3-28-2013

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

Christian DeLozier
University of Pennsylvania, delozier@cis.upenn.edu

Richard A. Eisenberg
University of Pennsylvania, eir@cis.upenn.edu

Santosh Nagarakatte
Rutgers University - New Brunswick/Piscataway, santosh.nagarakatte@rutgers.edu

Peter-Michael Osera
University of Pennsylvania, posera@cis.upenn.edu

Milo Martin
University of Pennsylvania, milom@cis.upenn.edu

See next page for additional authors

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-13-05.

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

Abstract

C++ remains a widely used programming language, despite retaining many unsafe features from C. These unsafe features often lead to violations of type and memory safety, which manifest as buffer overflows, use-after-free vulnerabilities, or abstraction violations. Malicious attackers are able to exploit such violations to compromise application and system security. This paper introduces Ironclad C++, an approach to bring 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, and thus Ironclad C++ programs can be compiled using standard, off-the-shelf C++ compilers. However, not all valid C++ programs are valid Ironclad C++ programs. To determine whether or not a C++ program is a valid Ironclad C++ program, Ironclad C++ uses a syntactic source code validator that statically prevents the use of unsafe C++ features. For properties that are difficult to check statically Ironclad C++ applies dynamic checking to enforce memory safety using templated smart pointer classes. Drawing from years of research on enforcing memory safety, Ironclad C++ utilizes and improves upon prior techniques to significantly reduce the overhead of enforcing memory safety in C++. To demonstrate the effectiveness of this approach, we translate (with the assistance of a semi-automatic refactoring tool) and test a set of performance benchmarks, multiple bug-detection suites, and the open-source database leveldb. These benchmarks incur a performance overhead of 12% on average as compared to the unsafe original C++ code, which is small compared to prior approaches for providing comprehensive memory safety in C and C++.

Disciplines

Computer Engineering

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-13-05.

Author(s)

Christian DeLozier, Richard A. Eisenberg, Santosh Nagarakatte, Peter-Michael Osera, Milo Martin, and Stephan A. Zdancewic

CIS Technical Report #MS-CIS-13-05
March 28, 2013

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
{delozier, eir, santoshn, posera, milom, stevez}@cis.upenn.edu

Abstract

C++ remains a widely used programming language, despite retaining many unsafe features from C. These unsafe features often lead to violations of type and memory safety, which manifest as buffer overflows, use-after-free vulnerabilities, or abstraction violations. Malicious attackers are able to exploit such violations to compromise application and system security. This paper introduces Ironclad C++, an approach to bring 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, and thus Ironclad C++ programs can be compiled using standard, off-the-shelf C++ compilers. However, not all valid C++ programs are valid Ironclad C++ programs. To determine whether or not a C++ program is a valid Ironclad C++ program, Ironclad C++ uses a syntactic source code validator that statically prevents the use of unsafe C++ features. For properties that are difficult to check statically Ironclad C++ applies dynamic checking to enforce memory safety using templated smart pointer classes. Drawing from years of research on enforcing memory safety, Ironclad C++ utilizes and improves upon prior techniques to significantly reduce the overhead of enforcing memory safety in C++.

To demonstrate the effectiveness of this approach, we translate (with the assistance of a semi-automatic refactoring tool) and test a set of performance benchmarks, multiple bug-detection suites, and the open-source database leveldb. These benchmarks incur a performance overhead of 12% on average as compared to the unsafe original C++ code, which is small compared to prior approaches for providing comprehensive memory safety in C and C++.

1. Introduction

C and C++ are widely used programming languages for implementing web browsers, native mobile applications, compilers, databases, and other infrastructure software [28]. 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 [25].

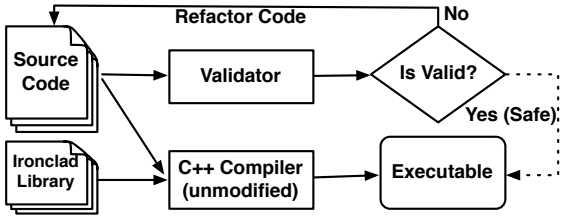

Figure 1. Workflow for Coding with Ironclad C++

Recognizing this problem, many approaches have been proposed to prevent memory safety violations or even enforce full memory safety in C and C-like languages [2, 4, 9, 10, 12, 13, 16, 20–22, 27, 31]. 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++ does provide advanced constructs that enable type-safe programming, such as templates and dynamically checked type-casts.

