Temporal Stack Safety for CHERI

Benjamin Cole
Clare College

A dissertation submitted to the University of Cambridge
in partial fulfilment of the requirements for Part III
of the Computer Science Tripos

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
UNITED KINGDOM

Email: bc465@cam.ac.uk

June 18, 2021

Declaration

I, Benjamin Cole of Clare College, being a candidate for Part III of the Computer Science Tripos, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

I, Benjamin Cole of Clare College, am content for my dissertation to be made available to the students and staff of the University.

Total word count: 11,992

Signed: 

Date: June 18, 2021

This dissertation is copyright ©2021 Benjamin Cole.
All trademarks used in this dissertation are hereby acknowledged.

Acknowledgements

This might be my project, but it would be nothing without the help from so many wonderful people in the CHERI group. I want to thank in particular: Peter Rugg, Jessica Clarke, Wes Filardo, Lawrence Esswood, and of course my supervisor, Robert Watson, for his wisdom and patience.

This project builds entirely on existing work by the CHERI group. I claim credit for nothing but the changes I applied on top.

Abstract

Memory-safety vulnerabilities are a scourge of computer programming, costing billions of dollars of damage every year. The CHERI project aims to make computing safer by enforcing memory safety at an architectural level, relying on capabilities as a new primitive for memory references. Capabilities are an effective method of preventing out-of-bounds memory accesses, and recent research has shown that their revocability makes efficient temporal safety for heap-allocated values possible too. However, there is currently no such scheme to protect the stack.

I introduce *capability-derived lifetimes*, a novel scheme to eliminate use-after-free and use-after-reallocation bugs for stack-allocated values in C and C++ applications using CHERI. By enforcing stricter alignment requirements for stack frames and associating metadata with each capability to encode the size of the pointed-to frame, it becomes possible to infer the frame-start address for any capability to the stack, and using a new *lifetime check* instruction alongside any pointer-type stores, raise an exception whenever a possibly-unsafe store happens that may cause a dangling capability to exist. I implement and evaluate a prototype using extensions to Clang, an industrial-strength C and C++ compiler; CheriBSD, a fork of the FreeBSD operating system adding CHERI support; and Toooba, an out-of-order superscalar CHERI-RISC-V processor core.

Microbenchmarks indicate function-calling overheads of under 6% and negligible per-store overheads in the common-case. Microarchitectural overheads are similarly small: the changes to Toooba caused only a 4% increase to static power consumption, with negligible area and timing overheads.

Contents

1 Introduction **1**
- 1.1 The Problem of Temporal Memory Safety . . . 2
- 1.2 Exploiting a Stack Temporal-Safety Vulnerability . . . 4
- 1.3 Design Goals and Evaluation Criteria . . . 4
- 1.4 Approach . . . 6
- 1.5 Contributions . . . 7

2 Background and Related Work **9**
- 2.1 Software-Only Methods for Stack Temporal Safety . . . 9
  - 2.1.1 Escape Analysis . . . 9
  - 2.1.2 Compiler-Enforced Temporal Safety . . . 10
- 2.2 Architectural Approaches . . . 10
  - 2.2.1 Cornucopia . . . 11
  - 2.2.2 Capability Lifetimes . . . 11
  - 2.2.3 Monotonic Capabilities . . . 12

3 The Capability-Derived Lifetimes Model **14**
- 3.1 Overview . . . 14
- 3.2 Additional Considerations . . . 17
- 3.3 Retaining Full C and C++ Compatibility . . . 17
- 3.4 Performance Optimisations . . . 18
- 3.5 Summary . . . 18

4 Proposed Architectural Changes **20**
- 4.1 Capability Metadata and Encoding . . . 20
- 4.2 Frame-Size Metadata Inheritance . . . 22
- 4.3 New Instructions . . . 22
  - 4.3.1 The `ccsc` Instruction . . . 22
  - 4.3.2 The `csfs` and `cgfs` Instructions . . . 23
  - 4.3.3 Architectural Optimisations . . . 24
- 4.4 Formal Modelling . . . 24

5 Prototype Implementation **26**
- 5.1 ISA Emulation Support: QEMU . . . 26
- 5.2 Compiler Support: LLVM . . . 27
  - 5.2.1 New Intrinsics . . . 28

5.2.2 Call-Frame Setup . . . 28
5.2.3 Inserting Lifetime Checks . . . 29
5.2.4 Handling Incompatible Functions . . . 31
5.2.5 Optimisation: Escape Analysis . . . 32
5.3 Operating System Support: CheriBSD . . . 33
5.3.1 Handling `StackLifetimeViolation` Exceptions . . . 33
5.3.2 The `caprevoke_stack()` System Call . . . 34
5.4 Microarchitectural Support: Toooba . . . 36

