Common Run-time Check Proposal

John Criswell

May 24, 2012

1 Introduction

There is growing interest in the LLVM community for creating programs with memory-safety run-time checks. Debugging tools such as ASan and SAFECode instrument programs with memory safety checks to locate programmer errors. The Clang compiler inserts light-weight checks into code to find simple errors with little overhead. Safe language implementations insert run-time checks into code to enforce array bounds checking as required by their language specifications. Security researchers use LLVM to develop faster versions of memory safety checks in the hopes of one day making such checks usable in production systems.

At their core, all of these systems do the same thing: they insert run-time checks to verify a) that getelementptr (GEP) instructions do not create buffer oveflows and/or b) that instructions accessing memory access valid memory objects.

Since there are optimizations from which all of these tools would benefit, it would make sense to create a common set of run-time checks for LLVM with a common set of optimization passes to optimize the placement of the checks. Individual tools can then specialize the checks as necessary.

2 Run-time Checks and Instrumentation

Below are the set of common run-time checks.

- `loadcheck (void * ptr, int length)`:
  The `loadcheck` call is used to instrument loads from memory. It ensures that the pointer points within a valid memory object and that the load will not read past the end of the memory object.
- `storecheck (void * ptr, int length)`:
  The `storecheck` call is used to instrument stores to memory (including

LLVM atomic operations). It ensures that the pointer points within a valid memory object and that the store will not write past the end of the memory object.

- `fastloadcheck (void * ptr, void * start, int objsize, int len)`:
  The `fastloadcheck()` function is identical to the `loadcheck()` function in functionality; the difference is that `fastloadcheck()` is passed the bounds of the memory object into which the pointer should point. It is an optimized version of `loadcheck()`.

- `faststorecheck (void * ptr, void * start, int objsize, int len)`:
  The `faststorecheck()` function is identical to the `storecheck()` function in functionality; the difference is that `faststorecheck()` is passed the bounds of the memory object into which the pointer should point. It is an optimized version of `storecheck()`.

- `gepcheck (void * src, void * dest)`:
  The `gepcheck()` function takes the source pointer operand and result of a GEP instruction; the check first determines whether the source pointer is within a valid memory object 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.

- `fastgepcheck (void * src, void * dest, void * base, int objsize)`:
  The `fastgepcheck()` function is a fast version of the `gepcheck()` function that is given the bounds of the memory object into which the GEP source and result pointers should point.

- `funccheck (void * f())`:
  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 enforce control-flow integrity.

- `free_check (void * p)`:
  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.

In addition to run-time checks, there are support functions that aid in implementing the aforementiond checks:

- `pool_register_heap(void * p, int size)`
  `pool_register_stack(void * p, int size)`
  `pool_register_global(void * p, int size)`:
  The `pool_register()` family of functions inform the run-time of new memory object allocations. For some systems, these will be no-ops. For others, the bounds information of the memory object will be registered in a side data structure.

- `pool_unregister_heap(void * p)`
  `pool_unregister_stack(void * p)`:
  The `pool_unregister()` family of functions informs the run-time system of a memory deallocation. For some systems, it may be a no-op. For others, it may remove the bounds information of a memory object from a side data structure or overwrite the contents of the memory object with a special value.
- `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()`.

3 Examples Applications

This section describes how the checks could be used in various tools.

3.1 Clang Bounds Checking

The Clang compiler can emit bounds checks when the `-fbounds-checking` option is used. These checks are not meant to be thorough; they merely provide a small amount of sanity checking and security to code. They should be fast and not cause false positives. As a result, the checks are only added to pointers that are within the same function as the allocation sites from which the pointers were derived.

The Clang compiler currently emits inline code using the `objectsize` intrinsic to perform these checks. However, it could instead emit `gepcheck` calls to perform those same checks. A generic optimization that converts calls of `gepcheck` to `fastgepcheck` would identify those checks that can be performed quickly. A Clang-specific pass could then remove those checks on pointers that do not refer to locally allocated memory objects (i.e., the checks that were not converted into `fastgepcheck`).

This would enable Clang's run-time checks to be optimized using the same optimizations that are being developed for ASan and SAFECode. These optimizations can remove checks on trivially safe GEPs, hoist checks out of monotonic loops, and prove GEPs safe statically using static array bounds checking.

3.2 Control-Flow Integrity

Control-flow integrity [1] (abbreviated CFI) is a security property that ensures that branches within a program only target valid control-flow destinations. It must ensure that indirect function calls only jump to the first address of valid function targets and that returns from functions only branch back to instructions following potential calls to the called function.