This paper presents Ironclad C++, an approach to bring comprehensive memory and type safety to C++. Ironclad C++ is, in essence, a library-augmented 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 additional language extensions or compiler changes are required. As shown in Figure 1, Ironclad C++ code is compiled using an unmodified off-the-shelf C++ compiler but Ironclad C++ includes a syntactic validation pass that checks whether the input is a legal program.

In the following paragraphs, we describe some key principles of efficient type safety for C, survey their implementations in prior work, and describe 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 typically requires whole-program analysis at compile time or language extensions. For example, CCured [22] uses a whole-program type inference at compile time to distinguish between singleton and array pointers, Cyclone [16] 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 that can significantly increase the overhead of dynamic checking [20, 24] 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, 21, 31], Ironclad C++ facilitates the use of conservative garbage collection [7] by targeting two challenges 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, 26]. 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, 14, 26], 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 alone fails to prevent dangling pointers to the 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 [22]. 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, prior smart pointers schemes could not 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, 26].

Experimentally evaluating Ironclad C++. 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.

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

- 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.

Type	Capabilities	Safety Checks
ptr	Dereference	Null Check
lptr	Dereference	Null Check
	Receive address-of object	Lifetime Check
	Receive *this* pointer	Lifetime Check
aptr	Dereference	Bounds Check
	Index	Bounds Check
	Arithmetic	No 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 allowed, but is bounds checked when the array pointer is dereferenced

Table 1. This table details the capabilities and safety checks required for each pointer type. As more letters are added to the type, the capabilities increase, but the efficiency decreases due to additional required checks.

2. 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 6, statically enforces that disallowed constructs are not used. This section first describes safety without considering arrays or memory deallocation (Section 2.1), then adds support for arrays and pointer arithmetic (Section 2.2). Further sections discuss memory deallocation safety for the heap (Section 3) and stack (Section 4).

2.1 Strong Static Typing with `ptr<T>`

Ironclad C++ requires that all raw C++ pointer types are replaced 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>(args)`, which uses `new` internally but returns a `ptr<T>` (rather than returning a `T*`). Accordingly, the following C++ code:

```
Rectangle* r = new Rectangle(2, 5);
```

Would be rewritten in Ironclad C++ as:

```
ptr<Rectangle> r = new_obj<Rectangle>(2, 5);
```

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.*

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` to a `ptr<T>`. 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) on 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<U>(...)`, 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 [15]. 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. Thus, the only possible illegal memory access via a `ptr` is dereferencing the `NULL` pointer. The overloaded dereference operations (`*` and `->`) in `ptr` check for `NULL` prior to performing the dereference. Though a `NULL` dereference can be detected as a segmentation fault, the explicit check is necessary to prevent field accesses on `NULL` pointers. If a program attempts to access the member `foo->x` (`foo == NULL`) of type `struct Foo{int A[1000000]; int x;}`, the address accessed is actually 0x3D0900. If that address is mapped, a SEGFAULT will not occur. Checking for `NULL` before dereference also catches the dereference of a `ptr` that resulted from an invalid dynamically checked pointer cast.

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;
}
```

Example: Stack Allocation Safety

```
void compute(int * p, int * q){
  int x = 0;
  p = q;
  p = &x;
}
void create() {
  int * p, * q;
  compute(p,q);
}
```

```
void compute(lptr<int> & p, lptr<int> & q){
  int x = 0;
  p = q;  // Same scope, check passes
  p = &x; // Deeper scope, check fails
}
void create() {
  lptr<int> p, q;
  compute(p,q);
}
```

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

2.2 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* f = new Foo[number];
```

Would be rewritten in Ironclad C++ as:

```
aptr<Foo> f = new_array<Foo>(number);
```

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`.

