Submitted to OOPSLA 2012 — April 2012

Ironclad C++

A Pragmatic Approach to Type-Safe Programming in 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

Unsafe languages, such as C and C++, suffer from security vulnerabilities due to unchecked memory accesses that can result in buffer overflow and use-after-free errors. Despite these shortcomings, a significant amount of software is developed using these unsafe languages. Modern languages prevent memory safety errors, but using a different language is often not an option due to project constraints or the need for extensive code modifications. On the other hand, many prior efforts have provided tools for enforcing memory safety in C and C++, but these tools often require invasive toolchain changes, such as a non-standard compiler or source-to-source translator. This paper presents Ironclad C++, a checked safe subset of C++. Ironclad C++ uses a source code validator to statically restrict the use of C++ features that may lead to memory safety errors. For properties that are difficult to enforce statically, Ironclad C++ enforces memory safety by inserting dynamic checks using templated pointer classes, provided by the Ironclad C++ library.

To demonstrate the effectiveness of this approach, we translate and test a set of performance benchmarks, multiple bug-detection suites, and the open-source database *leveldb*. On the benchmarks, we observe a runtime overhead of 20% as compared to the unsafe original C++ code. We find that modern, clean C++ code can be refactored to Ironclad C++ using simple transformations.

1. Introduction

In a recent online survey on programming languages [14], C++ ranked near the top in two seemingly conflicting categories: (1) widespread use of applications developed in the language and (2) the ability to shoot oneself in the foot. Much of the popularity of C++ stems from its flexibility and the low-level control available to the programmer. However, this flexibility and low-level control comes at the price of safety. Unintended programming errors such as buffer overflows (accessing location beyond the object), use-after-free errors (accessing memory locations that have been freed), and erroneous type casts allow programmers to inadvertently shoot themselves in the foot by breaking programming abstractions. More dangerously, these errors can be used as attack vectors by malicious attackers [27].

Type-safe programming languages, such as Java, C#, Go, and many others, prevent memory safety errors by enforcing type safety, dynamically checking for out-of-bound memory accesses, and relying on garbage collection. However, migrating to a new language often requires significant code modifications, which may not be feasible. Switching to a new language may also require a new tool-chain and new libraries, further impeding the possibility of migrating to a new language from C++.

Alternatively, many systems have been designed to retrofit C and C++ for memory safety with limited or no source code modifications [2, 3, 7, 8, 11, 13, 15, 20, 23–25, 31, 37]. These schemes introduce dynamic checks on each memory access through source-to-source translation, compiler transformation, binary instrumentation, or other methods. However, these approaches suffer from one or more limitations: large runtime overhead, significant tool-chain modifications, and inability to detect all memory safety errors.

As a compromise between the extremes of migrating to a new language and retrofitting safety, this paper proposes *Ironclad C++*, a checked, safe subset of C++. Ironclad C++ leverages the insight that C++—unlike C—provides advanced features for building safe abstractions, including templates, objects, operator overloading, and dynamically checked type casts. These language features enable—but do not force—programmers to create modern, clean, and strongly typed C++ programs that are highly immune to memory safety violations and security vulnerabilities.

For example, C++'s `string` class is much safer than C's `char*`-style strings, and should therefore be used in place of them. Nevertheless, these C-style strings have been carried over to C++, along with many other unsafe features, including: `void*` pointers for polymorphism, unchecked type casts, unsafe variable argument functions (*e.g.*, `printf()`), and raw unchecked pointer dereference, indexing, and arithmetic. For each of these unsafe constructs, C++ provides a safer alternative, respectively: templates and inheritance, dynamically checked type casts, variadic template functions, and `string` and other class libraries for safe pointer usage.

We hypothesize that, by providing programmers with sufficient safe alternatives via convenient libraries, we can define a practical, type- and memory-safe subset of C++. To achieve this goal of offering a pragmatic methodology for type-safe programming in C++, Ironclad C++ must address several concerns. First, to provide a usable memory safety solution, Ironclad C++ must maintain the benefits of using C++, such as performance and control over memory layout.

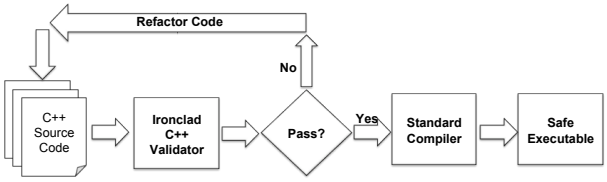

Figure 1. Workflow for Coding with Ironclad C++

Second, Ironclad C++ should be compatible with the existing C++ ecosystem, including standard compilers, debuggers, and other development tools. Third, to lower the barrier to adoption, the additional programmer effort required should be modest, both in initial porting efforts and day-to-day programming.

Ironclad C++ differs from other libraries and informal coding "best practice" recommendations because its static validation tool verifies the conformance of the code (using a simple, local analysis) just prior to compilation. Programs that conform to the Ironclad C++ requirements are still valid C++ programs, so an unmodified, standard C++ compiler can be used to generate the final program executable. Ironclad C++ programs therefore retain complete compatibility with debuggers and other development tools. Furthermore, such programs remain human-readable and maintainable. To facilitate the initial porting of C++ code, Ironclad C++ also provides a refactoring tool designed to alleviate the most tedious aspects of migrating a codebase from unsafe C++ to Ironclad C++. Figure 1 describes the Ironclad C++ workflow.

Ironclad C++ employs a number of techniques to enforce memory safety. In Ironclad C++, all raw pointers are replaced with one from a suite of newly designed template-based "smart pointer" wrapper classes—for example `int*` becomes `ptr<int>`. These pointer classes (and cast restrictions) enforce strong typing and array-bounds safety, but they do not address the issue of safe memory deallocation.

C++'s manual heap memory management and stack-allocated values are both potential pitfalls for memory safety violations. To prevent use-after-free dangling pointer dereference to heap values, Ironclad C++ relies on a conservative garbage collector. Although garbage collection eliminates heap deallocation errors, the automatic deallocation of a stack frame upon function exit can still be a source of dangling pointers. One potential solution, sometimes called *heapification*, is to prevent objects that escape the function from being stack allocated (and instead use the heap). However, we found that heapification resulted in significant performance overheads for some programs.

To prevent stack-related use-after-free errors without the overheads of full heapification, Ironclad C++ uses a novel combination of static and dynamic lifetime checks to enforce the invariant that a pointer to a stack object may not outlive the object itself. This approach is inspired by previous work on region-based memory management and static analysis of stack object usage [9, 10, 13]. To implement this hybrid approach, Ironclad C++ introduces another pointer template class (`lptr<t>`) that allows a stack address to escape but controls where the address can escape to.

To summarize, this paper offers four main contributions.

- We describe the design and implementation of Ironclad C++, a type-safe subset of C++ augmented with a safety-assist library, including a code validator tool (to ensure conformance) and refactoring tool (to assist in code conversion).
- We detail the hybrid static–dynamic checking for preventing use-after-free errors due to stack allocated data.
- We formalize and prove that the hybrid static–dynamic approach to safe stack allocation ensures the desired safety properties.
- We evaluate the usability and performance of Ironclad C++ by converting existing C/C++ programs into valid Ironclad C++ code and measure the associated performance overhead.

By using both the Ironclad C++ refactoring tool and manual code changes, we ported 50k lines of C/C++ code into conforming Ironclad C++. This paper describes our experiences with this effort, including a case study on converting a modern, open-source C++ application (the *leveldb* keystore database). As a measure of its effectiveness, Ironclad C++ detected all the bugs in multiple test suites designed for evaluating memory safety bug detectors. For performance-oriented benchmarks, Ironclad C++ introduces runtime overheads of 20% on average, in exchange for complete memory safety and full immunity to buffer overflow and use-after-free security vulnerabilities.