6 Evaluation **38**
6.1 Time Overheads . . . 38
6.2 Memory Overheads . . . 40
6.3 Microarchitectural Overheads . . . 41
6.4 Security Impact . . . 41
6.5 Software Compatibility . . . 42

7 Summary and Conclusions **43**

Appendices **44**

A A Believable Vulnerability **45**

B Microbenchmarks Source Code **48**

List of Figures

1.1 Temporal-safety violation examples. 3
1.2 A minimal control-flow exploit of a stack temporal safety vulnerability. 5

2.1 Monotonic capability lifetimes. 13

3.1 An example C program. 15
3.2 Implicit lifetime regions. 16
3.3 Performance-compatibility trade-offs of stack temporal safety schemes. 19

4.1 Histograms of static stack-frame sizes in Clang, Grep and `ocamlrun`. 21
4.2 The formal semantics of the `ccsc` instruction. 25

5.1 Lowered stack-frame layout. 30
5.2 A stack lifetime violation in a non-escaping function. 33
5.3 Control-flow paths between the userspace process and CheriBSD kernel. . . . 35

6.1 General overheads of capability-derived lifetimes. 39
6.2 Breakdown of capability-derived lifetime overhead sources. 40

List of Tables

3.1 Stack and heap interaction in the capability-derived lifetimes scheme. 17

6.1 PPA overheads of capability-derived lifetimes in Toooba. 41

Chapter 1

Introduction

Memory-safety violations are a scourge of computer programming: fully 70% of vulnerabilities identified in both Google Chrome and in Microsoft's software stack have been attributed to them [1, 2]. Despite being an extensively-studied problem [3], memory-safety vulnerabilities have facilitated attacks ranging from the original Morris worm [4] to the recent WannaCry ransomware attack [5], believed to have caused as much as $4bn of economic damage [6]. An efficient mitigation would spare countless hours of developer time and embarrassment, and would help to make the computing infrastructure upon which we increasingly rely immeasurably safer.

CHERI [7], an ISA extension, tackles the memory-safety problem at an architectural level. CHERI augments an instruction set with unforgeable, architectural *capabilities* that can be used to implement C and C++ pointers [8, 9]. Capabilities encode more than just a reference to a memory address: their metadata include explicit *bounds* to constrain the range of addresses in which they can be used, as well as *validity tags* that allow capabilities to be revoked. Any attempt to dereference an untagged capability, or a valid capability used beyond its bounds, is quickly met with a hardware-level exception. CHERI, combined with an appropriate memory allocator and runtime environment, enforces full *spatial* memory safety for applications, rendering countless classical vulnerability categories like buffer overflows impossible.

However, *temporal* safety in CHERI – the problem of preventing the dereferencing of capabilities outside the *lifetimes* of the objects they point to – is far from solved. Cornucopia [10], an implementation of CHERIvoke [11], extends CHERI to provide efficient temporal safety for *heap*-allocated values, but at the time of writing, no such analogue exists to protect the *stack*. The goal of this project was to extend CHERI and Cornucopia to incorporate protection for stack-allocated values as well.

1.1 The Problem of Temporal Memory Safety

Programmers often refer to objects as possessing a *lifetime*, a duration for which the object (and any pointer pointing to it) is valid. This lifetime begins at the point of initialisation [12], after memory to hold it has been acquired such as through a call to `malloc()` for a heap-allocated value, and continues exactly until that memory is released [12], such as through a call to `free()`. Dereferencing a pointer issued by some allocation outside the lifetime of that particular allocation constitutes a *temporal-safety violation*, even if the underlying memory has since been reused for something else. A simple mismatch in data interpretations like this can easily be enough to let an attacker gain influence over program execution [11].

For stack-allocated values, including local variables in C and C++, lifetimes correspond to lexical scopes. Function-wide local variables are typically allocated and initialised during a function's prologue, by incrementing or decrementing the stack pointer, and deallocated during the epilogue by returning the stack pointer to its original location. A pointer to a local variable in some function can therefore only be dereferenced *safely* while that function is still in execution and its stack frame remains in the call stack. Once the function terminates, its stack memory will be deallocated and possibly reallocated for the next function call; attempting to manipulate a capability that was supposed to point to the original frame might cause unintended consequences for the newer function call frame in its place.

Dereferencing a pointer after the pointed-to object has been deallocated is a *use-after-free* violation. Use-after-free violations on their own, while still erroneous, tend not to be overly dangerous because the lack of data confusion means they rarely lead to security vulnerabilities [11]. However, if at the time of dereference that address has already been reused, then we instead have a far more dangerous *use-after-reallocation* violation. A temporally-invalid capability to the stack might alias with a more-recently-called function's local variables or even return address, neither of which an attacker should be able to modify. Because of the stronger primitives such a vulnerability might give an attacker, this dissertation focuses on the automatic mitigation of use-after-reallocation vulnerabilities, with protection against use-after-free vulnerabilities seen as an added bonus. Figure 1.1 gives examples of heap and stack use-after-reallocation violations, indicating in particular where the invalid dereferences occur.

