C/C++ Thread Safety Analysis

DeLesley Hutchins
Google Inc.
Email: delesley@google.com

Aaron Ballman
CERT/SEI
Email: aballman@cert.org

Dean Sutherland
Email: dfsuther@cs.cmu.edu

***Abstract*—Writing multithreaded programs is hard. Static analysis tools can help developers by allowing threading policies to be formally specified and mechanically checked. They essentially provide a static type system for threads, and can detect potential race conditions and deadlocks.**

This paper describes Clang Thread Safety Analysis, a tool which uses annotations to declare and enforce thread safety policies in C and C++ programs. Clang is a production-quality C++ compiler which is available on most platforms, and the analysis can be enabled for any build with a simple warning flag: `-Wthread-safety`.

The analysis is deployed on a large scale at Google, where it has provided sufficient value in practice to drive widespread voluntary adoption. Contrary to popular belief, the need for annotations has not been a liability, and even confers some benefits with respect to software evolution and maintenance.

I. Introduction

Writing multithreaded programs is hard, because developers must consider the potential interactions between concurrently executing threads. Experience has shown that developers need help using concurrency correctly [1]. Many frameworks and libraries impose thread-related policies on their clients, but they often lack explicit documentation of those policies. Where such policies are clearly documented, that documentation frequently takes the form of explanatory prose rather than a checkable specification.

Static analysis tools can help developers by allowing threading policies to be formally specified and mechanically checked. Examples of threading policies are: "the mutex `mu` should always be locked before reading or writing variable `accountBalance`" and "the `draw()` method should only be invoked from the GUI thread."

Formal specification of policies provides two main benefits. First, the compiler can issue warnings on policy violations. Finding potential bugs at compile time is much less expensive in terms of engineer time than debugging failed unit tests, or worse, having subtle threading bugs hit production.

Second, specifications serve as a form of machine-checked documentation. Such documentation is especially important for software libraries and APIs, because engineers need to know the threading policy to correctly use them. Although documentation can be put in comments, our experience shows that comments quickly "rot" because they are not updated when variables are renamed or code is refactored.

This paper describes thread safety analysis for Clang. The analysis was originally implemented in GCC [2], but the GCC version is no longer being maintained. Clang is a production-quality C++ compiler, which is available on most platforms, including MacOS, Linux, and Windows. The analysis is currently implemented as a compiler warning. It has been deployed on a large scale at Google; all C++ code at Google is now compiled with thread safety analysis enabled by default.

II. Overview of the Analysis

Thread safety analysis works very much like a type system for multithreaded programs. It is based on theoretical work on race-free type systems [3]. In addition to declaring the type of data (`int`, `float`, etc.), the programmer may optionally declare how access to that data is controlled in a multithreaded environment.

Clang thread safety analysis uses annotations to declare threading policies. The annotations can be written using either GNU-style attributes (*e.g.*, `__attribute__ ((...))`) or C++11-style attributes (*e.g.*, `[[...]]`). For portability, the attributes are typically hidden behind macros that are disabled when not compiling with Clang. Examples in this paper assume the use of macros; actual attribute names, along with a complete list of all attributes, can be found in the Clang documentation [4].

Figure 1 demonstrates the basic concepts behind the analysis, using the classic bank account example. The `GUARDED_BY` attribute declares that a thread must lock `mu` before it can read or write to `balance`, thus ensuring that the increment and decrement operations are atomic. Similarly, `REQUIRES` declares that the calling thread must lock `mu` before calling `withdrawImpl`. Because the caller is assumed to have locked `mu`, it is safe to modify `balance` within the body of the method.

In the example, the `depositImpl()` method lacks a `REQUIRES` clause, so the analysis issues a warning. Thread safety analysis is not interprocedural, so caller requirements must be explicitly declared. There is also a warning in `transferFrom()`, because it fails to lock `b.mu` even though it correctly locks `this->mu`. The analysis understands that these are two separate mutexes in two different objects. Finally, there is a warning in the `withdraw()` method, because it fails to unlock `mu`. Every lock must have a corresponding unlock; the analysis detects both double locks and double unlocks. A function may acquire a lock without releasing it (or vice versa), but it must be annotated to specify this behavior.