2. Ironclad C++

The Ironclad C++ safe subset of C++ is constructed by first providing the programmer with safe idioms via the Ironclad C++ library and then disallowing the use of unsafe language features in code that conforms to the Ironclad C++ rules. In this way, Ironclad C++ provides memory and type safety for C++ using a combination of static code validation and dynamic safety checks. A code validator, described in Section 5, 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), and then discusses memory deallocation safety for the heap (Section 2.3) and stack (Section 2.4).

2.1 Strong Static Typing with `ptr<T>`

Ironclad C++ replaces all raw pointers with instances of a smart pointer wrapper class, `ptr<T>`. If we ignore—just for the moment—arrays, pointer arithmetic, and type casts, the only possible unsafe memory access is dereferencing a `NULL` pointer. To catch such errors, the `ptr` class overloads the dereference operator to include an explicit `NULL` check. By not overloading the pointer arithmetic or array index operators, `ptr` also trivially prevents bounds errors (this restriction

is lifted in the next subsection). To allocate objects on the heap, Ironclad C++ provides `new_obj<T>(args)`, a wrapper function for `new` that 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++ 2011's variadic templates allow the function `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 its variants, described below), provided by the Ironclad C++ library. Raw pointers are disallowed.*

Supporting pointer type casts safely. By disallowing raw pointers, Ironclad C++ also implicitly disallows both `void*` pointers and unchecked pointer-to-pointer casts. To support pointer-to-pointer casts, Ironclad C++ provides a `cast<T>(...)` template function to safely cast a `ptr` to a `ptr<T>`. The `cast<T>(...)` function is wrapper over C++'s existing `dynamic_cast` operation, which is used to cast between members of a class hierarchy. The compilation may fail when the template is instantiated (*e.g.*, when attempting a cast that can not be safely checked dynamically). Or, the the underlying `dynamic_cast` may fail at runtime due to an incompatible type and the resulting pointer is set to `NULL`, which is then by caught by `ptr`'s `NULL` dereference check. Casting from `void*` or integers to a pointer is not supported by `dynamic_cast`, so this use of `dynamic_cast` statically disallows creating a `ptr` from an integer and `void*` pointer. The programmer must rewrite code to eliminate `void*` pointers by using either 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 on non-pointer types, such as `int`s and `double`s. C-style unions, however, are not allowed within the Ironclad C++.

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

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 is greater than 0 and less than `N` before returning the requested value. 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 of the pointer is `NULL`, so a separate `NULL` check is not needed. Arbitrary pointer arithmetic is also 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` 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: since `ptr` does not provide indexing or pointer arithmetic operators, it avoids the performance and storage overheads incurred by the bounds checking for `aptr`. The Ironclad C++ refactoring tool, described in Section 5, performs a whole program type-inference analysis [25] to determine whether a raw C++ pointer should be replaced with a singleton (`ptr<T>`) or array pointer (`aptr<T>`). In addition, the Ironclad C++ library allows 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`. Finally, the Ironclad C++ library also provides bounds checked versions of common standard library functions, such as `memcpy`. These wrapper functions

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†

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. †: arbitrary pointer arithmetic is allowed, but bounds checked when the array pointer is dereference.

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

// Bounds Checking
float * computeAreas(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;
}

// Stack Allocation Safety
Circle cached[7];
int cacheCounter = 0;

Circle * clone(Circle * circle){
  cached[cacheCounter % 7] = *circle;
}

void Circle::cacheSelf(){
  cache(this);
}
```

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

// Bounds Checking
aptr<float> computeAreas(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;
}

// Stack Allocation Safety
array<Circle,7> cached;
int cacheCounter = 0;

ptr<Circle> cache(lptr<Circle> circle){
  cached[cacheCounter % 7] = *circle;
}

void Circle::cacheSelf(){
  cache(this);
}
```

Figure 2. Comparison of C++ syntax and Ironclad C++ syntax.

take `aptr<T>` instances as arguments. As described in Section 5.1, the Ironclad C++ library also includes a `varray<T>` class to assist with safely implementing the "zero-sized array field as last member of an object as a variable-sized array" idiom used by some C++ programs.

Pointer initialization. To ensure that the dynamic checks used by the `aptr<T>` and `ptr<T>` classes catch all errors, the constructors for these pointers always initialize them appropriately. One particularly insidious corner case is when smart pointers appear in constructor initializer lists. To ensure proper initialization, all 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.

Rule (Pointer Initializers)**.** *An `aptr<T>` or `ptr<T>` object member must be initialized before use.*

2.3 Heap Deallocation Safety

Along with strong typing and bound checking, deallocation safety (*i.e.*, no dangling pointers) is the final requirement for complete memory safety. Preventing dangling pointers requires that the program does not attempt to access deallocated memory. The current instantiation of Ironclad C++ prevents such errors for heap objects by relying on a garbage collector, guaranteeing that memory allocated on the heap is not be deallocated while references to it remain. The use of garbage collection may raise concerns regarding its use in memory constrained environments and the non-deterministic behavior of collection. Exploring other possibilities, such as reference counting, remains as future work.

2.4 Stack Deallocation Safety

Although garbage collection can prevent use-after-free errors for heap-allocated objects, stack frames are not managed by the garbage collector and are still automatically allocated and deallocated on function entry and exit. In C++, the address of a stack object can escape to a location with a longer lifetime in a number of ways, including through the explicit use of the address-of operator or via the `this` parameter implicitly provided to every method call. To ensure deallocation safety for stack objects, Ironclad C++ provides *local pointers*, which are described in the next section.

3. Stack Deallocation Safety via `lptr`

3.1 Strawman: Heapification

One option for preventing use-after-free errors for stack allocations is *heapification* [25], which is the process of heap allocating an object that was originally allocated on the stack. To ensure deallocation safety through heapification, a sufficient condition is that any object whose address might escape the function must be allocated on the heap. Such heapification would heap allocate any variable that has its address taken, but in C++ method calls and non-default constructors also potentially allow the `this` pointer to escape. Thus, unless the definitions for all such methods and constructors are available for analysis, a static code analysis tool must

conservatively assume that any such method or constructor invocation leaks, forcing heapification.

When such draconian heapification is enforced, we find that it may severely degrade performance in some cases (further discussed in Section 7.2). To avoid those performance penalties, Ironclad C++ provides additional templated pointer classes that, cooperatively with the static code validator, prevent use-after-free errors for stack allocations while avoiding heapification in almost all cases.

3.2 Dynamic Lifetime Checking

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>`, referred to as *local pointers*. Prior work on preventing use-after-free errors has introduced some notion of a local pointer [9, 21], but these efforts have been focused on purely static enforcement. Local pointers in Ironclad C++ combine static enforcement and dynamic checking, simplifying the necessary analysis. Local pointers, and the rules regarding their use, allow Ironclad C++ to enforce the following invariant:

Invariant (Pointer lifetimes)**.** *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 stack allocations, Ironclad C++ must ensure that when the address of an allocation escapes, the address does not escape to a pointer with a longer lifetime than the allocation.

Local pointers record the lower bound on addresses that they may point to. 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 lower bound 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 referend.

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

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.

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

public:
  lptr(T * newData) : data(newData){
    lowerBound = getCurrentStackPointer();
    if(newData < lowerBound){
      // Address 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 &&
       (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.