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    int *p = malloc(sizeof(int));
    free(p);
    int *q = malloc(sizeof(int));
    *q = 1;

    // The invalid dereference:
    *p = 2;

    printf("The value: %i\n", *q);
    free(q);
}

// The value: 2
```

(a) Heap temporal-safety violation

```c
#include <stdio.h>

void callee(int **p) {
    int x = 10;
    *p = &x;
}

int main(int argc, char **argv) {
    int *p;
    callee(&p);
    printf("The value: ");

    // The invalid dereference:
    printf("%i\n", *p);
}

// The value: 32766
```

(b) Stack temporal-safety violation

Figure 1.1: *Temporal-safety violation examples.* Both of these make use of *use-after-reallocation* violations to produce unexpected results. In the heap example, calling `free(p)` released the memory for the next allocation, which the subsequent call to `malloc()` then released for the second allocation, `q`. Reusing heap addresses in this fashion helps to improve cache efficiency. Because `p` and `q` both point to the same address, mistakenly updating the value pointed to by `p` actually updates the value pointed to by `q`. In the stack example, the local variable `x` pointed to by `p` is deallocated automatically by the return from `callee()`, and the address in question is reused simply by calling another function. `p` therefore becomes a temporally-invalid pointer as soon as `callee()` returns; merely the act of calling `printf()` clobbers the value stored there. In both cases, dereferencing a pointer expected to point to an object that has been deallocated since the pointer was released causes unexpected behaviour to occur.

1.2 Exploiting a Stack Temporal-Safety Vulnerability

Use-after-reallocation vulnerabilities present myriad ways of manipulating control flow. A type-confusion attack might use an invalid pointer to overwrite the function's return address, for example by attempting to write an innocent 64-bit integer intended for another purpose. CHERI capabilities prevent classical buffer-overflow attacks against such data using capability bounds, but do not automatically prevent a capability to an out-of-lifetime stack address from being dereferenced in a spatially-valid fashion. Tsampas *et al.* offered an example of how this could be used to change a function's return address in their paper on capability lifetimes [13]. So-called *stack smashing* attacks have been studied extensively [14, 15], and existing techniques like *stack canaries* [16] exist to mitigate return-address hijacking, albeit at a run-time cost likely to be greater than that of a hardware-level mitigation.

A subtler attack is to use a temporally-invalid data capability to influence a data-dependent control-flow decision, modifying stack-allocated local variables to change the victim function's behaviour. A minimal example of such an attack is given in Figure 1.2: the trick is that the temporally-invalid capability points to memory that overlaps with the call frame in execution, in this case `attemptAdminLogin`, so an attempt to write to it actually overwrites the stack-resident Boolean flag denoting whether the user is an admin. In this particular exploit, a carefully-prepared user input would offer privilege escalation for the attacker because it enables them to bypass a permissions check. Importantly, observe that the attack requires no *spatial* bounds violation.

Such vulnerabilities often appear contrived and overly-artificial. Appendix A provides an obfuscated variant to emphasise how easily this kind of error could creep into a large software system – especially one featuring modular software components that make false assumptions about each others' guarantees. Even in the minimal variant, neither function individually behaves incorrectly or maliciously; it is only in their composition that the vulnerability emerges. A recent Microsoft paper [17] showed an example of how a simple stack use-after-free violation in a JavaScript runtime could lead to a code execution vulnerability.

1.3 Design Goals and Evaluation Criteria

An ideal mitigation against such vulnerabilities would be simple and complement programmers' existing programming language mental models. New instructions should be of a similar *style* to the rest of the instruction set to ensure cohesion, much like when CHERI itself is applied to underlying ISAs. For any hope of real-world adoption, performance overheads

```c
#include <assert.h>
#include <stdio.h>
#include <string.h>

#define NAME_LENGTH 32

typedef struct user { char name[NAME_LENGTH]; } user_t;

// Imagine this sets a user-controlled value, e.g. from a UI field.
void setUserName(user_t *user) {
    memset(&user->name, 1, NAME_LENGTH);
}

// Imagine this loads the bit from a database.
int userIsAdmin(user_t *user) { return 0; }

// Sets up an initial user. The bug is that the stored pointer becomes
// out-of-lifetime as soon as the function returns.
void setupUser(user_t **user) {
    user_t newUser;
    strcpy(newUser.name, "Mr Anonymous Student");
    *user = &newUser;
}

// A valid login function that tests whether the user is an admin.
void attemptAdminLogin(user_t *user) {
    int isAuthenticated = userIsAdmin(user);
    assert(!isAuthenticated);
    setUserName(user);
    if (isAuthenticated) {
        printf("Authenticated\n");
    } else {
        printf("Not authenticated\n");
    }
}

int main(int argc, char **argv) {
    user_t *user;
    setupUser(&user);
    attemptAdminLogin(user);
    return 0;
}