A. Running the Analysis

To run the analysis, first download and install Clang [5]. Then, compile with the `-Wthread-safety` flag:

```
clang -c -Wthread-safety example.cpp
```

```
#include "mutex.h"

class BankAcct {
  Mutex mu;
  int   balance GUARDED_BY(mu);

  void depositImpl(int amount) {
    // WARNING! Must lock mu.
    balance += amount;
  }

  void withdrawImpl(int amount) REQUIRES(mu) {
    // OK. Caller must have locked mu.
    balance -= amount;
  }

public:
  void withdraw(int amount) {
    mu.lock();
    // OK.  We've locked mu.
    withdrawImpl(amount);
    // WARNING!  Failed to unlock mu.
  }

  void transferFrom(BankAcct& b, int amount) {
    mu.lock();
    // WARNING! Must lock b.mu.
    b.withdrawImpl(amount);
    // OK.  depositImpl() has no requirements.
    depositImpl(amount);
    mu.unlock();
  }
};
```

Fig. 1. Thread Safety Annotations

Note that this example assumes the presence of a suitably annotated mutex.h [4] that declares which methods perform locking and unlocking.

B. Thread Roles

Thread safety analysis was originally designed to enforce locking policies such as the one previously described, but locks are not the only way to ensure safety. Another common pattern in many systems is to assign different *roles* to different threads, such as "worker thread" or "GUI thread" [6].

The same concepts used for mutexes and locking can also be used for thread roles, as shown in Figure 2. Here, a widget library has two threads, one to handle user input, like mouse clicks, and one to handle rendering. It also enforces a constraint: the draw() method should only be invoked only by the GUI thread. The analysis will warn if draw() is invoked directly from onClick().

The rest of this paper will focus discussion on mutexes in the interest of brevity, but there are analogous examples for thread roles.

III. Basic Concepts

Clang thread safety analysis is based on a calculus of capabilities [7] [8]. To read or write to a particular location in memory, a thread must have the *capability*, or permission, to do so. A capability can be thought of as an unforgeable key,

```
#include "ThreadRole.h"

ThreadRole Input_Thread;
ThreadRole GUI_Thread;

class Widget {
public:
  virtual void onClick() REQUIRES(Input_Thread);
  virtual void draw()    REQUIRES(GUI_Thread);
};

class Button : public Widget {
public:
  void onClick() override {
    depressed = true;
    draw();    // WARNING!
  }
};
```

Fig. 2. Thread Roles

or token, which the thread must present to perform the read or write.

Capabilities can be either *unique* or *shared*. A unique capability cannot be copied, so only one thread can hold the capability at any one time. A shared capability may have multiple copies that are shared among multiple threads. Uniqueness is enforced by a linear type system [9].

The analysis enforces a single-writer/multiple-reader discipline. Writing to a guarded location requires a unique capability, and reading from a guarded location requires either a unique or a shared capability. In other words, many threads can read from a location at the same time because they can share the capability, but only one thread can write to it. Moreover, a thread cannot write to a memory location at the same time that another thread is reading from it, because a capability cannot be both shared and unique at the same time.

This discipline ensures that programs are free of data races, where a *data race* is defined as a situation that occurs when multiple threads attempt to access the same location in memory at the same time, and at least one of the accesses is a write [10]. Because write operations require a unique capability, no other thread can access the memory location at that time.

A. Background: Uniqueness and Linear Logic

Linear logic is a formal theory for reasoning about resources; it can be used to express logical statements like: "You cannot have your cake and eat it too" [9]. A unique, or *linear*, variable must be used exactly once; it cannot be duplicated (used multiple times) or forgotten (not used).

A unique object is *produced* at one point in the program, and then later *consumed*. Functions that use the object without consuming it must be written using a hand-off protocol. The caller hands the object to the function, thus relinquishing control of it; the function hands the object back to the caller when it returns.

For example, if std::stringstream were a linear type, stream programs would be written as follows:

```
std::stringstream ss;          // produce ss
auto& ss2 = ss << "Hello";     // consume ss
auto& ss3 = ss2 << "World.";   // consume ss2
return ss3.str();              // consume ss3
```

Notice that each stream variable is used exactly once. A linear type system is unaware that ss and ss2 refer to the same stream; the calls to << conceptually consume one stream and produce another with a different name. Attempting to use ss a second time would be flagged as a use-after-consume error. Failure to call ss3.str() before returning would also be an error because ss3 would then be unused.

