Proposed Schemes for Stack Temporal Safety

Ben Cole

Contents

Overview

The purpose of this document is to explain briefly the problem of stack temporal safety, and to explain the protection schemes that I plan to implement for my Master's project.

Temporal memory safety, in the context of programming languages, means ensuring that code cannot dereference any pointers to memory that has not yet been allocated, or has been deallocated and possibly reallocated for something else. Dereferencing a pointer beyond the lifetime of the object it points to is a *temporal safety violation*; doing so with a pointer to a stack-allocated address is a *stack temporal safety violation*.

In this project I aim to develop software-only and joint hardware-software mitigations using CHERI to protect against stack temporal safety violations, focusing specifically on avoiding *use-after-free* violations. A primary objective is to make such violations impossible; secondary objectives are a low runtime overhead and compatibility with as large a subset of C and C++ code as possible.

Schemes to Implement

I propose to implement and analyse four related schemes for stack temporal safety:

1. A naive adaptation of Cornucopia [1], a temporal safety scheme for heap-allocated memory. I anticipate this to be orders of magnitude too slow, because applications continually reuse stack addresses, so revocation will be required far too frequently to be viable.
2. *Shadow stack lifetime tracking (SSLT)*, a partially-new software-only scheme inspired by CETS [2] but optimised specifically for stack memory. The scheme should work for all valid C and C++ code, albeit with a noticeable performance overhead.
3. *Address-derived capability lifetimes (ADCL)*, a novel hardware-software scheme incorporating ideas both from *capability lifetimes* [3] and *monotonic capabilities* [4] that uses a new technique for specifying stack-frame sizes. These techniques all offer hardware support for theoretically faster validation of valid capability stores, but at the cost of breaking some valid C/C++ code. My proposed scheme serves as a "best of both worlds," suffering the main disadvantages of neither.
4. *ADCL with SSLT fallback*: use the ADCL scheme for performance, but fall back to the SSLT scheme in the event of a hardware-detected stack lifetime exception. This should provide fast, hardware-accelerated performance in the common case, but reserves a slower software-only scheme for providing full temporal safety. This improves on the proposed hardware-only schemes because it does not break valid C and C++ code; it will also hopefully outperform existing software-only schemes because of its common-case hardware acceleration.

Following is a description of each of these in more detail. I want to thank Lawrence Esswood for his help in fleshing out the ideas: he put together a proposal for a variant of the ADCL+SSLT scheme [5] with tips for implementation.

Scheme 1: Revocation

Intended to show that schemes for heap temporal safety are inappropriate for stack memory, I plan to adapt the existing Cornucopia revocation framework [1] to issue a revocation sweep after all function calls. Unlike heap addresses, which the userspace allocator is free to allocate in any order and has no obligation to reuse straight away (other than for cache efficiency), stack addresses tend to be reused immediately for subsequent function calls, so quarantining would not be an option. This is why the revocation pass would be needed after every function call, which I anticipate to be orders of magnitude too expensive.

As an optimisation, *escape analysis* could reduce this to only those functions with possibly-escaping pointers to local variables, but I predict this will still impose a substantial overhead for those that are detected. Escape analysis is undecidable in general, so any escape analysis algorithm necessarily will detect some false positives.

Scheme 2: Shadow Stack Lifetime Tracking (SSLT)

The CETS scheme [2] uses a "lock" and "key" value for each allocation. Pointers are augmented with a reference to a dynamically-allocated "lock" that tracks an "age" for some particular address. The pointer also includes a "key" value, which is the expected age to be found at this address. Pointer dereferences only succeed if the pointer's key matches the value stored at the lock address, and the value stored in the lock is modified (e.g. incremented) on any deallocation. This provides full temporal safety because temporal violations will be detected whenever the program attempts to dereference a temporally invalid pointer. For stack temporal safety, CETS allocates one lock per stack frame.

Cornucopia already provides heap temporal safety at a lower overhead than CETS, so I would only implement its stack-safety component. Because stack allocations are never deallocated explicitly, there would be no need to prevent double frees, so I would not need to maintain a mapping of freeable pointers like CETS. Further, an optimisation not explicitly mentioned in the CETS paper is that the overhead of performing the key and value comparison need not be suffered if the function has no escaping stack allocations, so I would use an escape analysis pass to restrict this scheme only to functions with escaping pointers to stack-allocated values. Compile-time *temporal check elimination* would additionally remove some redundant comparisons to decrease the overhead.