// OUTPUT: "Authenticated" !!
```

Figure 1.2: *A minimal control-flow exploit of a stack temporal safety vulnerability.*

should be low, at least in the common-case, and software compatibility high. Microarchitecturally, that means avoiding changes that dramatically increase chip power consumption or area, or lengthen the critical-path for signals to traverse within one clock cycle. And, finally, as a security-motivated change, any proposed scheme should offer strong security guarantees: ideally, use-after-free and use-after-reallocation vulnerabilities should be mitigated *deterministically*.

1.4 Approach

CHERI's lack of built-in temporal safety prevents it from automatically mitigating such attacks. As a result, I was able to reproduce the attack against the CHERI-RISC-V architecture, running in QEMU.

However, CHERI's focus on architectural visibility of capabilities is a promising starting-point for a solution. As mentioned previously, Cornucopia [10] is a recent extension comprising hardware and software additions to CHERI to provide efficient temporal safety for *heap*-allocated memory. It works by *quarantining* freed addresses, then, once the *quarantine buffer* exceeds a tuneable size threshold, issuing a whole-address-space *revocation sweep* to remove any capabilities pointing to any revoked region. The quarantine buffer amortises the cost of scanning the whole address space, but the immediate reuse of stack addresses between successive function calls means that stack addresses cannot be quarantined in this way. At the time of writing, CHERI does not offer an analogous mitigation to protect the stack, but the fact that capabilities are architecturally visible and revocable provides a convenient starting point for potential solutions.

One important observation about stack-allocated objects is that, unlike those on the heap, the difference between their addresses hints at which object should outlive the other. Stacks, by their very nature, allocate and deallocate values using push and pop operations. For a downwards-growing stack, an object logically cannot be deallocated without all objects at numerically lower addresses being deallocated first. Hence, a capability pointing to an address numerically greater than its own memory location will never become temporally invalid: deallocating the pointed-to object is impossible without also deallocating the capability that points to it.

This dissertation proposes the idea of *capability-derived lifetimes*, which uses this observation to control the propagation of capabilities to prevent temporally-invalid stack capabilities from ever existing. New capability metadata indicating the power-of-two-aligned size of the stack frame pointed to, combined with stricter address-alignment requirements for the beginnings of functions' call frames, make it possible to infer unambiguously for any

pair of stack capabilities which one will be deallocated first. Inserting additional instructions to check the relative lifetimes of capabilities involved in store instructions, comparing the longevities of capabilities being stored against those pointing to the destination address, permits an efficient method of preventing even the creation of temporally-invalid stack capabilities. Finally, by creating a new instruction to perform such a check, and using that instruction to raise a new hardware-level exception with a custom trap handler to be called whenever a violation occurs, it becomes possible to detect in software whenever a lifetime violation occurred during the execution of a particular function, and fall back on a Cornucopia-style revocation sweep if one did. This provides automatic protection against *all* stack use-after-reallocation and use-after-free vulnerabilities, for all functions with compatible stack-frame sizes, avoiding the intolerable run-time overhead of issuing a revocation sweep after every function call without breaking any valid C or C++ code like competing lifetime-based solutions would.

Like Filardo *et al.* in the Cornucopia paper [10], I assume a threat model consisting of a non-malicious but fallible programmer capable of introducing stack temporal-safety vulnerabilities in any C (or C++) code they write. I assume that they write their programs exclusively in C – they cannot accidentally circumvent compiler-inserted protections by modifying the binary it generates, but they have total control over the program's C-level source code. They may attempt to read and write stack capabilities to any spatially-valid memory location at any time, including to the heap and process-wide global variables. While capability-derived lifetimes could in theory be used to protect kernel code, we principally focus on userspace code and its associated interactions with the OS via system calls. Because stack deallocation happens implicitly upon function return, rather than an explicit call like `free()` for heap addresses, this dissertation ignores the problem of double-free vulnerabilities.

1.5 Contributions

The contributions of this dissertation are threefold:

1. I propose *capability-derived lifetimes* (CDL), a novel abstract model for CHERI C and C++ stack temporal safety.
2. I describe a possible realisation of capability-derived lifetimes applied to the CHERI-RISC-V instruction set.
3. I evaluate a prototype implementation of capability-derived lifetimes by modifying Clang, CheriBSD and Toooba, an out-of-order CHERI-RISC-V processor core.

Overall, capability-derived lifetimes offer lower function-calling overheads of 5.9%, and

effectively zero per-store overhead for well-behaved code running on a superscalar processor. The microarchitural changes required were non-invasive, bringing negligible area overheads and only a 4% static power consumption increase. Time constraints prevented me from implementing full C and C++ compatibility in my compiler modifications, but the resulting prototype is stable enough for comprehensive microbenchmarks and could in principle be adapted for full compatibility with more development time.

Chapter 2

Background and Related Work

As the source of so many vulnerabilities, memory safety has already received extensive research attention from industry and academia. Schemes for temporal memory safety can be partitioned into those that detect potential violations statically, versus those that defer the detection until runtime. Some schemes require architectural support: others provide temporal safety in a purely software-driven manner. Szekeres *et al.* [3] provide further classification, including distinguishing between pointer-based and object-based schemes. Here we explore a range of existing and proposed schemes for stack temporal safety, focusing specifically on their safety guarantees, software compatibility, and performance overheads.

2.1 Software-Only Methods for Stack Temporal Safety

Software-only methods are a compelling solution for temporal safety because they require no hardware changes from users. Modern build systems like CMake [18] already help to abstract away the actual compiler in use: a well-implemented software-only scheme could in principle be enabled as part of a build system much like a new compiler optimisation, hugely simplifying incremental roll-out across a large codebase.

2.1.1 Escape Analysis

In object-oriented languages, an object is said to *escape* a method if it can be accessed from outside that method [19]. *Escape analysis* is a compiler analysis for determining whether a particular value may be escaping. Like many interesting analyses, escape analysis in general is undecidable, so all real-world escape-analysis routines are forced to over-approximate.

A commonly-proposed solution to the problem of stack temporal safety is to *reduce it to a*