B. Naming of Capabilities

Passing unique capabilities explicitly, following the pattern described previously, would be needlessly tedious, because every read or write operation would introduce a new name. Instead, Clang thread safety analysis tracks capabilities as unnamed objects that are passed implicitly. The resulting type system is formally equivalent to linear logic but is easier to use in practical programming.

Each capability is associated with a named C++ object, which identifies the capability and provides operations to produce and consume it. The C++ object itself is not unique. For example, if mu is a mutex, mu.lock() produces a unique, but unnamed, capability of type Cap<mu> (a dependent type). Similarly, mu.unlock() consumes an implicit parameter of type Cap<mu>. Operations that read or write to data that is guarded by mu follow a hand-off protocol: they consume an implicit parameter of type Cap<mu> and produce an implicit result of type Cap<mu>.

C. Erasure Semantics

Because capabilities are implicit and are used only for type-checking purposes, they have no run time effect. As a result, capabilities can be fully erased from an annotated program, yielding an unannoted program with identical behavior.

In Clang, this erasure property is expressed in two ways. First, recommended practice is to hide the annotations behind macros, where they can be literally erased by redefining the macros to be empty. However, literal erasure is unnecessary. The analysis is entirely static and is implemented as a compile time warning; it cannot affect Clang code generation in any way.

IV. Thread Safety Annotations

This section provides a brief overview of the main annotations that are supported by the analysis. The full list can be found in the Clang documentation [4].

GUARDED_BY(...) and PT_GUARDED_BY(...)

GUARDED_BY is an attribute on a data member; it declares that the data is protected by the given capability. Read operations on the data require at least a shared capability; write operations require a unique capability.

PT_GUARDED_BY is similar but is intended for use on pointers and smart pointers. There is no constraint on the data member itself; rather, the data it points to is protected by the given capability.

```
Mutex mu;
int *p2  PT_GUARDED_BY(mu);

void test() {
  *p2 = 42;       // Warning!
  p2 = new int;   // OK  (no GUARDED_BY).
}
```

REQUIRES(...) and REQUIRES_SHARED(...)

REQUIRES is an attribute on functions; it declares that the calling thread must have unique possession of the given capability. More than one capability may be specified, and a function may have multiple REQUIRES attributes. REQUIRES_SHARED is similar, but the specified capabilities may be either shared or unique.

Formally, the REQUIRES clause states that a function takes the given capability as an implicit argument and hands it back to the caller when it returns, as an implicit result. Thus, the caller must hold the capability on entry to the function and will still hold it on exit.

```
Mutex mu;
int a GUARDED_BY(mu);

void foo() REQUIRES(mu) {
  a = 0;  // OK.
}

void test() {
  foo();  // Warning!  Requires mu.
}
```

ACQUIRE(...) and RELEASE(...)

The ACQUIRE attribute annotates a function that produces a unique capability (or capabilities), for example, by acquiring it from some other thread. The caller must not-hold the given capability on entry, and will hold the capability on exit.

RELEASE annotates a function that consumes a unique capability, (*e.g.*, by handing it off to some other thread). The caller must hold the given capability on entry, and will not-hold it on exit.

ACQUIRE_SHARED and RELEASE_SHARED are similar, but produce and consume shared capabilities.

Formally, the ACQUIRE clause states that the function produces and returns a unique capability as an implicit result; RELEASE states that the function takes the capability as an implicit argument and consumes it.

Attempts to acquire a capability that is already held or to release a capability that is not held are diagnosed with a compile time warning.

CAPABILITY(...)

The CAPABILITY attribute is placed on a struct, class or a typedef; it specifies that objects of that type can be used to identify a capability. For example, the threading libraries at Google define the Mutex class as follows:

```
class CAPABILITY("mutex") Mutex {
public:
  void lock()           ACQUIRE(this);
  void readerLock()     ACQUIRE_SHARED(this);
  void unlock()         RELEASE(this);
  void readerUnlock()   RELEASE_SHARED(this);
};
```

Mutexes are ordinary C++ objects. However, each mutex object has a capability associated with it; the lock() and unlock() methods acquire and release that capability.

Note that Clang thread safety analysis makes no attempt to verify the correctness of the underlying Mutex implementation. Rather, the annotations allow the interface of Mutex to be expressed in terms of capabilities. We assume that the underlying code implements that interface correctly, *e.g.*, by ensuring that only one thread can hold the mutex at any one time.