2.3 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 initializing 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> y;
  Foo() : x(*y){}
};
```

To ensure proper initialization, smart pointer initializers must appear in an initializer list before any dereference of

the smart pointer, any use of the `this` pointer as a function argument, and any method calls. The static validator enforces this requirement.

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

2.4 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 safe 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++. `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 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>`).*

3. Support for Heap-Precise Garbage Collection

Along with strong typing and bounds checking, deallocation safety (*i.e.*, no dangling pointers) is the final requirement for complete type and memory safety. To prevent dangling pointers to heap allocated objects, Ironclad C++ uses a conservative garbage collector to delay deallocation until a time at which it is known to be safe.[1] Garbage collection prevents dangling pointers without the overheads associated with dynamic checking on each pointer dereference [2, 21] or maintaining reference counts [12]. In most of the benchmarks we tested, conservative garbage collection is fast and incurs low memory overheads (see Section 8). However, one concern with conservative garbage is memory leaks due to non-pointer data that "looks" like a pointer [14, 26, 29].

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 (though 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, 26].

Building on these prior works, Ironclad C++ adopts opt-in *heap-precise* garbage collection. Ironclad C++'s heap-precise garbage collector treats the program roots conservatively (*e.g.*, stack, registers, and globals) but precisely marks heap allocations using optional user-defined mark methods. To support incremental adoption, heap allocations are marked conservatively by default, but if the type being allocated is a precise type (*i.e.*, the programmer has added a `mark()` method and the class inherits non-virtually from the `IroncladPreciseGC` class), the garbage collector uses the `mark()` method during collection to precisely identify pointers and pointer-containing members in the object. A class's `mark()` method must call `mark()` on all its pointer fields, object fields, and base classes from which it inherits. Although provided by the programmer, unlike prior systems that utilize programmer-supplied marking [3, 26], each `mark()` method is statically verified for correctness by the Ironclad C++ validator (Section 6). For allocating arrays of primitive data types, Ironclad C++ includes template specializations that informs the garbage collector that the allocation contains no pointers and thus does not need to be marked.

4. 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.

[1] For domains in which garbage collection is not applicable, Ironclad C++ still provides type and bounds safety.

4.1 The Perils of Heapification

A common approach for preventing use-after-free errors for stack allocations in garbage collected system is simply to restrict stack allocations by employing heapification [22], 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++ codes, 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 8.5).

4.2 Dynamic Lifetime Checking

To reduce the need for heapification, Ironclad C++ provides stack allocation safety by allowing pointers to stack allocations to escape but controlling how the escaped pointers are used during execution. It does this by introducing 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, 18], but these efforts have been focused on purely static enforcement through sophisticated program analyses. Local pointers in Ironclad C++ combine static enforcement and dynamic checking, providing flexibility and simplifying the necessary analysis. Local pointers, and the rules regarding their use, allow Ironclad C++ to enforce the following invariant:

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

For pointers to the heap, this invariant is enforced through the use of garbage collection, which guarantees that a heap allocation will not be deallocated while a reference to it remains. For the stack, Ironclad C++ must ensure that when the address within a stack frame escapes, the address does not escape to a pointer with a longer lifetime than the stack frame.

Local pointers record the lower bound on addresses that they may point to.[2] Through a combination of static restrictions and dynamic checks, these local pointers are allowed to point only to heap-allocated values or values at the same level or above in the call stack. In the concrete implementation, shown in Figure 3, a local pointer records the current stack pointer in its `lowerBound` field upon construction. The local pointer then applies a dynamic check on pointer assignment (by overloading the assignment operator) to determine if it will outlive its new referent.

Local pointers ensure that each assignment into or out of the local pointer will not create a dangling reference.

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