heap temporal safety problem by moving escaping local variables to the heap and inserting `malloc()` and `free()` calls as appropriate [20]. Non-escaping local variables suffer no space or time overhead, although the necessary presence of false positives means that some variables incur the non-negligible burden of a heap allocation upon all function invocations.

The primary downside of such an approach is that one loses the fast allocation and deallocation speeds associated with stack-resident values. Allocating and deallocating values on the stack simply requires adjusting the stack pointer; `malloc()` and `free()` calls are comparatively far more expensive. Their nondeterministic execution time also means that many coding standards for embedded or real-time systems forbid their usage after program initialisation [21].

A standout feature of this approach, however, is its interoperability with existing code. Modern C and C++ applications typically make extensive use of library code: the ability to adapt a temporal safety scheme gradually across a codebase makes the scheme far easier to adopt.

2.1.2 Compiler-Enforced Temporal Safety

Compiler-Enforced Temporal Safety (CETS) [22] is a software scheme that uses a combination of compiler-based instrumentation and metadata structures to provide full temporal safety for C programs. CETS is a *lock-and-key* scheme, in which allocated objects are tagged with a lifetime counter ("the lock"), and pointers are augmented with an expected lifetime value ("the key") that must match for a dereference to succeed. Incrementing the lifetime counter of an address during deallocation ensures that subsequent attempts to dereference any pointers to it will fail. Applied on top of their existing scheme for spatial safety, the authors measured an average 48% overhead as a result of their mitigations.

The main sources of overheads were the additional memory loads and stores for manipulation of lock and key values, as well as the extra comparisons of key values for pointer-dereferencing operations. These could potentially have been implemented in hardware if architectural modifications were allowed.

2.2 Architectural Approaches

Clearly, software-only approaches are not ideal for performance-conscious users. CHERI's unique architectural advantages, such as its hardware-accelerated bounds-checking and architecturally-visible capabilities that are amenable to revocation, meant that many of the primitives implemented using software for CETS come "for free" with CHERI. For that

reason, I wanted to explore how the CHERI architecture could be used to optimise temporal safety schemes.

2.2.1 Cornucopia

Consider Cornucopia [10], the previously-mentioned CHERI extension providing full protection against heap use-after-reallocation vulnerabilities. Upon a call to `free()`, rather than reusing the freed memory region, the memory allocator instead adds that region to a *quarantine buffer*. It then *revokes* all capabilities pointing to quarantined memory addresses once the amount of quarantined memory exceeds a user-configurable threshold, implemented as a mostly-asynchronous sweep through a process's entire address space. This ensures that heap addresses will never be reallocated until no capabilities to them remain, making use-after-reallocation impossible.

Such a scheme would not be possible without CHERI's architecturally-visible and revocable capabilities, and nor would it be as safe without CHERI's architecturally-enforced bounds-checking to prevent out-of-bounds dereferences from accessing out-of-lifetime addresses. Using asynchronous revocation, Filardo *et al.* achieved an average execution-time overhead of just 2% for complete protection against heap use-after-reallocation vulnerabilities – substantially lower than that of CETS, albeit while only protecting the heap. Crucially, while a whole-address-space sweep is an expensive operation, Cornucopia's quarantining procedure amortises the cost across a fixed but large quantity of memory deallocations.

Unfortunately, as was explained in Section 1.4, the nature of stack-memory usage renders effective quarantining impossible. Memory regions used by one function call are likely to be reused immediately by the next, and executing revocation sweeps after all function calls would pose an untenable overhead. For that reason, Cornucopia is inappropriate for the problem of temporal stack safety. What Cornucopia did show, however, is the efficiency of integrating architectural support for higher-level protection mechanisms. Motivated by this, the rest of this chapter explores schemes that similarly exploit the CHERI architecture's unique advantages to provide temporal *stack* memory safety.

2.2.2 Capability Lifetimes

The *capability lifetimes* model [13] captures explicitly in architecture the already-familiar idea of object lifetimes. Tsampas *et al.* described an extension to capability systems, naming CHERI in particular, in which capabilities be augmented to include a *lifetime counter* that encodes the longevity of the pointed-to object. Capabilities to objects in "inner" lifetime regions, such as a callee's stack frame, should have a greater lifetime value than those in

"outer" lifetime regions, such as a caller's stack frame. Then, a modified store instruction can compare such lifetimes and reject any stores that would cause a capability pointing to a shorter-lifetime region to be stored in a longer-lifetime address. Combined with the spatial-safety properties of capability systems to prevent out-of-bounds accesses, this makes stack temporal-safety violations architecturally impossible. Stack capabilities would only ever be able to point to regions of the same or longer lifetime, so deallocating the object pointed to would entail deallocating any capability pointing to it as well.

Unfortunately, a weakness of the scheme is that eliminating all counter-lifetime stores breaks some valid (albeit unusual) C and C++ code. C semantics dictate only when a pointer may be dereferenced: they impose no constraints on when or where they may be stored. Additionally, the proposed 16 additional bits that would be required to encode such lifetimes [13] may not prove microarchitecturally viable if the increase to the size of capabilities causes a corresponding decrease to D-cache efficiency. Implementing capability lifetimes in hardware would require trading-off directly between software compatibility and microarchitectural efficiency: I wanted to devise a scheme that offered both.

2.2.3 Monotonic Capabilities

Finally, *monotonic capabilitites* [23] build on capability lifetimes to incorporate an additional observation: that *the addresses themselves* of stack-allocated objects necessarily must correspond monotonically to their lifetimes. For a downwards-growing stack, if object A were allocated before object B, then B's address necessarily must be numerically lower than A's, and A logically cannot be deallocated (popped) without also deallocating B; therefore, object B can safely hold a reference to object A, and the fact that B is still in-lifetime implies that A must be too. Figure 2.1 shows a graphical representation of this property.

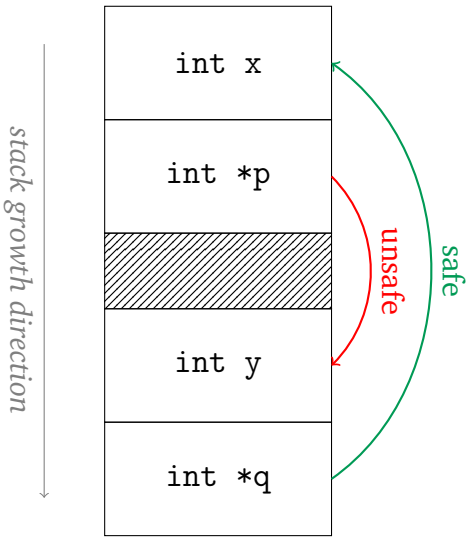
Unfortunately this breaks even more valid code than the capability lifetimes scheme because circular references between objects in the same stack frame would no longer be permitted. Relying purely on the addresses of stack-resident objects, rather than any extra metadata, causes the equivalence of addresses in the same stack frame to be lost. Nonetheless, the idea that lifetimes can be derived from addresses posed an excellent starting point for the primary contribution of this dissertation, the capability-derived lifetimes model.