Rather than allocating a lock for each stack frame, which would incur dynamic allocations being made upon certain function calls, I propose instead to use a *shadow stack lifetime map*, inspired by Cornucopia's *shadow bitmap* that indicates which regions have been revoked. The shadow stack lifetime map will contain a lifetime counter for each possible stack frame start position. If we force stack frames to be no smaller than 64 bytes and aligned to at least 64 bytes in memory (these properties will be useful for Scheme 4), and if we use 8-byte lifetime counters, then the shadow stack lifetime map will increase stack memory usage by only 12.5%. Given that stacks tend to be small and non-expandable, this should be a tolerable space overhead.

To implement this, a compiler pass should insert into each function's epilogue some code that increments all indices in the shadow stack lifetime map that span the function's stack frame. This procedure is highly amenable to vectorisation. Dereferences for loads and stores should have age-checking logic inserted around them, as in the CETS paper.

Additional Considerations

- In theory, the lifetime counters will only need to be incremented after a pointer to the current stack frame has escaped. We can approximate this at compile-time by only emitting the instructions to increment the values if the function contains a possibly-escaping local variable.
- Placing loads and stores at the end of a function is inconvenient because memory operations typically have a high latency, especially when they require DRAM accesses. However, by inserting them during an intermediate LLVM pass rather than through binary instrumentation, it's possible that the LLVM backend's instruction scheduler will be able to schedule the loads and stores earlier.

Scheme 3: Address-Derived Capability Lifetimes (ADCL)

As explained previously, my scheme builds on the ideas of capability lifetimes and monotonic capabilities. Here I explain these two techniques before presenting the new scheme.

Capability Lifetimes

The *capability lifetimes* scheme [3] augments architectural capabilities to stack addresses with *lifetime counters* that track the "lifetime" of the stack allocation to which they point – essentially the number of functions that were already in the stack when the function was invoked. The key observation is that objects allocated to inner stack frames will always be outlived by objects allocated to outer stack frames, and this can be detected dynamically by comparing their respective lifetime counter values. Storing a capability to a shorter-lifetime address at a longer-lifetime address (e.g. storing a pointer to a callee's stack frame in the caller's stack frame) triggers a hardware exception as a result. The paper showed that this provides full temporal safety, but at the cost of breaking any valid C or C++ code that stores capabilities in the opposite orientation without actually dereferencing them outside of their lifetimes. The paper also ignores the possibility of storing capabilities to the stack on the heap, so I assume it must be disallowed.

The following code explains this more concretely:

```
struct object { struct object *other; }

void inner(struct object *objectFromOuter, struct object *objectOnHeap)
{
    struct object objectFromInner;
    struct object objectFromInner2;

    // ALLOWED
    objectFromInner.other = objectFromOuter;

    // ALLOWED (Circular references within a frame)
    objectFromInner.other = &objectFromInner2;
    objectFromInner2.other = &objectFromInner;

    // NOT ALLOWED (Runtime exception)
    objectFromOuter->other = &objectFromInner;

    // NOT ALLOWED (Runtime exception)
    objectOnHeap->other = objectFromInner;

    // When the function returns, the pointers objectFromOuter->other and
    // objectOnHeap->other would be temporally invalid. We prevent that from occurring
    // by forbidding the possibly-dangerous stores.
}

void outer()
{
    struct object objectFromOuter;
    struct object *objectOnHeap = new struct object;
    inner(&objectFromOuter, objectOnHeap);
    delete objectOnHeap;
}
```

The main problem with this scheme is that all capabilities (or, at least, all stack capabilities) need to be

augmented with a lifetime counter value. This adds a substantial per-capability space overhead. It may be microarchitecturally unviable as a result, and would certainly suffer worse D-cache performance.

Monotonic Capabilities

Monotonic capabilities [4] reduce the memory overhead of storing lifetime counters with each capability by *comparing the addresses themselves* to determine whether a store should be allowed. Expressed as code, the main observation is:

```
// OUTER SCOPE
int a;

// INNER SCOPE
{
    int b;
    assert(&b < &a);
}
```

Unfortunately, while this preserves full temporal safety with lower memory usage, it breaks even more valid C and C++ code: for example, under this scheme it would be impossible for two stack-allocated pointers to hold circular references to each other, such as:

```
struct object { struct object *other }

void myFunction()
{
    object a, b;
    a.other = &b;
    b.other = &a;
}
```

Even some linearisable code would be broken: the compiler is forbidden from rearranging elements of a struct, for example. Clearly, this scheme is unlikely to work for existing large code bases if implemented directly.

The Address-Derived Capability Lifetimes Scheme

I wanted to design a scheme that incorporates ideas from both of these, with the objectives being that it:

- Provide full temporal safety for stack memory.
- Support as large a subset of valid C and C++ code as possible.
- Have a per-capability overhead sufficiently small to be viable in microarchitecture.
- Not require substantial modifications to the existing CHERI ISA.

