# SAFECode Software Architecture Manual

John Criswell

July 25, 2019

# 1 Introduction

Welcome to the SAFECode Software Architecture Manual! The purpose of this manual is to give the reader an overview of how the SAFECode transform passes are organized, the reason for this organization, and an introduction to the source code layout.

# 2 SAFECode Compiler Structure

## 2.1 SAFECode Compiler Design Principles

SAFECode is implemented as a set of LLVM analysis and transform passes. One important question in designing SAFECode was how to split up the work of transforms between different LLVM passes.

The following principles guide the design of SAFECode's software architecture. They are based on standard good practice in compiler design as well as observations made while maintaining and improving SAFECode:

1. **Separation of Concerns:**

   SAFECode passes should be as simple as possible. Previous versions of SAFECode had passes that performed both run-time check instrumentation and optimization of the run-time checks. This created a situation in which code was complex and passes had a dozen different options to turn features on and off.

   By separating concerns, each pass is smaller, easier to read, and easier to understand. It also makes the software more flexible; features can be enabled and disabled by simply choosing to run or not run a particular transform pass. This is useful for measuring the impact of optimizations as well as allowing tools like `bugpoint` to better isolate bugs.

2. **Enable Integration into LLVM and Clang:**

   At some point in the future, we (or others) may want to integrate parts of SAFECode into LLVM and/or Clang. Doing so would have many benefits, including wide-scale adoption, better integration with the compiler toolchain, and additional developers.

   To make integration into other LLVM projects easier, SAFECode attempts to adhere to the next principle.

3. **Make Whole-Program Analysis an Optimization:**

   A simple approach to implementing SAFECode is to first run whole-program analysis passes to infer properties about the program and then to have transforms use this information to instrument the code with run-time checks when necessary. The problem is that LLVM performs whole-program analysis in the linker; the linker, in turn, runs LLVM transform passes more or less unconditionally.

   Therefore, we want SAFECode instrumentation passes to require no whole-program analysis at all and write the more sophisticated features into optimizations on run-time checks. The front-end (e.g., Clang) can then decide whether to instrument a program and run a simple transform pass. The linker can then check to see if the program contains any run-time checks and, if so, improve or optimize those checks using whole-program analysis techniques.

## 2.2 Compilation Phases

SAFECode's various transform passes can be, roughly speaking, grouped into several phases as follows:

1. **Check Insertion Phase:**

   In this phase, SAFECode examines the code for operations which may cause a memory safety error and inserts run-time checks as needed. These run-time checks are simple and do not assume that everything about the program is known. They are designed so that they can be used by a front-end (like Clang) to instrument programs.

2. **Check Optimization Phase:**

   During this phase, SAFECode attempts to optimize the run-time checks it inserted in the Check Insertion Phase. Some of these optimizations do not require whole program analysis and could be integrated into a front-end compiler; others do require whole-program analysis and would normally be implemented in an optimizing linker.

   A key feature of these optimization passes is that they work on both instrumented and uninstrumented code. If there are no run-time checks to optimize, they should do nothing.

An important optimization that is executed during this phase is Automatic Pool Allocation. Automatic Pool Allocation will change all heap allocations to allocate memory out of distinct pools, and it will also modify run-time checks to include pool handles; the run-time checks can use these pool handles to speed up their checks or to make their checks more strict.

3. **Check Completion Phase:**

   The Check Completion Phase uses whole-program analysis to modify the run-time checks in a program with completeness information. Completeness means that everything that can be known about a memory object is known to the compiler, and therefore the run-time check can be more strict about what it considers to be correct behavior.

4. **Debug Instrumentation Phase:**

   Finally, there's a phase for instrumenting the run-time checks with debug information if the user wants to use SAFECode more as a debugger than as a production-use memory safety system.

# 3 Source Code Layout

The SAFECode analysis and transformation sources are organized as follows:

1. `lib/ArrayBoundChecks`: This library contains several analysis passes for static array bounds checking.

2. `lib/InsertPoolChecks`: This library contains the transform passes for inserting run-time checks and for inserting code to register memory objects within individual pools. It also contains the CompleteChecks pass which implements the Check Completion Phase.

3. `lib/OptimizeChecks`: This library contains several passes for optimizing run-time checks.

4. `lib/RewriteOOB`: This library contains passes for implementing Ruwase/Lam pointer rewriting. This code allows SAFECode to tolerate out-of-bounds pointers that are never dereferenced.

5. `lib/DebugInstrumentation`: This library implements code that modifies run-time checks to contain additional debug information (if such debug information is present in the program). It is used in SAFECode's debug tool mode.

6. `lib/DanglingPointers`: This library contains a pass that modifies a program to perform dangling pointer detection.