```c
void f()
{
    int x;
    int *p;
    g(&x, &p);
}

void g(int *x, int **p)
{
    int y;
    int *q;

    q = x;    // Safe
    *p = &y;  // Unsafe
}
```

(a) Example code



(b) Stack arrangement

Figure 2.1: *Monotonic capability lifetimes.* We assume a downwards-growing stack. The safety of a capability store can be determined efficiently simply by comparing the two addresses. However, this eliminates the possibility of any circular references.

Chapter 3

The Capability-Derived Lifetimes Model

Capability-derived lifetimes (CDL) are a novel stack temporal safety scheme that combine and extend the ideas of capability lifetimes and monotonic capabilities. The aim is to prevent the existence of temporally-invalid stack capabilities by detecting at runtime whenever a capability to a longer-lifetime stack address is stored in the call frame of one of its callees, just like the lifetime-based schemes explored in Section 2.2.

A new encoding scheme dramatically reduces per-capability lifetime encoding overheads compared with the original capability lifetimes scheme, without imposing additional intra-frame ordering restrictions like monotonic capabilities do. By using a dedicated validation instruction rather than complicating the semantics of stores, a compiler could elide the check for any store it can prove safe, meaning that stores proven by the compiler to contain no escaping variables incur no runtime overhead. Employing a separate instruction for checks makes the scheme *opt-in*, rather than forcing the change of semantics architecturally. Finally, by falling back to a software handler such as a Cornucopia-style revocation sweep whenever a violation happens, the CDL scheme is *theoretically*[1] compatible with *all legal C and C++ programs*, a feat that neither of the aforementioned lifetime-based schemes achieve.

3.1 Overview

Previously-discussed lifetime-based schemes essentially use the same principle: capabilities pointing to longer-lifetime stack addresses are "safe" while those pointing to shorter-lifetime stack addresses are not. Both schemes rely on associating a "lifetime" value with capabili-

[1] Unfortunately, time contraints prevented me from implementing stack-passed function arguments, so my prototype is *not* compatible with all C and C++ code. See Section 5.2.2 for full details.