The main idea behind the *address-derived capability lifetimes* scheme is to constrain the flow of capabilities by comparing their addresses *but ignoring the last few bits*. If we had a way to specify how many bits to ignore, then this would implicitly describe power-of-two-aligned stack frames without as much of an overhead of having to store per-capability lifetime counters.

I propose to reserve just three bits of per-capability metadata to describe the stack frame's size. Seven of the eight possible values shall indicate a range of power-of-two stack frame sizes from 64 bytes to 2048 bytes. The eighth state shall be reserved for capabilities pointing to non-stack addresses: this means that whether a capability points to the stack shall be an architecturally-visible property, and hence that stack- and non-stack capabilities can be used identically as operands. This also avoids the problem of having to decide whether to

include a check for ambiguous capabilities that could viably point to the stack or the heap at compile-time, such as:

```
int *p = (random() > 0.5) ? aStackPointer : aHeapPointer;
```

Address comparison will be performed by applying an AND mask to the base addresses of the source and target capability. Interpreting the three bits as an unsigned integer $N \in [0, 7]$, the function

$$\text{mask}(N) := \begin{cases} ((2^{64} - 1) \ll (N + 6)) \mod 2^{64} & \text{if } N \leq 6 \\ 0 & \text{if } N = 7 \end{cases}$$

computes the mask that should be applied. The special case for $N =$ `0b111` ensures correct functionality for non-stack capabilities: heap capabilities can be stored on the stack, but stack capabilities cannot be stored on the heap, identical to the capability lifetimes scheme.

I plan to implement a new instruction, `CSCCheckStackLifetime`, that uses an operand format identical (by design) to the existing `CSC` (Capability Store Capability) instruction. The instruction compares the addresses of the operand capabilities after applying each capability's AND mask to its base address. The base should be used rather than the capability's address field, because the address may not lie within the capability's bounds.

If the addresses are correctly arranged, then this instruction will do nothing. If the addresses are incorrectly oriented, then the instruction will raise a new `StackLifetimeViolation` exception. In pseudocode, assuming a downwards-growing stack:

```
CSCCheckStackLifetime(capability destination, capability source) {

    destinationMask = mask(destination.stackFrameSizeBits)
    sourceMask = mask(destination.stackFrameSizeBits)

    if (source.base & sourceMask < destination.base & destinationMask) {
        throw StackLifetimeViolation
    }
}
```

I plan to implement: a Sail model of this instruction; a QEMU implementation of its behaviour; a microarchitectural implementation in the Flute processor; and the relevant modifications to Clang/LLVM and CheriBSD to use the new instruction.

Additional Considerations

- The motivation for implementing the scheme as separate "check" and "store" operations, rather than as a single "store or throw an exception" instruction, is that this permits a theoretical check-eliding compiler pass that eliminates the call to the check instruction if data-flow analysis confirms that the operands definitely will not include stack addresses. If I have time, I'd like to implement this feature.
- Forcing stack frames to align to powers of two wastes some of the process's address space. However, selecting 64 bytes as the smallest possible size means that D-cache hit rates are unlikely to be affected, since most caches use a granularity of at least 64 bytes anyway.
- Not all stack frames will fit into the 2 KiB limit. The immediate solution will be to spill the largest allocations to the heap. Time permitting, there are some optimisations that I would like to try:
  - It's only the arrangement of address-taken values that matters. Non-address-taken variables can reside outside of the 2 KiB beginning of the stack frame.

– Using escape analysis we could do even better, packing only the function's *possibly-escaping* address-taken variables into the 2 KiB slot and ignoring the address check for address-taken variables to which a pointer never leaves the stack frame.

– While it will probably be out of scope for this project, it may be possible to arrange the possibly-escaping and address-taken values across multiple 2 KiB frames by dividing them into chunks of 2 KiB or smaller with no circular dependencies between the chunks, then linearising the chunks using a topological sort to obtain a memory ordering. Different chunks would essentially be treated as having separate lifetime levels, but the analysis would have confirmed from the topology of the points-to graph that this will never cause a `StackLifetimeViolation` exception. Unfortunately, I suspect that even being able to determine whether such a split into chunks exists is an NP-complete problem.

Scheme 4: ADCL with SSLT Fallback

The address-derived capability lifetimes scheme provides full temporal safety, hopefully with a substantially smaller overhead than existing software-only schemes and definitely with a smaller per-capability spatial overhead than the capability lifetimes scheme. However, like the capability lifetimes scheme, it would break valid C and C++ programs.