Rule (Stack Pointers)**.** *Any possible address of a stack value 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 can 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.

```
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 (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.*

As a final rule for enforcing stack deallocation safety, a reference field may not be initialized directly from a reference constructor argument. Without this restriction, a reference field of a record can refer to a stack value that may have been deallocated.

Rule (Reference Initializers)**.** *A reference field of a record may not be initialized from a reference argument.*

With these lifetime safety dynamic checks and a few restrictions placed on the static use of local pointers by the Ironclad C++ code validator, Ironclad C++ provides deallocation safety for stack objects without the need for heapification in most situations. In our case study on `leveldb`, we identified a single example, shown in Figure 4, in which the use of local pointers was insufficient to avoid heapification. 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.

4. Formalizing the Ironclad Language

In light of the system's complexity, we developed a formalism to verify the correctness of Ironclad C++'s most interesting features. Trying to capture the complete Ironclad C++ system requires that we model much of the C++ language itself. This goal is impratical for our present purposes. Furthermore, type safety in the context of object-oriented programming [1, 16] as well as memory safety in the presence of pointers [18, 23–25], have both been thoroughly explored. Thus our formalism, *Core Ironclad*, focuses instead on showing the correctness of Ironclad's pointer lifetime invariant: if a pointer p points to a location ℓ, then ℓ outlives p. This invariant is maintained through the use of `lptrs` and `ptrs` in the Ironclad C++ implementation; correspondingly, our formalized lptrs and ptrs form the center of Core Ironclad.

Core Ironclad's metatheory turns out to be straightforward, but the design of the system itself required careful thought. This section briefly highlights the salient points of Core Ironclad focusing on those design decisions that relate to the Ironclad system and its lifetime checking system.[1]

4.1 Language

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. What is left is a small, C++-like core calculus with just enough features to cover the interesting parts of the pointer lifetime checking system.

4.2 Syntax

Figure 5 gives the syntax of the calculus. Core Ironclad is a statement-and-expression language where values are ptrs and lptrs, along with simple classes and methods. While there is no inheritance, classes are important because the this pointer is a potential source of trouble — as a built-in special form, Ironclad C++ cannot automatically wrap this in a smart pointer. Each class has a default, no-argument constructor that initializes its members to be invalid pointers and is invoked when calling new $C()$ or when creating an object on the stack. There are interesting technical issues that arise with the this pointer within constructors and destructors but they are addressed in prior work [28] and are ultimately orthogonal concerns. Methods are syntactically required to have a single argument and to return a result.

Core Ironclad enforces the pointer lifetime invariant with respect to pointers in the heap and in the stack (collectively, the *store*). Locations ℓ in the store are the combination of a base location x^n coupled with a path $x_1 \,..\, x_m$ in the style of Rossie and Friedman [19], Wasserrab [35], and Ramananandro [28]. The use of paths allows locations to refer to the inner members of a class. For example, consider the following definitions:

$$\begin{array}{l}\mathsf{struct}\ C\{\mathsf{ptr}\langle C\rangle\ a;\ \};\\ \mathsf{struct}\ B\{C\ c;\ \};\end{array}$$

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`.

Base locations in Core Ironclad are interesting because they not only represent a unique location in the store x but also its *position* n in the stack. The stack in Core Ironclad grows with increasing indices. For example, location x^3 sits one stack frame lower than x^4. Data in the heap exists at index 0. This is consistent with the intuition that earlier stack

[1] For a complete treatment of Core Ironclad including metatheory and proofs, please view our draft technical report at `http://www.cis.upenn.edu/~posera/papers/coreironclad-draft.pdf`

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

$$
\begin{array}{llll}
\text{Locations} & \ell & ::= & x^n@y_1\,..\,y_m \\
\text{Pointer values} & pv & ::= & \ell \mid \mathsf{bad\_ptr} \\
\text{Values} & v & ::= & \mathsf{ptr}\,(pv) \mid \mathsf{lptr}\,(pv) \\
\text{Store} & \Sigma & ::= & \cdot \mid \Sigma\,[\ell \mapsto v] \\
\text{Store typing} & \Psi & ::= & \cdot \mid \Psi\,[\ell : \tau] \\
\text{Class context} & \Delta & ::= & \{cls_1\,..\,cls_m\}
\end{array}
$$

Figure 5. Core Ironclad Syntax

frames outlive later frames; the heap simply outlives all stack frames.

The store Σ in Core Ironclad is a mapping from locations to store values v. The store typing Ψ is an analogous mapping from locations to types. Store values are class tags C or pointer values $\mathsf{lptr}\,(pv)$ and $\mathsf{ptr}\,(pv)$ that may reference live locations or the null location bad_ptr. A class tag is assigned to locations that represent the base of some object. In the above example, the location $x^1@$ (with the empty path) maps to the class tag B and $x^1@c$ maps to the class tag C. In this scheme, an object is defined by all locations that share the prefix of a location with a class tag value. The tags themselves are necessary to facilitate class method lookup.

4.3 Semantics

Typing and evaluation of statements and expressions, written

$$
\begin{array}{rr}
\Psi \vdash^n_\Delta s\ \mathsf{ok} & \text{Statement well-formedness} \\
\Psi \vdash^n_\Delta e : \tau & \text{Expression typing} \\
(\Sigma, s) \longrightarrow^n_\Delta (\Sigma', s') & \text{Statement evaluation} \\
(\Sigma, e) \longrightarrow^n_\Delta (\Sigma', e') & \text{Expression evaluation}
\end{array}
$$

are straightforward in Core Ironclad. However, several aspects of these semantics deserve special mention.

4.3.1 Locations

Core Ironclad is a *location-based* language. That is, rather than expressions evaluating to typical values, expressions evaluate to locations that can either be assigned into or used in a store lookup. The primary motivation for this design decision is to be able to model C++ objects that have temporary, yet stable storage such as those returned from methods. Such objects, while being temporary, can still be the subject of an assignment or mutated via method calls.

This set up also simplifies the evaluation rules in several places. For example, field access simply appends onto the current location's path.

$$
\overline{(\Sigma, (x^{n'}@\pi).x') \longrightarrow^n_\Delta (\Sigma, x^{n'}@\pi +\!\!+ x')}
$$

Also, Core Ironclad does not need to syntactically distinguish between left-values and right-values since the assignment rule can appropriately use the locations it receives.

4.3.2 The Stack

Because Core Ironclad deals with the stack explicitly, the typing and evaluation judgments all note the current stack frame n. This is important for type-checking and evaluating variables.

$$
\frac{\Psi(x^n@\,) = \tau}{\Psi \vdash^n_\Delta x : \tau}
$$

$$
\overline{(\Sigma, x) \longrightarrow^n_\Delta (\Sigma, x^n@\,)}
$$

A variable evaluates to a location in stack frame n where the names of the variable and the location coincide.

Core Ironclad embeds the active call frame within the term language using a 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

$$
\frac{\begin{array}{l}\vdots \\ \Sigma' = \ldots[this^{n+1}@ \mapsto \mathsf{lptr}\,(\ell_1)]\end{array}}{(\Sigma, \ell_1.f(x_2{}^{n_2}@\pi_2)) \longrightarrow^n_\Delta (\Sigma', \{s;\ \mathsf{return}\ e\})}
$$

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 an `lptr`. Consequently, Core Ironclad treats the this pointer as an 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 to reflect the new stack frame:

$$
\frac{(\Sigma, s) \longrightarrow^{n+1}_\Delta (\Sigma', s')}{(\Sigma, \{s;\ \mathsf{return}\ e\}) \longrightarrow^n_\Delta (\Sigma', \{s';\ \mathsf{return}\ e\})}