```
void outer() {
    char x[96];
    middle();
}
```

```
void middle() {
    char y[192];
    inner();
}
```

```
void inner() {
    char z[32];
}
```

Figure 3.1: An example C program.

ties, either explicitly as metadata or implicitly by treating the address *as* the lifetime value. Capability-derived lifetimes take a different approach: rather than recording the "depth" of the frame being pointed to, we instead round functions' frame sizes to the next power-of-two, encode that *size* as capability metadata, and use stricter function-alignment requirements to let us *derive* the start position of the pointed-to stack frame without having to encode it in the enabling capability.

As an example, suppose a function had a 200-byte stack frame. This would be padded up to 256 bytes, and the function's prologue would need to shift the stack pointer down until its last eight bits were all zero. Suppose the stack pointer at the time of invocation held the address `0x1234`: then this would be shifted down to `0x1200`, and any address inside that stack frame would be of the format `0x11XX`.

To infer the start of a frame from a capability pointing to it, we would read its *frame size metadata* and infer the size and alignment. Given a capability to the address `0x11ab` with metadata declaring its frame to be 256 bytes large, the beginning of that stack frame *must* be the address `0x1200`, which we can compute by appling the and-mask `0xff00` to obtain the end of the frame, followed by adding its size `0x0100`. I refer to the inferred frame-start address `0x1200` as the capability's *implied lifetime*.

The ability to infer frame-start locations is powerful because it lets us implement precisely the same store-rejection behaviour – stores should be rejected if they would lead to a shorter-lifetime capability being stored in a longer-lifetime stack frame – using far fewer per-capability metadata bits. Figure 3.2 shows the lifetime derivation process visually and reveals how successive frames compose.

The original capability lifetimes scheme required a number of metadata bits logarithmic in the expected maximum stack-frame depth, and fails if the program ever exceeds this. Tsampas *et al.* suggested using 16 bits [13] for this. In comparison, capability-derived lifetimes require a number of metadata bits that scales logarithmically with respect to the maximum stack-frame size. Given that programmers are typically encouraged to use the heap for large allocations, this quantity is essentially a constant, meaning that CDL offers a reduction from $O(\log n)$ metadata bits to $O(1)$. For example, one could use just three bits of metadata to describe seven possible frame sizes from 64 bytes to 4 KiB, plus a required additional "not

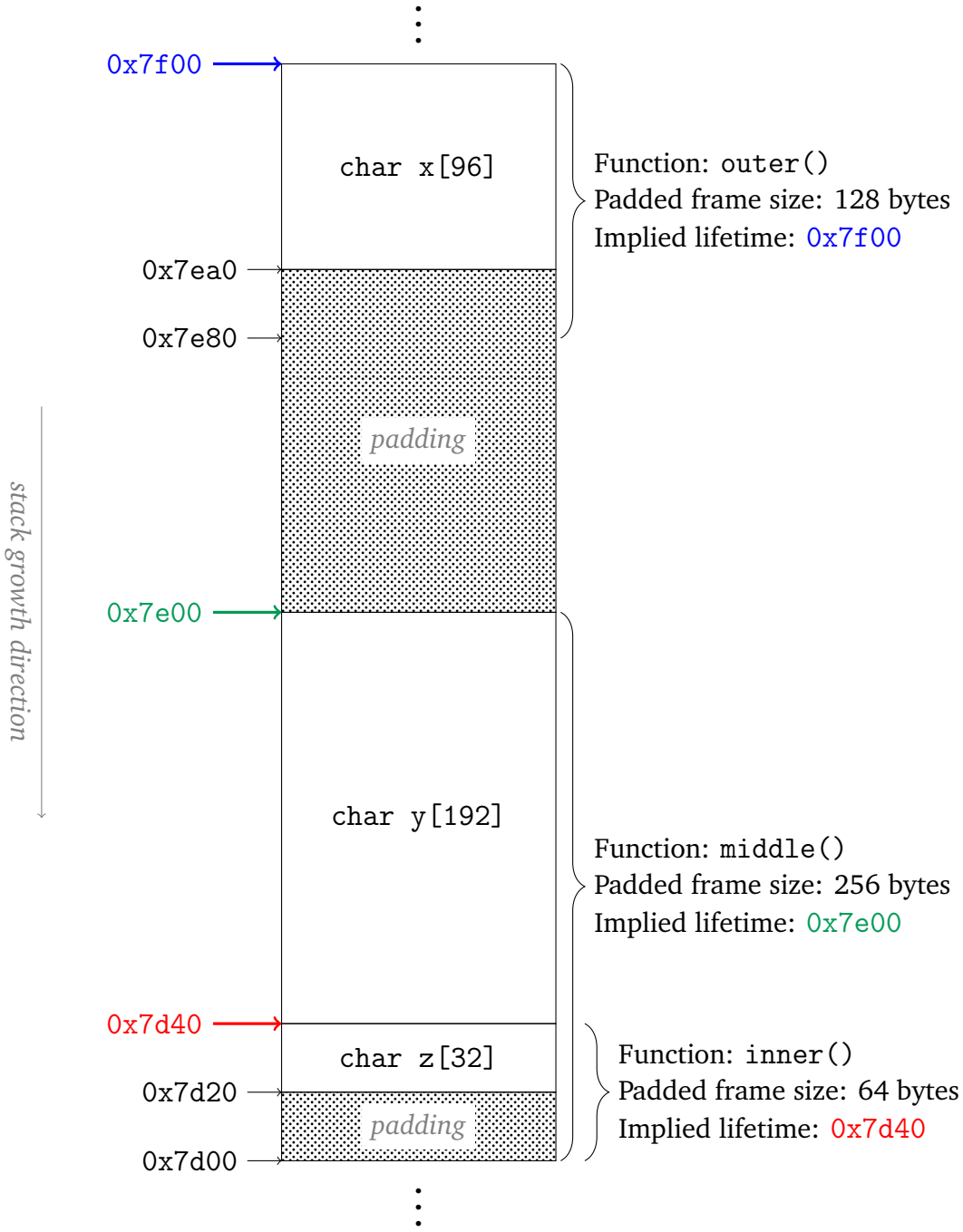