The final scheme I want to implement is to use ADCL but with a fallback to SSLT in the `StackLifetimeViolation` trap handler. This way, the common case for well-behaved code remains fast, using a hardware-implemented `CSCCheckStackLifetime` instruction when the compiler deems a store to be possibly-escaping and possibly-stack, but reverting to a slower software-only scheme only when the fast hardware scheme would otherwise reject valid code.

Recall that the lifetime-based schemes are all conservative: a `StackLifetimeViolation` exception does not mean that a temporal safety violation has occurred, it merely indicates that the lifetime-based scheme can no longer guarantee that one will not occur. The hope is that this is sufficiently rare that performance remains closer to that of ADCL than SSLT.

Summary

- **[Compile-time] If the source operand of a CSC instruction is a "possibly-stack" capability**

  Issue a `CSCCheckStackLifetime` instruction immediately before the `CSC` instruction, using exactly the same operands.

- **[Compile-time] If a function required insertion of any `CSCCheckStackLifetime` instructions**

  Emit code in the function's epilogue to increment all relevant entries in the shadow stack lifetime map.

  We have to guarantee that the lifetime counters are updated whenever the SSLT scheme might be used. The absence of a `CSCCheckStackLifetime` instruction guarantees that a `StackLifetimeViolation` exception will not be thrown.

- **[Runtime] When a StackLifetimeViolation occurs**

  Replace the target capability with a zero-length one (meaning that its base and end are the same address) derived from the frame or stack pointer. Reserve the fourth state of the local-global bits to indicate that the capability is in SSLT mode.

- **[Runtime] When a capability bounds exception occurs**

  In the trap handler, check whether the local-global bits are in SSLT mode. If they are, attempt to dereference the pointer using the SSLT scheme, all from the trap handler. If not, use the existing capability bounds exception-handling logic (most likely terminating the process).

Note that this applies for both loads and stores: the destination capability may also be in SSLT mode.

- **[Runtime] When the SSLT scheme detects a definite temporal safety violation**

  In this case, we need a new type of exception to distinguish *possible* temporal safety violations (`StackLifetimeViolation` exceptions) from confirmed temporal safety violations. For this, I propose to throw a new exception `TemporalSafetyViolation`, which should halt the program much like a CHERI bounds exception would.

To be clear: I don't expect using the trap handler to be fast. I do, however, expect it to be rare enough that it shouldn't impact performance dramatically, especially for "well-behaved" programs. Programs that do not use possibly-escaping local variables will suffer no time penalty, since the shadow stack lifetime map will never be updated, and with an escape analysis pass it would be possible to confirm statically that no such exceptions will ever occur. For systems with *demand paging* it's possible that there will be no space overhead for the shadow stack lifetime map either. With lazy page zeroing, there will be no zeroing overhead at process startup.

One issue is that the lifetime counters may eventually overflow if a process uses a very large number of function calls before it terminates. A solution would be to perform a revocation sweep before this happens. Recall that, in this model, not only are capabilities architecturally visible, but it can also be determined architecturally whether a capability points to the stack, and whether it is in SSLT mode. A revocation sweep would be able to revoke all SSLT-mode capabilities whose lifetime counters are now too old, and then reset all counters in the shadow stack lifetime map to zero. With this, the scheme would continue working indefinitely, and such sweeps would likely be far less frequent than those used for heap temporal safety for almost all programs.

Comparison Table

Name	Type	Expected Overhead	Breaks valid code?
Revocation	Hardware+Software+Compiler	Dreadful	No
SSLT	Software+Compiler	About 100%	No
ADCL	Hardware+Compiler	Low	Yes
ADCL+SSLT	Hardware+Software+Software	Still fairly low	No

References

[1] "Cornucopia: Temporal Safety for CHERI Heaps - IEEE Conference Publication." https://ieeexplore.ieee.org/document/9152640.

[2] 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*, Jun. 2010, pp. 31–40, doi: 10.1145/1806651.1806657.

[3] S. Tsampas, D. Devriese, and F. Piessens, "Temporal Safety for Stack Allocated Memory on Capability Machines," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, 20190601, pp. 243–256, doi: 10.1109/CSF.2019.00024.

[4] "Toward Complete Stack Safety for Capability Machines (PriSC 2021) - POPL 2021." https://popl21.sigplan.org/details/prisc-2021-papers/2/Toward-Complete-Stack-Safety-for-Capability-Machines.

[5] "Stack temporal safety with local global bits," *GitHub*. https://github.com/CTSRD-CHERI/cheri-architecture.