$$

Finally, when a frame expression returns, the frame expres-

sion is replaced with the location of the return value.

$$\frac{\begin{array}{l} x^n \textbf{ fresh for } \Sigma \\ \Sigma_2 = \mathsf{copy\_store}(\Sigma, \ell, x^n) \\ \Sigma' = \Sigma\,\Sigma_2 \backslash n+1 \end{array}}{(\Sigma, \{\mathsf{skip};\ \mathsf{return}\,\ell\}) \longrightarrow^n_\Delta (\Sigma', x^n@\,)}$$

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 is needed here because the type system enforces that the return value cannot be a lptr.

4.3.3 Dynamic Checks

In addition to null checks on dereference, dynamic checks are necessary during pointer assignment to ensure that the pointer lifetime invariant holds. The dynamic check when assigning a (non-null) lptr to a ptr verifies 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.

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

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.

$$\frac{\begin{array}{l} \Sigma({x_1}^{n_1}@\pi_1) = \mathsf{lptr}\,(pv_1) \\ \Sigma(\ell_2) = \mathsf{lptr}\,({x_2}^{n_2}@\pi_2) \\ n_2 \leq n_1 \\ \Sigma' = \Sigma\,[{x_1}^{n_1}@\pi_1 \mapsto \mathsf{lptr}\,({x_2}^{n_2}@\pi_2)] \end{array}}{(\Sigma, {x_1}^{n_1}@\pi_1 = \ell_2) \longrightarrow^n_\Delta (\Sigma', \mathsf{skip})}$$

4.3.4 Store Consistency

In addition to typing for statements and expressions, Core Ironclad contains a series of judgments that ensure the well-formedness and consistency of the class context, store typing, and store itself. The relation $\mathsf{wf}\,(\Delta, \Psi, \Sigma, n)$ summarizes these judgments and says that the class context, store typing, and store are all consistent with each other up to stack height n and that no bindings exist above that stack height.

The store consistency judgment is of particular interest because it expresses precisely the key invariants of the Ironclad system. Store consistency boils down to the consistency of the individual bindings of the store. In particular, two of these binding consistency rules for pointers capture these invariants.

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

$$\frac{\begin{array}{l} \Sigma(x^n@\pi) = \mathsf{ptr}\,({x'}^{n'}@\pi') \\ \Psi({x'}^{n'}@\pi') = \tau \\ n' = 0 \end{array}}{\Psi; \Sigma \vdash x^n@\pi : \mathsf{ptr}\langle\tau\rangle\ \mathsf{ok}}$$

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.

$$\frac{\begin{array}{l} \Sigma(x^n@\,) = \mathsf{lptr}\,({x'}^{n'}@\pi') \\ \Psi({x'}^{n'}@\pi') = \tau \\ n' \leq n \end{array}}{\Psi; \Sigma \vdash x^n@\ : \mathsf{lptr}\langle\tau\rangle\ \mathsf{ok}}$$

This final property is precisely the critical invariant of the system which now has a precise definition within the context of Core Ironclad:

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

4.4 Metatheory

Proving Core Ironclad type safe establishes that this invariant holds. This result follows from standard Progress and Preservation theorems.

Theorem 1 (Progress)**.**
1. If $\mathsf{wf}\,(\Delta, \Psi, \Sigma, n)$ *and* $\Psi \vdash^n_\Delta s\ \mathsf{ok}$ *then* s *is* skip*,* error*, or* $(\Sigma, s) \longrightarrow^n_\Delta (\Sigma', s')$*.*
2. If $\mathsf{wf}\,(\Delta, \Psi, \Sigma, n)$ *and* $\Psi \vdash^n_\Delta e : \tau$ *then* e *is* ℓ*,* error*, or* $(\Sigma, e) \longrightarrow^n_\Delta (\Sigma', e')$*.*

The Progress theorem says that if a statement or expression is well-typed in a consistent context, then the term is done evaluating or it can take a step

To state the Preservation theorem, it is necessary to define a store typing subset relation that holds only at stack frames at or below n:

Definition. $\Psi \subseteq_n \Psi'$ *means that for every binding of the form* $[{x'}^{n'}@\pi : \tau] \in \Psi$ *such that* $n' \leq n$*, then* $[{x'}^{n'}@\pi : \tau] \in \Psi'$*.*

Theorem 2 (Preservation)**.**
1. If $\mathsf{wf}\,(\Delta, \Psi, \Sigma, n)$*,* $\Psi \vdash^n_\Delta s\ \mathsf{ok}$*, and* $(\Sigma, s) \longrightarrow^n_\Delta (\Sigma', s')$*, then there exists* Ψ' *such that* $\mathsf{wf}\,(\Delta, \Psi', \Sigma', n)$*,* $\Psi' \vdash^n_\Delta s'\ \mathsf{ok}$*, and* $\Psi \subseteq_n \Psi'$*.*
2. If $\mathsf{wf}\,(\Delta, \Psi, \Sigma, n)$*,* $\Psi \vdash^n_\Delta e : \tau$*, and* $(\Sigma, s) \longrightarrow^n_\Delta (\Sigma', s')$*, then there exists* Ψ' *such that* $\mathsf{wf}\,(\Delta, \Psi', \Sigma', n)$*,* $\Psi' \vdash^n_\Delta e : \tau$*, and* $\Psi \subseteq_n \Psi'$*.*

The Preservation theorem says that if a statement or expression is well-typed in a consistent context and then steps, the resulting statement or expression is well-typed in an extension of the original context that is also consistent.

Both proofs follow from straightforward mutual inductions on the typing and evaluation derivations, respectively. A number of technical lemmas are necessary to reason about how the store grows and shrinks over the course of evaluation.

The proofs of the Progress and Preservation theorems implie that the lifetime invariant holds within Core Ironclad.

Corollary (Pointer lifetime safety)**.** *The pointer lifetime invariant is preserved throughout the program's execution.*

Proof. Progress shows that a well-formed program is finished or it takes a step. Preservation shows that the well-formedness summary relation is preserved by evaluation. And this relation contains the store consistency judgment which enforces the pointer lifetime invariant. Thus, the invariant is preserved as well. □

5. Implementation

Ironclad C++ provides three components that assist the programmer in enforcing type safety in C++. The Ironclad C++ library provides safe constructs to replace unsafe C++ features. The code validator restricts code to the Ironclad C++ safe subset. Finally, the refactoring tool provides assistance for transforming unsafe C++ code to Ironclad C++.

5.1 Library

The Ironclad C++ library provides templated pointer classes, helper functions for these smart pointers, and additional functions for interfacing with unsafe code. Beyond what was discussed in Section 2 and Section 3, the library implementation has a few additional aspects. For example, to safely handle command-line arguments to `main()`, the Ironclad C++ library provides a function to convert `argv` (which has the type `char**`) to a `aptr<aptr<char>>`. Using C++11 variadic templates, the Ironclad C++ library provides a type-safe version of `printf`.

As noted in Section 2, all pointer casts in Ironclad C++ use `dynamic_cast`. Ironclad C++ provides a `cast` method that accepts smart pointers and forwards the underlying pointer to `dynamic_cast`. Using `cast` on a pointer of type `ptr<T>` also requires that the type `T` has a virtual table, meaning that `T` must contain a virtual method. If `T` does not contain a virtual method, a compiler error will occur, and the programmer must add a virtual method to allow checked casts on pointers to type `T`. Pointer upcasts, which do not require an explicit cast, are handled implicitly in the copy constructor of the `ptr<T>` class and function in the same manner as raw C++ pointers.

The Ironclad C++ library provides a variable length array, `varray<T>`, for handling variable length records. This idiom is commonly used to allow a variable length array to (1) be stored contiguously with the structure, (2) avoid an additional load for a heap allocated array, and (3) to save space (compared to using a static-sized array of the maximum size necessary). To use this construct, the `varray<T>` is placed at the end of a record of type `S`. To allocate a record of type `S`, the programmer must use the method `new_variable_ptr<S,T>(...)`. When called, the variable size allocation function allocates the variable amount of memory and uses C++'s placement new construct to initialize the elements of the array. The code determines the location of the `size` field of the `varray` using the `offsetof` macro from the C standard library. As `varray` is a *standard-layout class* as defined by the C++ standard [17] and the `size` field is located at the beginning of `varray`, the location returned by offsetof provides the correct offset to the `varray`'s size field. Thus, the size of the `varray` can be correctly initialized to enforce the proper array bounds checking for variable length structures.