TRY_ACQUIRE(b, ...) and TRY_ACQUIRE_SHARED(b, ...)

These are attributes on a function or method that attempts to acquire the given capability and returns a boolean value indicating success or failure. The argument b must be true or false, to specify which return value indicates success.

NO_THREAD_SAFETY_ANALYSIS

NO_THREAD_SAFETY_ANALYSIS is an attribute on functions that turns off thread safety checking for the annotated function. It provides a means for programmers to opt out of the analysis for functions that either (a) are deliberately thread-unsafe, or (b) are thread-safe, but too complicated for the analysis to understand.

Negative Requirements

All of the previously described requirements discussed are positive requirements, where a function requires that certain capabilities be held on entry. However, the analysis can also track negative requirements, where a function requires that a capability be *not*-held on entry.

Positive requirements are used to prevent race conditions. Negative requirements are used to prevent deadlock. Many mutex implementations are not reentrant, because making them reentrant entails a significant performance cost. Attempting to acquire a non-reentrant mutex that is already held will deadlock the program.

To avoid deadlock, acquiring a capability requires a proof that the capability is not currently held. The analysis represents this proof as a negative capability, which is expressed using the ! negation operator:

```
Mutex mu;
int a GUARDED_BY(mu);

void clear() REQUIRES(!mu) {
  mu.lock();
  a = 0;
  mu.unlock();
}

void reset() {
  mu.lock();
  // Warning!  Caller cannot hold 'mu'.
  clear();
  mu.unlock();
}
```

Negative capabilities are tracked in much the same way as positive capabilities, but there is a bit of extra subtlety.

Positive requirements are typically confined within the class or the module in which they are declared. For example, if a thread-safe class declares a private mutex, and does all locking and unlocking of that mutex internally, then there is no reason clients of the class need to know that the mutex exists.

Negative requirements lack this property. If a class declares a private mutex mu, and locks mu internally, then clients should theoretically have to provide proof that they have *not* locked mu before calling any methods of the class. Moreover, there is no way for a client function to prove that it does *not* hold mu, except by adding REQUIRES(!mu) to the function definition. As a result, negative requirements tend to propagate throughout the code base, which breaks encapsulation.

To avoid such propagation, the analysis restricts the visibility of negative capabilities. The analyzer assumes that it holds a negative capability for any object that is not defined within the current lexical scope. The scope of a class member is assumed to be its enclosing class, while the scope of a global variable is the translation unit in which it is defined.

Unfortunately, this visibility-based assumption is unsound. For example, a class with a private mutex may lock the mutex, and then call some external routine, which calls a method in the original class that attempts to lock the mutex a second time. The analysis will generate a false negative in this case.

Based on our experience in deploying thread safety analysis at Google, we believe this to be a minor problem. It is relatively easy to avoid this situation by following good software design principles and maintaining proper separation of concerns. Moreover, when compiled in debug mode, the Google mutex implementation does a run time check to see if the mutex is already held, so this particular error can be caught by unit tests at run time.

V. Implementation

The Clang C++ compiler provides a sophisticated infrastructure for implementing warnings and static analysis. Clang initially parses a C++ input file to an abstract syntax tree (AST), which is an accurate representation of the original source code, down to the location of parentheses. In contrast, many compilers, including GCC, lower to an intermediate language during parsing. The accuracy of the AST makes it easier to emit quality diagnostics, but complicates the analysis in other respects.

The Clang semantic analyzer (Sema) decorates the AST with semantic information. Name lookup, function overloading, operator overloading, template instantiation, and type checking are all performed by Sema when constructing the AST. Clang inserts special AST nodes for implicit C++ operations, such as automatic casts, LValue-to-RValue conversions,

implicit destructor calls, and so on, so the AST provides an accurate model of C++ program semantics.

Finally, the Clang analysis infrastructure constructs a control flow graph (CFG) for each function in the AST. This is not a lowering step; each statement in the CFG points back to the AST node that created it. The CFG is shared infrastructure; the thread safety analysis is only one of its many clients.

A. Analysis Algorithm

The thread safety analysis algorithm is flow-sensitive, but not path-sensitive. It starts by performing a topological sort of the CFG, and identifying back edges. It then walks the CFG in topological order, and computes the set of capabilities that are known to be held, or known not to be held, at every program point.