```
template<typename T> class lptr {
  T * data;
  size_t lowerBound;

  lptr(T * newData) : data(newData) {
    lowerBound = getCurrentStackPointer();
    if(newData < lowerBound) {
      // Points to an infinite lifetime object
      lowerBound |= 1;
    }
  }

  operator ptr<T> () {
    // Check that object has infinite lifetime
    if( lowerBound & 1 == 0 ) exit(-1);
    ...
  }

  lptr<T>& operator= (lptr other) {
    if(other.data != NULL &&
       (other.lowerBound & 1) == 0 &&
       (lowerBound ^ 1) > other.data) exit(-1);
    ...
  }
};
```

Figure 3. Pseudo-C++ implementation of local pointer checking. Casts necessary for type checking have been removed for clarity.

For all stack safety checks, pointers of type `aptr<T>` and `ptr<T>` are assumed to hold only addresses that point to values stored in the heap or globals. The checks required for local pointers can be split into the following cases.

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`, and a flag in the `lptr` is set to indicate that it currently holds the address of a heap or global value.

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. As explained in the previous case, a flag bit in the `lptr` is set when it receives the address of a heap or global value. Therefore, if the flag bit is set, the address may be assigned into the `ptr`. If the flag bit is not set, then the address held by the `lptr` points to a value stored on the stack and cannot be held by a `ptr`, so 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.

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

Figure 4. Case in leveldb under which dynamic lifetime checking could not avoid heapification

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

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, 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.*

Rule (Local Pointer Location)**.** *A local pointer may not exist on the heap.*

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 codes we examined, the single example requiring heapification occurred in our conversion of the benchmark `leveldb`. The relevant code is shown in Figure 4. Here, the address of a stack value is stored in the field of a heap object, which caused the local pointer assignment check to fail at runtime. Even though the code does not actually create a dangling reference, Ironclad C++ could not provide this guarantee. Therefore, the programmer must heap-allocate the object to ensure safety.

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 are not allowed to be `NULL` and must therefore be initialized as soon as they are declared. In Ironclad C++, the creation of a `NULL` reference is not possible. Once a reference has been initialized, the location that the reference points to cannot change. Thus, Ironclad C++ needs to prevent the initialization of a reference to memory that is currently invalid.

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.*

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 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.

5. 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*. For the sake of brevity, this paper includes only those features of Core Ironclad that deal with the pointer lifetime invariant. For a complete account of Core Ironclad, including a complete language description and full proofs of type safety, please see our technical report draft.[3]

5.1 Locations and the Store

Core Ironclad consists of statements s and expressions e, evaluated in a context Δ with classes (omitting inheritance)

[3] The Core Ironclad TR is available at `http://goo.gl/nbOqO`.

and methods. The store Σ contains both the stack and the heap which maps locations ℓ to values v. A value is tagged as being either a ptr or an lptr, along with a *pointer-value*, pv, which is either a valid location or null. 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 [17]. 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 with the same name but existing in different stack frames.

Paths allow locations to refer to the inner member of an object. For example, consider the following 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. The location $x^1@c.a$ refers to the a field of the `C` sub-object within `B`.