```
// File:  main.cpp
void heap_call(){
  ptr<Foo> heap_foo = new_ptr<Foo>();
  heap_foo->f();
}
// Leaks this pointer to global scope
void stack_call(){
  Foo stack_foo;
  stack_foo.f();
}
```

```
// File:  foo.hpp
class Foo{
public:
  void f();
};
```

```
// File:  foo.cpp
void Foo::f(){
  setLast(this);
}
```

```
// File:  cache.hpp
class Foo;
ptr<Foo> last;
static void setLast(lptr<Foo> foo){
  last = foo;
}
```

Figure 6. Example in which local pointers can identify that a function leaks the address of a stack allocation to a global scope. In *main.cpp*, the definition of `f()` is not available, so a whole-program analysis would be required to otherwise catch this error.

To implement `lptr<T>` and `laptr<T>`, the lower-bound field of the local pointer is set to the value of the current stack pointer. As we are unaware of a standards-compliant method to obtain information about the current stack, the implementation uses non-portable x86 inline assembly within the constructor to obtain the value of the stack pointer register. For this to work correctly, the `lptr` constructor must also not be inlined, which we enforce using the `noinline` function attribute, which is honored by common compilers (including gcc, LLVM/clang, and icc). In addition, to reduce the memory overhead of local pointers, the `lptr` implementation encodes whether it currently holds the address of a heap or global value in the bottom bit of the lower-bound field, which will always otherwise be zero due to word alignment of stack frames.

5.2 Validator Tool

The Ironclad C++ validator enforces conformance to the

Ironclad C++ safe subset. The validator builds upon LLVM's clang compiler front-end, which handles source code parsing and abstract syntax tree (AST) generation. The AST provided by the clang front-end includes type information and template instantiations. Given this AST, the Ironclad C++ validator applies the simple rules described in previous sections to declarations and expressions. For example, in Figure 6, the validator needs to only locally validate that the function receiving the `this` pointer accepts an `lptr<T>` for that argument, rather than needing to perform a comprehensive cross-file escape analysis. The current implementation of the validator is a stand-alone tool, but we ultimately envision a validator integrated with the C++ compiler and invoked by setting a command line flag that checks for conformance during compilation.

5.3 Refactoring Tool

To assist in migrating code from unsafe C++ to Ironclad C++, we implemented a refactoring tool to handle the mundane code transformations, such as rewriting the type `T*` to `ptr<T>`. This refactoring tool is also built upon clang/LLVM compiler front-end. In general, the refactoring tool is intended to be used only once to initially transform code to Ironclad C++.

To determine whether to replace an unsafe pointer with a singleton (`ptr`) or array smart pointer (`aptr`), the Ironclad C++ refactoring tool implements an analysis similar to the whole program pointer type inference in CCured [25]. Unlike the analysis used for CCured, Ironclad C++ distinguishes only between singleton (SAFE) and array (SEQ) pointers because it does not contain an analog for the WILD pointer type in CCured.

The refactoring tool analyzes all pointers in the program by identifying all pointers declarations that are used as an array and propagating the array pointer type to all pointers that are assigned to an array pointer. A pointer is identified as an array pointer when it is used in a pointer arithmetic expression, used with the array indexing operator, or assigned to the result of a `new []` expression. Once the refactoring tool has identified all pointers that are directly used as an array, any pointers that are assigned to an array pointer must be updated. The refactoring tool stores the interactions between pointers in an update set and iterates over the set of all pointers until a fixed point is reached or all pointers are identified as array pointers.

5.4 Modifications to 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. For example, the `vector` container provides an overloaded indexing operator that does not ensure that the requested index is within the size of the `vector`. Instead of requiring that the STL implementations also be translated to Ironclad C++, we instead modified parts of the STL to provide safety by default. For example, we inserted a runtime check for all indexing operations on `vector` and `string` objects. Ideally, a system using Ironclad C++ would include a completely safe version of the STL, but for our benchmarks, which make limited use of the STL, modifying a small number of containers was sufficient. Ironclad C++ also requires that STL containers use the `gc_allocator` provided by the Boehm-Demers-Weiser (BDW) conservative garbage collector, which can be specified as a template parameter. This allocator ensures that memory allocated by the container is not deallocated prematurely and is tracked by the garbage collector.

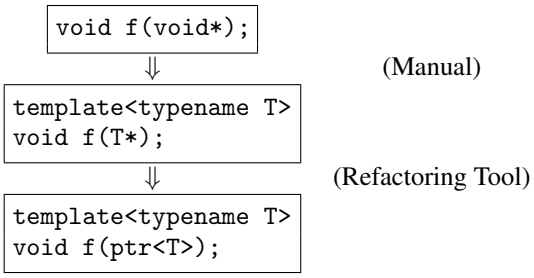

Figure 7. Common refactoring to remove `void *` function arguments. First we manually add templates. Then the refactoring tool adds `ptr`s.

6. Refactoring to Ironclad C++

To evaluate the usability and applicability of Ironclad C++ in existing C/C++ code, we refactored multiple performance benchmarks and an open-source application to Ironclad C++. Overall, our test cases amount to 50K lines of C/C++ code. We characterize the relevant C++ language features used by these applications in Table 2. Most of the applications translated to Ironclad C++ in a straightforward manner. With the refactoring tool, few manual modifications were required illustrating the expressiveness of Ironclad C++. In the following subsections, we describe a few of the corner cases that we were forced to handle primarily due to the use of C style type unsafe code.

6.1 Case Study: leveldb

To evaluate the applicability of Ironclad C++ to a modern C++ code base, we translated an open-source key-value storage library, leveldb. The leveldb application, developed at Google, has 16,188 lines of C++ code (excluding comments and whitespace). We chose to refactor leveldb because it is designed for performance and implements its own data structures, such as a skip list and an LRU cache. Refactoring leveldb from unsafe C++ to Ironclad C++ was largely straightforward, due to its use of modern C++ features. Leveldb contains 66 classes, 21 of which inherit from at least one base class. Roughly half of all variable and parameter declarations in leveldb are pointers, and a small number of `void` pointers were present in the original code. Before refactoring, it was necessary to manually modify 106 lines of code to remove `void` pointers.

Benchmark	Language	Class	POD	Inherit	Casts	Pointers	References	LOC	CLOC	%CLOC
blackscholes	C	1	1	N/A	2	20%	N/A	405	51	13%
bzip2	C	2	2	N/A	221	47%	N/A	5731	653	11%
lbm	C	1	1	N/A	4	57%	N/A	904	104	12%
sjeng	C	2	2	N/A	7	46%	N/A	10544	691	7%
astar	C++	25	14	0	83	7%	35%	4280	156	4%
canneal	C++	3	0	0	4	27%	29%	2817	151	5%
fluidanimate	C++	4	2	0	12	7%	7%	2785	195	7%
leveldb	C++	66	8	21	202	49%	24%	16188	1933	12%
namd	C++	15	3	4	7	46%	7%	3886	398	10%
streamcluster	C++	5	2	2	27	37%	0%	1767	142	11%
swaptions	C++	1	1	0	16	39%	0%	1095	122	11%

Table 2. This table lists the characteristics of the evaluated applications. From left to right, the columns detail the benchmark name, most prevalant source language, number of classes (including structs), number of plain-old-data classes, number of classes that inherit from one or more bases, number of casts, pointer declarations as a percentage of total declarations, reference declarations as a percentage of total declarations, lines of code, changed lines of code, and percentage of changed lines of code.

To enable our refactoring tool to work well with leveldb, we first manually replaced all `void` pointers through function templating, as shown in Figure 7. Although, using function templates was amenable in most cases, there was one exception with leveldb. Leveldb has a virtual method `schedule` in its `Environment` base class that takes a function pointer and `void` pointer as arguments and uses the `void` pointer as an argument to the function pointer. We could not template this method because C++ disallows templated virtual methods. To address this problem, we used inheritance, leveraging dynamic dispatch, instead of compile-time templating. We replace the function pointer and `void` pointer arguments to schedule with a pointer to a `ScheduledRunner` class that includes a pure virtual `run` method. We then replaced each call to `schedule` with a `Runner` class that inherits from the `ScheduledRunner` base class. We observed in this case, and others, that `void` pointers were often used to interact with threading libraries, such as `pthreads`. With the new common threading interface provided by C++11 (that does not require `void` pointers), the need for `void` pointers will further decrease in C++ code, thus allowing code to better conform to Ironclad C++.

Apart from removing `void` pointers, we had to make two additional changes. The `Slice` class in leveldb contains a constructor that take a `const char *`. Refactoring this constructor to Ironclad C++ makes the parameter `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 one implicit user-defined conversion allowed by the C++ standard [17]. Thus, we were forced to explicitly convert string literals to `aptr<const char>` in these cases as shown in Figure 8. These modifications are included in the 12% of total lines of code modified. Second, we also modified the variable length structures used by leveldb in two separate classes to use the safe `varray<T>` class described in Section 5.1.

While evaluating the performance of refactored leveldb, a few benchmarks exhibited unexpectedly large memory overheads. When we viewed a dump of the state of the garbage collector, we noticed that the application had frequently allocated memory blocks of size 4,112 bytes. The conservative collector allocates blocks in multiples of size 4,096 bytes, so for every allocation of 4,112 bytes, 4,080 bytes were wasted. The root of these allocations was the custom allocator used by leveldb that was designed to allocate memory blocks of size 4,096 bytes. When translated to Ironclad C++, the custom allocator requested 16 extra bytes of memory that were required for pointer meta-data. We were able to greatly reduce the memory overhead for these benchmarks by fixing the allocation size to accommodate the additional 16 bytes of meta-data.

6.2 SPEC and PARSEC Benchmarks

We refactored both C and C++ benchmarks from the SPEC benchmark suite to Ironclad C++. Most C++ benchmarks translated to Ironclad C++ without requiring manual modifications. For the C benchmarks, we initially needed to modify the code just to compile the code with a C++ compiler due to some notable language differences between the two languages [33]. The most frequent modification required for C++ compliance was the transition from `malloc` to the type-safe `new` and `new []` operators. Rather than attempting to automatically convert arguments to `malloc` to arguments to `new` and `new []`, we manually transformed the C benchmark code to use the C++ allocation methods. Aside from the modifications to translate from C to C++, additional refactorings were required for some benchmarks. Most of these corner cases stemmed from a lack of type safety in small sections of the overall code-base. The PARSEC benchmarks, which are written in modern C++, are more type-safe than the SPEC benchmarks, and they were easily refactored to Ironclad C++ in a straightforward manner. We give examples of the additional refactorings for four benchmarks.