When the analyzer encounters a call to a function that is annotated with ACQUIRE, it adds a capability to the set; when it encounters a call to a function that is annotated with RELEASE, it removes it from the set. Similarly, it looks for REQUIRES attributes on function calls, and GUARDED_BY on loads or stores to variables. It checks that the appropriate capability is in the current set, and issues a warning if it is not.

When the analyzer encounters a join point in the CFG, it checks to confirm that every predecessor basic block has *the same* set of capabilities on exit. Back edges are handled similarly: a loop must have the same set of capabilities on entry to and exit from the loop.

Because the analysis is not path-sensitive, it cannot handle control-flow situations in which a mutex might or might not be held, depending on which branch was taken. For example:

```
void foo() {
  if (b) mutex.lock();
  // Warning: b may or may not be held here.
  doSomething();
  if (b) mutex.unlock();
}

void lockAll() {
  // Warning: capability sets do not match
  // at start and end of loop.
  for (unsigned i=0; i < n; ++i)
    mutexArray[i].lock();
}
```

Although this seems like a serious limitation, we have found that conditionally held locks are relatively unimportant in practical programming. Reading or writing to a guarded location in memory requires that the mutex be held unconditionally, so attempting to track locks that *might* be held has little benefit in practice, and usually indicates overly complex or poor-quality code.

Requiring that capability sets be the same at join points also speeds up the algorithm considerably. The analyzer need not iterate to a fixpoint; thus it traverses every statement in the program exactly once. Consequently, the computational complexity of the analysis is $O(n)$ with respect to code size. The compile time overhead of the warning is minimal.

B. Intermediate Representation

Each capability is associated with a C++ object. C++ objects are run time entities, that are identified by C++ expressions. The same object may be identified by different expressions in different scopes. For example:

```
class Foo {
  Mutex mu;
  bool compare(const Foo& other)
      REQUIRES(this->mu, other.mu);
}

void bar() {
  Foo a;
  Foo *b;
  ...
  a.mu.lock();
  b->mu.lock();
  // REQUIRES (&a)->mu, (*b).mu
  a.compare(*b);
  ...
}
```

Clang thread safety analysis is dependently typed: note that the REQUIRES clause depends on both this and other, which are parameters to compare. The analyzer performs variable substitution to obtain the appropriate expressions within bar(); it substitutes &a for this and *b for other.

Recall, however, that the Clang AST does not lower C++ expressions to an intermediate language; rather, it stores them in a format that accurately represents the original source code. Consequently, (&a)->mu and a.mu are different expressions.

A dependent type system must be able to compare expressions for semantic (not syntactic) equality. The analyzer implements a simple compiler intermediate representation (IR), and lowers Clang expressions to the IR for comparison. It also converts the Clang CFG into single static assignment (SSA) form so that the analyzer will not be confused by local variables that take on different values in different places.

C. Limitations

Clang thread safety analysis has a number of limitations. The three major ones are:

No attributes on types. Thread safety attributes are attached to declarations rather than types. For example, it is not possible to write vector<int GUARDED_BY(mu)>, or (int GUARDED_BY(mu))[10]. If attributes could be attached to types, PT_GUARDED_BY would be unnecessary.

Attaching attributes to types would result in a better and more accurate analysis. However, it was deemed infeasible for C++ because it would require invasive changes to the C++ type system that could potentially affect core C++ semantics in subtle ways, such as template instantiation and function overloading.

No dependent type parameters. Race-free type systems as described in the literature often allow classes to be parameterized by the objects that are responsible for controlling access. [11] [3] For example, assume a Graph class has a list of nodes, and a single mutex that protects all of them. In this case, the

Node class should technically be parameterized by the graph *object* that guards it (similar to inner classes in Java), but that relationship cannot be easily expressed with attributes.

No alias analysis. C++ programs typically make heavy use of pointer aliasing; we currently lack an alias analysis. This can occasionally cause false positives, such as when a program locks a mutex using one alias, but the GUARDED_BY attribute refers to the same mutex using a different alias.

VI. Experimental Results and Conclusion

Clang thread safety analysis is currently deployed on a wide scale at Google. The analysis is turned on by default, across the company, for every C++ build. Over 20,000 C++ files are currently annotated, with more than 140,000 annotations, and those numbers are increasing every day. The annotated code spans a wide range of projects, including many of Google's core services. Use of the annotations at Google is entirely voluntary, so the high level of adoption suggests that engineering teams at Google have found the annotations to be useful.