To simplify the system and to support temporary objects, expressions evaluate to locations. The value denoted by the expression is stored at the location the expression evaluates to. Thus, we can use the same evaluation relation for expressions on either side of an assignment.

5.2 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.

$$\frac{\Sigma(\ell_1) = \mathsf{ptr}\,(pv_1) \qquad \Sigma(\ell_2) = \mathsf{ptr}\,(pv_2) \qquad \Sigma' = \Sigma\,[\ell_1 \mapsto \mathsf{ptr}\,(pv_2)]}{(\Sigma, \ell_1 = \ell_2) \underset{\mathsf{stmt}}{\longrightarrow}^n_\Delta (\Sigma', \mathsf{skip})}$$

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.

$$\frac{\Sigma(\ell_1) = \mathsf{ptr}\,(pv_1) \qquad \Sigma(\ell_2) = \mathsf{lptr}\,(x^0@\pi) \qquad \Sigma' = \Sigma\,[\ell_1 \mapsto \mathsf{ptr}\,(x^0@\pi)]}{(\Sigma, \ell_1 = \ell_2) \underset{\mathsf{stmt}}{\longrightarrow}^n_\Delta (\Sigma', \mathsf{skip})}$$

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.

$$\frac{\Sigma({x_1}^{n_1}@\pi_1) = \mathsf{lptr}\,(pv_1) \qquad \Sigma(\ell_2) = \mathsf{lptr}\,({x_2}^{n_2}@\pi_2) \qquad \Sigma' = \Sigma\,[{x_1}^{n_1}@\pi_1 \mapsto \mathsf{lptr}\,({x_2}^{n_2}@\pi_2)] \qquad n_2 \le n_1}{(\Sigma, {x_1}^{n_1}@\pi_1 = \ell_2) \underset{\mathsf{stmt}}{\longrightarrow}^n_\Delta (\Sigma', \mathsf{skip})}$$

In addition to standard typing rules for statements and expressions, Core Ironclad enforces a consistency judgment over the store by way of a store typing Ψ which maps locations to types τ. In particular, 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).

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

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

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

5.3 Statement of Theorem

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* $n_2 \le n_1$*.*

We now can 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. 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* $\mathsf{wf}\,(\Delta, \Psi, \Sigma, k)$*,* $\Psi \overset{\Delta;n}{\vdash_{\mathit{stmt}}} s\ \mathsf{ok}$*, and* $(\Sigma, s) \underset{\mathit{stmt}}{\longrightarrow}^n_\Delta (\Sigma', s')$*, 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 type preservation. That is, the standard type preservation lemma tells us that the wf judgment is preserved by evaluation.

Theorem (Preservation)**.**

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

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.

6. 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 validator ensures that any memory/type safety violation will be prevented by either: (1) the standard C++ type checker or (2) the dynamic checks performed by the smart pointers. Specifically, the validator checks 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 and reference class members are safely initialized. Finally, the validator checks 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 overloaded on the `ptr<T>` class, 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. Static validation is performed on the AST after preprocessing and template instantiation have already been performed by the clang front-end. The Ironclad C++ validator applies simple, local checks to 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 might be to integrate the static validation into the compiler that is invoked with a command line flag during compilation (much as GCC's `-std=` flag ensures the code conforms to a specific language standard).

7. 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 a few, key 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 7.3).

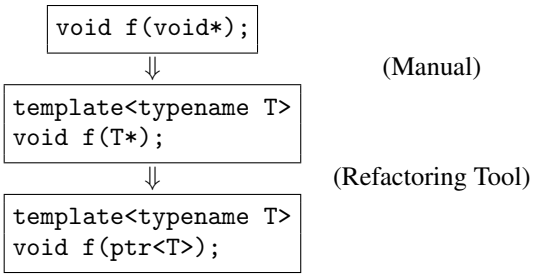

Figure 5. Common refactoring to remove `void*` pointers. First we manually add templates. Then the refactoring tool adds `ptr`s.

7.1 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`.

7.2 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 2.1, `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 5. 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 datastructures used in the benchmarks. For example, `lbm` used an errant cast from a `double***` to a `double**` that was removed by correcting the pointer types to allow compilation without an unsafe cast.

7.3 Step 3: Automated Refactoring

Once we manually modified the benchmarks to use type-safe features, we applied a custom automated refactoring