```
class Slice{
  Slice(const char * arg){ ... }
};

void append(Slice & s);

void f()
  append(``StrLiteral'');
}
```

```
class Slice{
  Slice(aptr<const char> arg){ ... }
};

void append(Slice & s);

void f()
  append(aptr<const char>(``StrLiteral''));
}
```

Figure 8. An explicit conversion in leveldb due to the C++ restriction of one user-defined implicit conversion.

Astar. This benchmark implicitly converts the `this` pointer of the `rvector` class, which has three floating point fields, to an array of type `float *` using an unchecked cast. Further, the `rvector` class provides an unsafe function operator that performs pointer arithmetic on the `this` pointer. As neither method is type-safe, we (1) rewrote the function operator to return the `x`, `y`, or `z` field of the `rvector` class based on an index input (and fail if the index was not in range), (2) removed the unsafe implicit conversion, and (3) replaced occurrences of `rvector[index]` with `rvector(index)`.

Bzip2. The original code for bzip2 contains a large number of `void` pointers. To use only type-safe pointers, we removed `void` pointers by identifying the type `T` to which each void pointer was cast and replaced `void *` with `T *`. In most cases, this refactoring was straightforward. In one case, an unchecked cast was used to convert a `void *` struct field to either an `EState *` or `DState *`. The code required that `EState` and `DState` were related because the same struct field was used to hold both. To allow this, we created an inheritance hierarchy with a base class `State`. To allow checked casts from the base class `State` to `EState` and `DState`, we augmented the `State` class with an empty virtual method, which allows our library to use `dynamic_cast` on pointers of type `State`, `EState`, and `DState`. We observed no measurable runtime overheads due to the inheritance hierarchy and checked casts.

Lbm. This benchmark uses unsafe casts to store flags in the bottom bits of a floating point number. We specialize a case of `cast` to allow the same size conversion from `double *` to `long *`. We were also required to modify the type of the matrix used for most of the computations in lbm. In the original version, an unsafe cast from a `double***` to a `double**` was required for compilation, but this was only necessary because the type of the variable was incorrect. Thus, we modified the code to type-check without an unsafe cast.

Namd. This application along with many others, uses functions for allocating arrays on a given address alignment. These methods require unsafe casts that Ironclad C++ does not allow. Therefore, the Ironclad C++ library provides an `allocate_aligned_array` method that takes a size and an alignment. This method provides a replacement for custom aligned allocation routines.

6.3 Bug Detection Effectiveness

We also tested the effectiveness of the safety checks inserted by Ironclad C++. We used multiple suites of known bugs, including selected programs from BugBench [22], 30 test cases from the NIST Juliet Suite [26], and the Wilander test suite [36]. We chose NIST test cases from the various array-out-of-bounds vulnerability categories by selecting a single flow-variant for each data-type and attack. From the BugBench suite, we evaluated *gzip*, *man*, *ncompress*, and *polymorph*. On the test cases from the BugBench suite and NIST Juliet Suite, the Ironclad C++ versions safely aborted on the buggy inputs and executed correctly otherwise. The Ironclad C++ version of the Wilander test suite also safely aborted on all tests.

7. Evaluation of Runtime Overhead

Although memory safety is a desirable property, it would ideally come at the cost of only a small overhead during execution. To evaluate the performance implications, we executed the refactored programs described in the previous section. Our test system is an Intel Core2 2.66 Ghz processor. The benchmarks were compiled using gcc version 4.5.1. To reduce overhead, the library forces the compiler to inline all dereference and indexing operators using gcc's `always_inline` function attribute. Garbage collection in Ironclad C++ is provided by the the Boehm-Demers-Weiser conservative garbage collector [4].

7.1 Runtime

Figure 9 shows the increase in runtime with safety checking of code that passes the Ironclad C++ code validator. With all safety checks enabled and garbage collection, the average performance overhead is 20%. Checks for null pointers are responsible for a 4% runtime increase. Adding array bounds checking increases the overall performance overhead to 19%. The addition of dynamic lifetime checking has a negligible impact on performance in most cases, because assignments to local pointers are reasonably rare in these benchmarks. The performance impacts of garbage collection is small on average. The garbage collector uses a different memory allocation algorithm, which results in different memory alignment for some benchmarks, which appears to be largely responsible for the small performance improvements observed in some benchmarks.

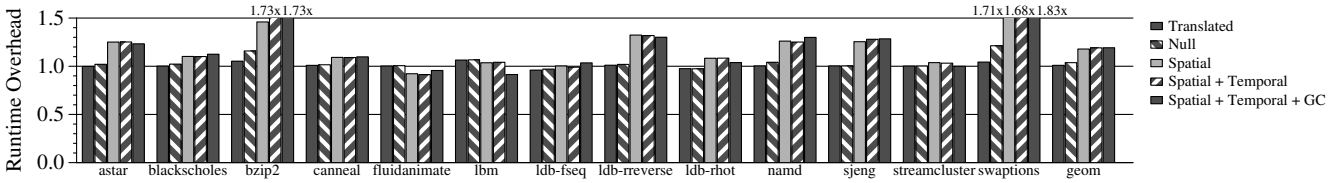

Figure 9. Runtime overheads for Ironclad C++ pointers under various levels of safety checking

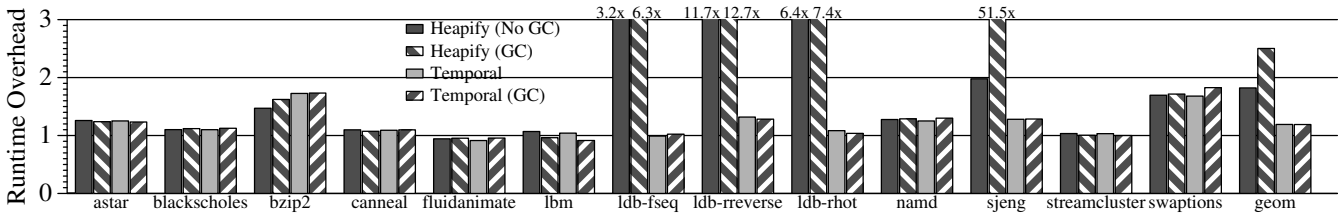

Figure 10. Runtime overhead comparison of dynamic lifetime checking versus heapification

7.2 Comparison to Heapification

Many languages, such as Go and C#, choose to heapify stack values when the address may escape. To examine the trade-off between dynamic lifetime checking and heapification, we compared the runtime overheads of the two approaches, shown in Figure 10. The amount of heapification and temporal checking necessary for each benchmark varies depending on how many stack allocations are used in the original code. For example, blackscholes allocates five large arrays on the heap but rarely uses the stack. Thus, the difference in its runtime when comparing heapification and dynamic lifetime checking is negligible. However, we do observe a few cases in which dynamic lifetime checking provides a significant runtime advantage compared to heapification.

In sjeng, multiple buffers are allocated on the stack in the search routine. These buffers are subsequently passed to other functions, thus requiring that Ironclad C++ ensures that references to these buffers do not escape to the heap. With dynamic lifetime checking, we observe a small increase in runtime overhead compared to the overhead for bounds checking in sjeng. On the other hand, if we are forced to heapify the buffers, we see an overhead of 2x without garbage collection enabled and an overhead of 51x with garbage collection.