The generic instrumentation passes could be used to instrumented and optimize code for CFI. First, the pass that adds `storecheck()` calls would instrument stores to ensure that they do not overwrite stack frames; a second pass would add calls to `funccheck()` to ensure that indirect function calls do not jump to invalid targets. Standard optimizations would be used to optimize the run-time checks.

After optimization, the compiler would use a CFI-specific transform to modify calls to `storecheck()` to mask bits off of pointers before store instructions. The code generator would be specialized to generate code with two stacks: one for control-flow information and one for stack object allocations. The control-flow stack would be placed in memory such that the upper bits of pointers would need to be set to access it. Since all potentially unsafe stores clear those bits before dereference, the control-flow stack is never modified by errant program stores.

3.3 ASan

The Address Sanitizer (ASan) is a memory safety error debugging tool that has been integrated into LLVM. It is primarily designed to detect invalid loads and stores and uses various unsound run-time techniques to detect and report array bounds violations and dangling pointer uses. ASan does this by using a side data structure (called the shadow memory) to record whether a memory address belongs to a valid memory object. Checks on loads and stores examine the shadow memory to see if a memory address belongs to a valid memory object.

Using the proposed common infrastructure, ASan would use a shared LLVM instrumentation pass to instrument all load, store, and atomic intrinsic operations with `loadcheck` and `storecheck` calls. Additionally, it would use another generic instrumentation pass to insert `pool_register` and `pool_unregister` calls.

Common optimization passes would optimize these run-time checks. Such optimizations might hoist checks out of loops, convert `loadcheck` and `storecheck` checks to `fastloadcheck` and `faststorecheck` checks, or remove checks that will always pass at run-time. Optimizations to remove `pool_register` calls would also improve performance; memory objects that are always accessed in a safe manner would not need to be registered.

Note that many of these optimizations are the same ones that would optimize the Clang bounds checks. In fact, a program could be compiled with both the

ASan and Clang run-time checks, and a common set of optimizations would optimize the checks from both.

Naturally, ASan's implementation of the run-time checks would need to be specialized. ASan's implementation of `loadcheck` and `storecheck` would need to check to see if the pointer used by the load or store points into a valid memory object by consulting the shadow memory. For speed, ASan would want the implementation of `loadcheck` and `storecheck` inlined, so an ASan-specific transform pass might transform calls to `loadcheck` and `storecheck` into inline code that performs the check. ASan might also transform `pool_register` calls. The `pool_register_stack` call, for example, might be used to tell ASan where to insert guard memory around stack objects; calls to `pool_register_heap` may just be removed since ASan opts to use its own `malloc()` implementation to update shadow memory on allocation and deallocation.

3.4 SAFECode

The SAFECode compiler is an extension of Clang that instruments code with memory safety checks; its checks are designed to be sound [3] and to catch invalid loads and stores, array and structure indexing violations, indirect function pointer violations, and uninitialized pointer uses [3, 2]. SAFECode is able to employ optimizations such as automatic pool allocation and type-safe load/store check elimination to reduce performance overhead.

A unique feature of SAFECode is that it can permit out-of-bounds pointers so long as they are not used in a memory access operation. To do this, SAFECode changes out-of-bound pointers into special values called *rewrite pointers* that point to unmapped memory. If the out-of-bound pointer is ever used to access memory, either a `loadcheck/storecheck` will detect the error, or an MMU fault will detect the error.

Similar to ASan, SAFECode would instrument code using generic instrumentation passes. It would use one pass to insert `loadcheck` calls, another to insert `storecheck` calls, another to insert `gepcheck` calls, a fourth to insert `funccheck` calls, and a fifth pass to insert `free_check` calls. Like ASan and Clang, SAFECode would optimize the run-time checks using the generic run-time check optimization passes. Note that optimization passes for SAFECode would benefit both ASan and Clang since SAFECode uses a superset of the checks that ASan and Clang use.

Like ASan, SAFECode would need to specialize the run-time checks. SAFECode uses multiple side data structures to track exact object bounds meta-data, so it would need to add parameters to the `loadcheck`, `storecheck`, and `gepcheck` checks. To support the out-of-bound pointer rewriting feature, SAFECode's implementation of `gepcheck` would not terminate the program when an indexing error is detected; a `gepcheck` that passes would return the `dest` parameter while a `gepcheck` that fails would return a rewrite pointer. A special transform pass would replace the use of a pointer checked with `gepcheck` with the return value of the `gepcheck` call. In this way, if a `gepcheck` fails, the program continue to run, but if the return value is out-of-bounds, it will be detected when it is used for a load or store.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13:4:1–4:40, November 2009.

[2] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for C with very low overhead. In *International Conference on Software Engineering*, pages 162–171, Shanghai, China, May 2006.

[3] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 144–157, Ottawa, Canada, June 2006.