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

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 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 replace every `T*` with an `aptr<T>`, but this would cause the resulting code to execute unnecessary dynamic checks. Thus, the refactoring tool implements a best-effort analysis similar to the whole program pointer type inference in CCured [22] 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 also generates `mark()` methods for heap-precise garbage collection.

7.4 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 refactoring is intended to be performed only once, the number of lines modified post-refactoring were 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 iteration of the 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 `ptr<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 to 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 to 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 [15]. Thus, we added an explicit conversion from string literals to `aptr<const char>` at each call site.

7.5 Step 5: Performance Tuning

Finally, we identified modifications to three of the benchmark programs as examples of the sort of performance tun-

ing that can reduce the performance penalty of providing memory safety.

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 array indexing) 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 normalized runtime from 1.53x to 1.35x.

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. 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 runtime overhead from 1.67x to 1.45x.

7.6 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.

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.

7.7 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`) [19], thirty array-out-of-bounds vulnerability test cases from the NIST Juliet Suite [23], and the Wilander test suite [30]. 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.

8. Experimental Evaluation

The previous section established the feasibility of bringing full type and memory safety to C++ at the cost of refactoring programs to follow 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.

8.1 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, and our test system contains an Intel 2.66Ghz Core2 proces-

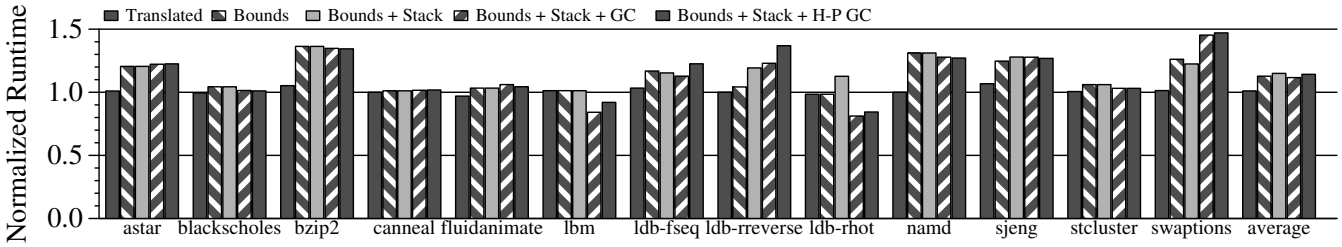

Figure 6. Normalized runtimes for (adding checking from left to right) refactored Ironclad C++ code with no checking, bounds checked arrays, safe stack allocations, safe heap allocations, and heap-precise garbage collection.

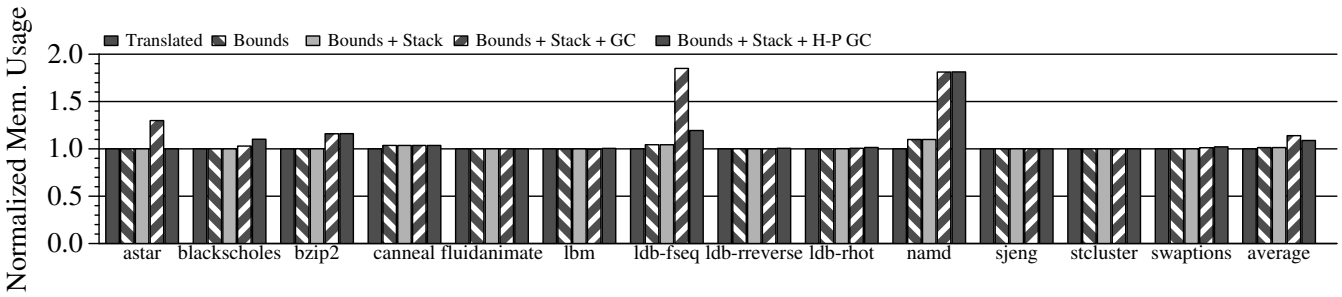

Figure 7. Normalized memory usage for Ironclad C++ code with (from left to right) bounds checking metadata, bounds and stack checking metadata, both metadata with conservative GC, and both metadata with heap-precise GC.

sor. 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 pointers. 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 implemented the heap-precise collector by extending the Boehm-Demers-Weiser conservative garbage collector [7] to use the marking algorithm described in Section 3. We used Valgrind's Massif tool to measure memory usage overheads.

8.2 Overall Performance

The overall performance overhead for bringing type and memory safety to the refactored programs is just 12% on average. Figure 6 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 that the performance of the original codes is the same as that of the refactored, strongly typed benchmarks when dynamic bound checking, dynamic lifetime checking, and the garbage collector are all disabled. These results indicate that there is negligible overhead from replacing raw pointers with smart pointers. The second bar from the left in each group of Figure 6 ("Bounds") shows that the performance overhead of bounds checking is responsible for almost all of the 12% overall performance overhead.

8.3 Overheads of Garbage Collection

Figure 6 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 never deallocate any 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 7) 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 7).

8.4 Benefits of Heap-Precise Collection