We also observe significant runtime overheads under heapification for the leveldb benchmarks. The leveldb code base uses stack objects more frequently than the rest of our benchmarks, and therefore required that we move more objects from the stack to the heap in the heapified version. In this case, we see large runtime overheads even without garbage collection enabled. With garbage collection and heapification, leveldb exhibits overheads of 5.4x, 12.8x, and 6.5x for the fillseq, readreverse, and readhot benchmarks. On these microbenchmarks, dynamic lifetime checking compares favorably to heapification at overheads of 1.04x, 1.30x, and 1.04x. When objects are frequently declared on the stack and their addresses may escape, we find that dynamic lifetime checking provides a performance advantage over heapification.

7.3 Memory Overhead

We also measured the memory overheads for each benchmark. These overheads were recorded using the *pages-as-heap* option of valgrind's massif tool. For the additional metadata required for Ironclad C++ smart pointers, we observe a memory overhead of 25%. With garbage collection enabled, the average overhead increases to 124%. Of the benchmarks that exhibit over 100% overhead under garbage collection, the maximum original memory usage was 59 MB. For smaller initial heap sizes, the data structures used for garbage collection more significantly affect the memory overhead. Both the *fillseq* benchmark from *leveldb* and *astar* from SPEC use a custom block allocator. Such custom allocators are known to cause the conservative garbage collector to be unable to collect unused memory due to pointers that may exist in a blocks of memory that have been returned to the custom allocator's pool.

7.4 Executable Size and Compile Time

To measure the impact on executable size of replacing raw pointers with templated classes, we used the linux size utility to determine the text segment component. As shown in Table 3, we see an increase in executable size of 57.9%. As the increase in size is measured in terms of kilobytes, this magnitude of code-size increase should only affect the most constrained environments. We also measured an 87% increase in compile time due to the addition of templates,

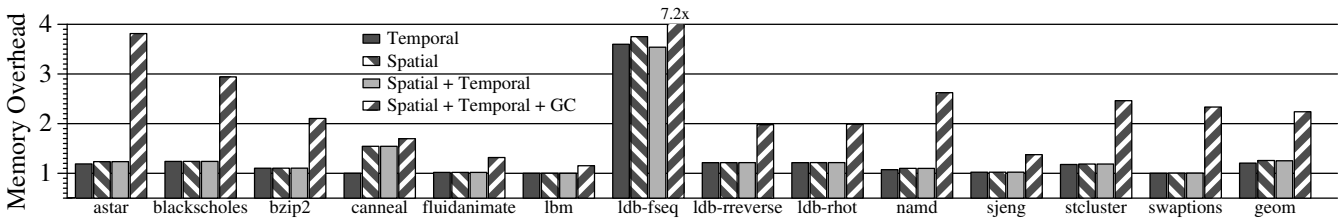

Figure 11. Memory Usage of Ironclad C++ under various levels of safety

Benchmark	Original Size (KB)	Ironclad Size (KB)	% Overhead
astar	43.2	71.3	65.3
blackscholes	7.1	10.9	53.8
bzip2	64.3	130.1	102.3
canneal	35.1	42.3	20.5
fluidanimate	19.7	24.9	26.7
lbm	14.4	32.9	128.8
leveldb	297.9	402.3	35.0
namd	324.5	640.7	97.5
sjeng	144.0	216.2	50.1
streamcluster	19.5	30.3	55.4
swaptions	17.2	34.8	101.9
Average	-	-	57.9

Table 3. Executable size overheads for each benchmark

but this increase was usually measured as a few seconds or, in some cases, less than a second. On the applications with a longer base compile time, such as leveldb, the percentage increase was lower than the average.

7.5 Optimizations

The reported benchmark results include tuned versions of two of the benchmarks.

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 `reduce2` function that simply bounds checks the start and end indices of the reduction on both input arrays, and then it runs the computation at unchecked speeds. With this optimization, the streamcluster benchmark executes with no measurable overhead compared to the original.

swaptions In our initial Ironclad C++ version of swaptions, we noticed an increase in runtime due to garbage collector overhead. We found that a specific method was repeatedly allocating and deallocating buffers of the same size, causing the collector to run frequently. Changing the function to reuse the allocated buffer whenever possible reduced the runtime overhead from 1.99x to 1.83x.

8. Related Work

8.1 Programming Languages

Many programming languages provide some level of type safety, array bounds checking, and prevention of use-after-free errors. We compare language attributes of some related type-safe languages in Table 4.

By default, all objects in Java are allocated on the garbage collected heap. Thus, Java prevents use-after-free errors. However, forcing all allocations to exist on the heap can have substantial performance implications.

The D programming language is syntactically similar to C and C++ and provides bounds checked pointers and garbage collection, though both of these measures can be circumvented. A variant of D, SafeD, restricts the set of language features that can be used in an attempt to provide memory safety [6]. In SafeD, all pointers must be bounds checked, memory must be managed using garbage collection, and pointer casts, including casts from one pointer type to another, are restricted. SafeD does not allow objects to be stack allocated.

In both C# and Go, the compiler may decide to heapify a value that may escape the stack, based on an escape analysis [12]. Both C# and Go also employ garbage collection to manage heap allocations.

Rust provides shared, unique, and unsafe pointers [29]. If the Rust compiler cannot prove that a referred-to value will outlive the reference, it will make a copy of the value for the program to refer to when possible. Copying a unique pointer creates a deep copy of the pointed to allocation. Rust does not provide a stack allocated array type.

BitC ensures memory safety through array bounds checking, garbage collection, and restrictions on the use of references [30]. BitC does not allow array references to escape, and by-reference parameters may escape only through first-class procedures.

Cyclone [18] provides non-null (`@`) and fat (`?`) pointers on top of standard pointers. For heap-allocated memory, it resorts to a garbage collector. However, Cyclone also provides a region-based type system [13] that makes all dereferences of dangling pointers to region-managed memory a type-checking error. The prevention of use-after-free for

Type-Safe Language	Memory Layout	Stack Allocated Objects	Stack Allocated Arrays	Transparent Allocation	Industrial Strength Tools	Garbage Collected	Object Oriented	TIOBE Index
Java	No	No	No	Yes	Yes	Yes	Yes	2
C#	Yes	Yes	Yes	No	Yes	Yes	Yes	5
Go	Yes	Yes	Yes	No	Yes	Yes	Yes*	51+
Rust	No	Yes	No	No	No	Yes	No	N/A
BitC	Yes	Yes	Yes	Yes	No	Yes	Yes*	N/A
SafeD	Yes	No	No	Yes	Yes	Yes	Yes	N/A (36)
Ironclad C++	Yes	Yes	Yes	Yes	Yes	Yes	Yes	N/A (3)

Table 4. Comparison of Ironclad C++ to other type-safe languages based on if programmers can control memory layout, if stack allocation is possible for objects and arrays, if the programmer fully controls where objects are allocated, if industrial strength compilers and tools are widely available, if the language uses garbage collection, if the language is object oriented, and TIOBE index ranking (or closest relative). *BitC and Go do not have inheritance.

stack allocations in Ironclad C++ can be seen as a compromise of this purely static region-based system where it trades dynamic checks for a simpler type system.