Because race conditions are insidious, Google uses both static analysis and dynamic analysis tools such as Thread Sanitizer [12]. We have found that these tools complement each other. Dynamic analysis operates without annotations and thus can be applied more widely. However, dynamic analysis can only detect race conditions in the subset of program executions that occur in test code. As a result, effective dynamic analysis requires good test coverage, and cannot report potential bugs until test time. Static analysis is less flexible, but covers all possible program executions; it also reports errors earlier, at compile time.

Although the need for handwritten annotations may appear to be a disadvantage, we have found that the annotations confer significant benefits with respect to software evolution and maintenance. Thread safety annotations are widely used in Google's core libraries and APIs. Annotating libraries has proven to be particularly important, because the annotations serve as a form of machine-checked documentation. The developers of a library and the clients of that library are usually different engineering teams. As a result, the client teams often do not fully understand the locking protocol employed by the library. Other documentation is usually out of date or non-existent, so it is easy to make mistakes. By using annotations, the locking protocol becomes part of the published API, and the compiler will warn about incorrect usage.

Annotations have also proven useful for enforcing internal design constraints as software evolves over time. For example, the initial design of a thread-safe class must establish certain constraints: locks are used in a particular way to protect private data. Over time, however, that class will be read and modified by many different engineers. Not only may the initial constraints be forgotten, they may change when code is refactored. When examining change logs, we found several cases in which an engineer added a new method to a class, forgot to acquire the appropriate locks, and consequently had to debug the resulting race condition by hand. When the constraints are explicitly specified with annotations, the compiler can prevent such bugs by mechanically checking new code for consistency with existing annotations.

The use of annotations does entail costs beyond the effort required to write the annotations. In particular, we have found that about 50% of the warnings produced by the analysis are caused not by incorrect code but rather by incorrect or missing annotations, such as failure to put a REQUIRES attribute on getter and setter methods. Thread safety annotations are roughly analogous to the C++ const qualifier in this regard.

Whether such warnings are false positives depends on your point of view. Google's philosophy is that incorrect annotations are "bugs in the documentation." Because APIs are read many times by many engineers, it is important that the public interfaces be accurately documented.

Excluding cases in which the annotations were clearly wrong, the false positive rate is otherwise quite low: less than 5%. Most false positives are caused by either (a) pointer aliasing, (b) conditionally acquired mutexes, or (c) initialization code that does not need to acquire a mutex.

Conclusion

Type systems for thread safety have previously been implemented for other languages, most notably Java [3] [11]. Clang thread safety analysis brings the benefit of such systems to C++. The analysis has been implemented in a production C++ compiler, tested in a production environment, and adopted internally by one of the world's largest software companies.

References

[1] K. Asanovic *et al.*, "A view of the parallel computing landscape," *Communications of the ACM*, vol. 52, no. 10, 2009.

[2] L.-C. Wu, "C/C++ thread safety annotations," 2008. [Online]. Available: https://docs.google.com/a/google.com/document/d/1_d9MvYX3LpjTk3nlubM5LE4dFmU91SDabVdWp9-VDxc

[3] M. Abadi, C. Flanagan, and S. N. Freund, "Types for safe locking: Static race detection for java," *ACM Transactions on Programming Languages and Systems*, vol. 28, no. 2, 2006.

[4] "Clang thread safety analysis documentation." [Online]. Available: http://clang.llvm.org/docs/ThreadSafetyAnalysis.html

[5] "Clang: A c-language front-end for llvm." [Online]. Available: http://clang.llvm.org

[6] D. F. Sutherland and W. L. Scherlis, "Composable thread coloring," *PPoPP '10: Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, 2010.

[7] K. Crary, D. Walker, and G. Morrisett, "Typed memory management in a calculus of capabilities," *Proceedings of POPL*, 1999.

[8] J. Boyland, J. Noble, and W. Retert, "Capabilities for sharing," *Proceedings of ECOOP*, 2001.

[9] J.-Y. Girard, "Linear logic," *Theoretical computer science*, vol. 50, no. 1, pp. 1–101, 1987.

[10] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs." *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 4, 1997.

[11] C. Boyapati and M. Rinard, "A parameterized type system for race-free Java programs," *Proceedings of OOPSLA*, 2001.

[12] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: data race detection in practice," *Workshop on Binary Instrumentation and Applications*, 2009.