Figure 7 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, 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% runtime overhead increase for heap-precise garbage collection due to data layout changes from the inclusion of virtual pointers in each tagged allocation. This hypothesis was confirmed by tagging allocations but using the completely conservative collector, which yielded the same runtime overheads.

8.5 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 7.5 eliminated the overhead from dynamic lifetime checking in `bzip2`.

In Figure 8, we compare dynamic lifetime checking to the other notable alternative for stack allocation safety: heapification. On average, heapification is 2x 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 27.5x (`sjeng`), 6.2x (`ldb-fseq`), 9.1x (`ldb-rreverse`), and 6.4x (`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.

8.6 Summary

Ironclad C++ enforces comprehensive memory safety for C++ at an average runtime overhead of 12%. Without heap-precise garbage collection, Ironclad C++ incurs a memory overhead of 14%. Heap-precise garbage collection mitigates a few of the worst-case benchmarks for conservative collection and further reduces the memory overhead to 9%. By avoiding heapification, Ironclad C++'s dynamic lifetime checking provides a 2x speedup over heapified code. It may be possible to further reduce the overheads of memory safety in Ironclad C++ through additional source-level refactorings similar to those demonstrated in Section 7.5, compiler optimization, or static analysis.

9. 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++.

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 *PLDI*, June 1994.

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

[4] E. D. Berger and B. G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *PLDI*, pages 158–168, June 2006.

[5] H.-J. Boehm. Space Efficient Conservative Garbage Collection. In *PLDI*, pages 197–206, June 1993.

[6] H.-J. Boehm and M. Spertus. Garbage collection in the next C++ standard. In *ISMM*, pages 30–38, Jun 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 *ICSE*, pages 162–171, 2006.

[10] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory Safety Without Runtime Checks or Garbage Collection. In *LCTES*, pages 69–80, 2003.

[11] D. Edelson and I. Pohl. A Copying Collector for C++. In *POPL*, pages 51–58, Jan. 1991.

[12] D. Gay, R. Ennals, and E. Brewer. Safe Manual Memory Management. In *ISMM*, Oct. 2007.

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

[14] M. Hirzel and A. Diwan. On the type accuracy of garbage collection. In *ISMM*, pages 1–11, Oct. 2000.

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

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

[17] J. Jonathan G. Rossie and D. P. Friedman. An Algebraic Semantics of Subobjects. In *OOPSLA*, Nov. 2002.

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

[19] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for Evaluating Bug Detection tools. In

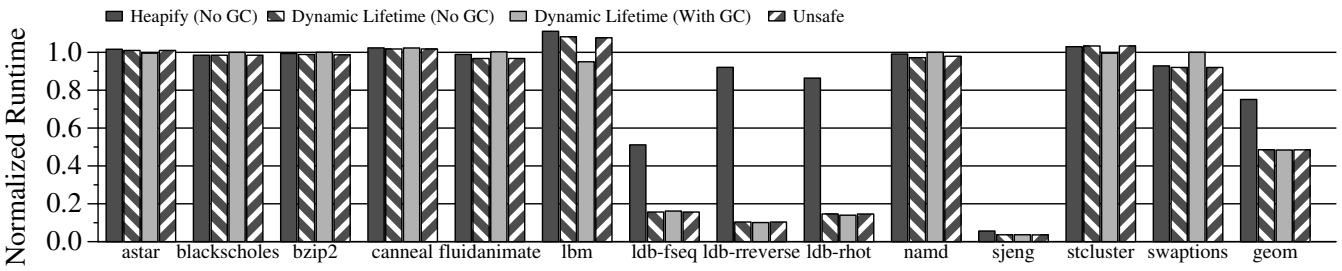

Figure 8. Runtime normalized to using heapification with GC. Bars from left to right are heapification with unsafe deallocation, dynamic lifetime checking (safe stack allocations), dynamic lifetime checking with GC (safe allocations), and unsafe deallocation.

In PLDI Workshop on the Evaluation of Software Defect Detection Tools, June 2005.

[20] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *PLDI*, June 2009.

[21] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *ISMM*, Jun 2010.

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

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

[24] Y. Oiwa. Implementation of the Memory-safe Full ANSI-C Compiler. In *PLDI*, pages 259–269, June 2009.

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

[26] J. Rafkind, A. Wick, M. Flatt, and J. Regehr. Precise Garbage Collection for C. In *ISMM*, Jun 2009.

[27] 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.

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

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

[30] J. Wilander and M. Kamkar. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. In *NDSS*, 2003.

[31] W. Xu, D. C. DuVarney, and R. Sekar. An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs. In *FSE*, pages 117–126, 2004.