27.5× (`sjeng`), 6.2× (`ldb-fseq`), 9.1× (`ldb-rreverse`), and 6.4× (`ldb-rhot`).

In almost all cases, dynamic lifetime checking avoids the use of heapification for enforcing stack deallocation safety. The runtime performance of dynamic lifetime checking is nearly identical to the performance of code with no safety checking for stack deallocation.

10. Conclusion

Ironclad C++ brings type safety to C++ at a runtime overhead of 12%. We demonstrated the feasibility of refactoring C and C++ code to Ironclad C++ with the help of a semi-automatic refactoring tool. With heap-precise garbage collection, Ironclad C++ provides an optional interface for precisely identifying heap pointers, which was shown to decrease average memory usage of garbage collection. We investigated both heapification and dynamic lifetime checking for enforcing stack deallocation safety and found that dynamic lifetime checking offered flexibility, memory control, and limited source code modifications as compared to heapification. Overall, our experiences and experimental results indicate that Ironclad C++ has the potential to be an effective, low-overhead, and pragmatic approach for bringing comprehensive memory safety to C++.

Acknowledgments

We would like to thank Emery Berger, Mathias Payer, and John Regehr for their comments and suggestions about this work. This research was funded in part by the U.S. Government by ONR award N000141110596 and NSF grants CNS-1116682 and CCF-1065166. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References

[1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, Boston, MA, 2001.

[2] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the SIGPLAN 1994 Conference on Programming Language Design and Implementation*, June 1994.

[3] J. Bartlett. Mostly-Copying Garbage Collection Picks Up Generations and C++. Technical report, DEC, 1989.

[4] E. D. Berger and B. G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 158–168, June 2006.

[5] H.-J. Boehm. Space Efficient Conservative Garbage Collection. In *Proceedings of the SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 197–206, June 1993.

[6] H.-J. Boehm and M. Spertus. Garbage collection in the next C++ standard. In *Proceedings of the 2009 International Symposium on Memory Management*, pages 30–38, June 2009.

[7] H.-J. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. *Software — Practice & Experience*, 18(9):807–820, Sept. 1988.

[8] D. Colvin, G. and Adler, D. *Smart Pointers - Boost 1.48.0.* Boost C++ Libraries, Jan. 2012. `www.boost.org/docs/libs/1_48_0/libs/smart_ptr/smart_ptr.htm`.

[9] D. Dhurjati and V. Adve. Backwards-Compatible Array Bounds Checking for C with Very Low Overhead. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 162–171, 2006.

[10] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory Safety Without Runtime Checks or Garbage Collection. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems (LCTES)*, pages 69–80, 2003.

[11] D. Edelson and I. Pohl. A Copying Collector for C++. In *Proceedings of The 18th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 51–58, Jan. 1991.

[12] D. Gay, R. Ennals, and E. Brewer. Safe Manual Memory Management. In *Proceedings of the 2007 International Symposium on Memory Management*, Oct. 2007.

[13] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-Based Memory Management in Cyclone. In *Proceedings of the SIGPLAN 2002 Conference on Programming Language Design and Implementation*, June 2002.

[14] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proc. of the Winter Usenix Conference*, 1992.

[15] M. Hirzel and A. Diwan. On the type accuracy of garbage collection. In *Proceedings of the 2000 International Symposium on Memory Management*, pages 1–11, Oct. 2004.

[16] International Standard ISO/IEC 14882:2011. *Programming Languages – C++*. International Organization for Standards, 2011.

[17] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the 2002 USENIX Annual Technical Conference*, June 2002.

[18] J. Jonathan G. Rossie and D. P. Friedman. An Algebraic Semantics of Subobjects. In *Proceedings of the 17th SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA)*, Nov. 2002.

[19] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[20] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, page 75, 2004.

[21] D. Lomet. Making Pointers Safe in System Programming Languages. *IEEE Transactions on Software Engineering*, pages 87 – 96, Jan. 1985.

[22] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for Evaluating Bug Detection tools. In *In PLDI Workshop on the Evaluation of Software Defect Detection Tools*, June 2005.

[23] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the SIGPLAN 2009 Conference on Programming Language Design and Implementation*, June 2009.

[24] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management*, June 2010.

[25] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems*, 27(3), May 2005.

[26] *NIST Juliet Test Suite for C/C++*. NIST, 2010. `http://samate.nist.gov/SRD/testCases/suites/Juliet-2010-12.c.cpp.zip`.

[27] Y. Oiwa. Implementation of the Memory-safe Full ANSI-C Compiler. In *Proceedings of the SIGPLAN 2009 Conference on Programming Language Design and Implementation*, pages 259–269, June 2009.

[28] P.-M. Osera, R. Eisenberg, C. DeLozier, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Core Ironclad. Technical Report MS-CIS-13-06, University of Pennsylvania, 2013.

[29] J. Pincus and B. Baker. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns. *IEEE Security & Privacy*, 2(4):20–27, 2004.

[30] J. Rafkind, A. Wick, M. Flatt, and J. Regehr. Precise Garbage Collection for C. In *Proceedings of the 2009 International Symposium on Memory Management*, June 2009.

[31] M. S. Simpson and R. K. Barua. MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime. In *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 199–208, 2010.

[32] B. Stroustrup. A Rationale for Semantically Enhanced Library Languages. In *Library-Centric Software Design*, page 44, 2005.

[33] B. Stroustrup. Software Development for Infrastructure. *Computer*, 45:47–58, Jan. 2012.

[34] E. Unger. *Severe memory problems on 32-bit Linux*, April 2012. `https://groups.google.com/d/topic/golang-nuts/qxlxu5RZAI0/discussion`.

[35] J. Wilander and M. Kamkar. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2003.

[36] W. Xu, D. C. DuVarney, and R. Sekar. An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 117–126, 2004.