SAFECode also contains a few run-time libraries that are linked into programs:

1. `runtime/BitmapPoolAllocator`: This run-time library implements a pool memory allocator for SAFECode.
2. `runtime/DebugRuntime`: This run-time library implements the run-time checks used by SAFECode. Despite the name, it currently contains implementations of both the production and debug mode run-time checks.

# 4 Run-time Checks and Instrumentation

## 4.1 Complete vs. Incomplete Checks

Most of the SAFECode run-time checks come in two flavors: complete and incomplete. A complete check attempts to find the bounds of a memory object into which a pointer points; if it cannot find the memory object, then the pointer must be invalid, and the check fails.

The problem with complete checks is that they only work when the compiler knows everything that can be known about a memory object. Sadly, this isn't always true; applications are linked with native code libraries compiled with other compilers (unthinkable, I know, but it happens); the SAFECode compiler cannot analyze native code, and so it does not know about memory objects allocated or freed by the library code (also known as *external code*).

Incomplete checks are SAFECode's way of permitting a mixture of SAFECode-compiled code and native external code; if a pointer could be manipulated by external code, SAFECode relaxes its run-time checks so that failure to find the referent memory object does not cause a run-time check to fail.

By default, SAFECode makes all of its checks incomplete checks (this is because each compilation unit treats other compilation units as external code). When used with libLTO, SAFECode can use points-to analysis to determine which incomplete checks can be converted into complete checks without causing false positives.

Incomplete checks admit the possibility that memory safety errors will escape detection. However, they make memory safety usable in practice, and so we use them.

## 4.2 Run-time Checks

Below are the run-time checks that SAFECode may add to a program. Note that many of these functions have alternate versions for pointers that are determined to be incomplete or unknown by SAFECode's points-to analysis algorithm.

- `poolcheck (void * pool, void * ptr, size_t length)`:
  The `poolcheck` call is used to instrument loads and stores to memory (including LLVM atomic operations). It ensures that the pointer points within a memory object in the pool and that the load or store will not read/write past the end of the memory object.

- `fastlscheck (void * ptr, void * start, size_t objsz, size_t len)`:
  The `fastlscheck()` function is identical to the `poolcheck()` function in functionality; the difference is that `fastlscheck()` is passed the bounds of the memory object into which the pointer should point. It is an optimized version of `poolcheck()` that does not need to search for object bounds information in a side data structure.

- `poolcheck_align (void * pool, void * ptr)`:
  The `poolcheck_align()` function is used when type-safe load/store optimizations are enabled. It is possible for a pointer which is type-safe to be loaded from a memory object which is not type-safe. When a type-safe pointer is loaded via a type-inconsistent pointer, `poolcheck_align()` verifies that the loaded pointer points within the specified pool at the correctly aligned offset for objects of its type. This ensures that no further checks are needed when the type-safe pointer is used for loads and stores.

- `free_check (void * pool, void * ptr)`:
  The `free_check()` function checks that the pointer points to the beginning of a valid heap object. It is used to catch invalid `free` calls for allocators not known to tolerate invalid deallocation requests.

- `boundscheck (void * pool, void * src, void * dest)`:
  The `boundscheck()` function takes a source pointer and a destination pointer that is computed from the source pointer; the checks first determines whether the source pointer is within a valid memory object within the specified pool and, if so, that the destination pointer is within the same memory object. It is primarily used for performing array and structure indexing checks on LLVM `getelementptr` instructions.

  If the destination pointer goes out of bounds, then `boundscheck()` returns a *rewrite pointer*. A rewrite pointer (or *OOB pointer*) point to an unmapped portion of the address space. They are used to allow pointers to go out of bounds so long as they are not dereferenced.

- `exactcheck (void * src, void * dest, void * base, int objsize)`:
  The `exactcheck()` function is a fast version of the `boundscheck()` function that does not need to do an object bounds lookup.

- `funccheck (void * ptr, void * targets[])`:
  The `funccheck()` function determines if a function pointer belongs to the set of valid function pointer targets for an indirect function call. It is used to ensure control-flow integrity.

SAFECode also instruments code with other functions to support the above run-time checks:

- `getActualValue()`: The `getActualValue()` function takes a value and determines if it is a rewrite pointer. If it is, it returns the actual out-of-bounds value that the rewrite pointer represents. Otherwise, it returns the original value.

  The `getActualValue()` function is primarily used for supporting the comparison of pointers that have gone outside their object bounds.

- `pool_register()`: The `pool_register()` family of functions register the bounds of allocated memory objects in side data-structures; these are used to map a pointer to the memory object to which it belongs in run-time checks.

  Note that some memory objects may not be registered if SAFECode determines that their bounds are never needed.

- `pool_reregister()`: The `pool_reregister()` function unregisters a memory object and registers a new object of the specified size. It is designed to support allocators like `realloc()`.