ALGOL 68 prevents use-after-free errors by checking the scope level of references on assignment. However, in practice, most implementations of ALGOL 68 implement only static checks rather than the specified dynamic checks due to efficiency concerns [21].

The Ada 95 compiler inserts runtime accessibility checks to ensure that a reference will not outlive the object it refers to [34]. In practice, the Ada 95 style guide suggests that programmers allocate dynamic data in the earliest scope possible, which indicates that the accessibility checks used are too restrictive for heap allocations [32]. Ada 95 also does not guarantee that memory will be deallocated, whether using a garbage collector or not, so the style guide encourages programmers to use "Unchecked Delete" to ensure that memory is deallocated. Using unchecked delete may lead to erroneous execution as defined by the Ada 95 reference.

8.2 Retrofitting Memory Safety

Many prior efforts have attempted to retrofit C for memory safety. SafeC replaces C pointers with fat pointers to prevent out-of-bounds and use-after-free errors in C [2]. However, SafeC suffers from large runtime overheads and was mainly designed for debugging C code rather than enforcing memory safety at all times. CCured implements a source-to-source translation pass that augments pointer types with SAFE, SEQ, and WILD identifiers [25]. CCured requires non-trivial transformations to avoid the use of WILD pointers, which are inefficient, and the source-to-source transformation used by CCured must be run before each compilation.

Dhurjati, Kowshik, Adve, and Lattner use a technique to automatically infer memory regions coupled with statically-checked restrictions on user code to ensure static memory safety for C programs [9]. For heap memory, their system uses their Automatic Pool Allocation algorithm to insert type-homogenous memory pools into the program. For stack memory, they disallow a stack address from being stored in a heap or global object or returned from a function. These restrictions are similar to what Ironclad C++ enforces. However, where Ironclad C++ enforces these restrictions by a combination of local pointers and dynamic checks, their system uses interprocedural data analyses to verify the same properties at compile time.

The Boost libraries provide scoped and reference-counted smart pointers that enable automatic memory management for the pointers they contain [5]. However, the reference-counted pointers provided do not handle sub-object references. If a pointer is created to the field of an object managed by a Boost smart pointer, the reference count will not account for the sub-object pointer, and the object may be prematurely deallocated. Boost's smart pointers do not perform bounds checking.

9. Conclusion

Ironclad C++, a safe subset of C++, provides type safety for C++ through code validation and runtime checks. We have shown that Ironclad C++ adds a runtime overhead of 20% over a set of C and C++ benchmarks. Further, we have shown that Ironclad C++ requires reasonable code modifications, especially for clean, modern C++ code bases such as leveldb. In our study of Ironclad C++, we investigated multiple alternatives for preventing use-after-free errors for stack allocations and found that dynamic lifetime checking offered flexibility, memory control, and limited source code modifications as compared to heapification. Ironclad C++ provides programmers with an effective and pragmatic mechanism for ensuring memory safety while maintaining many of the advantages of C++ and a standard development environment.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[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] 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.

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

[5] 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.

[6] *Memory-Safe-D-Spec*. D Programming Language, 2012. http://dlang.org/memory-safe-d.html.

[7] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2008.

[8] 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.

[9] 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.

[10] D. Gay and A. Aiken. Memory Management with Explicit Regions. In *Proceedings of the SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 313–323, June 1998.

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

[12] *FAQ*. The Go Programming Language, 2012. http://golang.org/doc/go_faq.html.

[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] *C++ - Programming languages*. The Hammer Principle, 2012. hammerprinciple.com/therighttool/c-2/.

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

[16] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: a Minimal Core Calculus for Java and GJ. In *Proceedings of the 14th SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Application (OOPSLA)*, Oct. 1999.

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

[18] 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.

[19] 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.

[20] R. W. M. Jones and P. H. J. Kelly. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In *Third International Workshop on Automated Debugging*, Nov. 1997.

[21] D. Lomet. Making Pointers Safe in System Programming Languages. *Software Engineering, IEEE Transactions on*, SE-11(1):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] J. Pincus and B. Baker. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns. *IEEE Security & Privacy*, 2(4):20–27, 2004.

[28] T. Ramananandro, G. D. Reis, and X. Leroy. Formal Verification of Object Layout for C++ Multiple Inheritance. In *Proceedings of The 38th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*, Jan. 2011.

[29] *Rust Reference Manual*. Rust, 2012. http://www.bitc-lang.org/docs/bitc/spec-0.10.html.

[30] J. S. Shapiro and S. Sridhar. *BitC 0.10 Language Specification*. The EROS Group, 2008. http://www.bitc-lang.org/docs/bitc/spec-0.10.html.

[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] *Ada 95 Quality and Style Guide*. Software Productivity Consortium, 2005. www.adaic.org/resources/add_content/docs/95style/html/cover.html.

[33] B. Stroustrup. Sibling Rivalry, C and C++. Technical report, AT&T Labs – Research, 2002.

[34] S. Taft and R. Duff. *Ada 95 Reference Manual: Language and Standard Libraries : International Standard Iso/Iec 8652:1995(E)*. Lecture notes in computer science. Springer, 1997.

[35] D. Wasserrab, T. Nipkow, G. Snelting, and F. Tip. An Operational Semantics and Type Safety Proof for Multiple Inheritance in C++. In *Proceedings of the 21st SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA)*, Oct. 2006.

[36] 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.

[37] 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.