Figure 3.2: *Implicit lifetime regions.* Shows a possible in-memory representation during the execution of `inner()` from Figure 3.1. Because all possibly-dangerous stack frames are aligned, the implicit lifetime of any stack capability can be determined using simple bitwise arithmetic. Notice that, because `inner()` uses a smaller stack frame than `middle()`, it actually resides within the space spanned by `middle()`, although capabilities to the two frames will have differing frame-size metadata. We therefore use the *start* of the frame, rather than the end, to stop frames like these from aliasing.

	Can be stored on the heap	Can be stored on the stack
Capability to the heap	Always	Always
Capability to the stack	Never	If lifetimes are correctly oriented

Table 3.1: *Stack and heap interaction in the capability-derived lifetimes scheme.* If an untagged or out-of-bounds capability is used then this will be caught by the store instruction's existing safety checks, so we consider only the temporal validity of stores here.

a stack capability" state to handle stores where one or more involved capabilities do not point to the stack.

To complete the scheme, the architecture uses a new "check" instruction to verify the operands to a capability store. A modified compiler would need to insert this throughout the generated binary: the instruction does nothing if the lifetimes are correct, but raises an exception if they are not. Using a separate checking instruction, rather than just modifying the semantics of the existing store instruction, makes the scheme opt-in for those wishing to adapt large codebases gradually.

3.2 Additional Considerations

One subtlety is in how stack and non-stack (i.e. heap or global) capabilities are permitted to interact. Table 3.1 shows which directions of store are to be permitted: in short, heap capabilities can always be stored on the stack, but stack capabilities can never be stored on the heap. Rejecting all stores of stack capabilities to the heap circumvents the problem of needing to track their propagation.

3.3 Retaining Full C and C++ Compatibility

A weakness of all previously-proposed lifetime-based schemes discussed in Section 2.2 is that they each disallow certain legal stores, such as forbidding a caller from holding a reference to a callee's stack frame that it never dereferences out-of-lifetime. Rather than rejecting unusual but nonetheless valid code, the capability-derived lifetimes scheme improves on its predecessors by delegating the handling of a failure to software. If the particular domain requires full C and C++ compatibility, then an operating-system trap handler can *fall back to a software-based scheme for temporal safety* whenever the lifetime-based approach fails to prove a store safe. On the other hand, platforms with strong real-time requirements might instead choose to use only C and C++ code that makes correctly-oriented stores and terminate any offending processes.

In my prototype, I chose to issue a Cornucopia-style revocation sweep at the end of any functions that experience a lifetime violation, guaranteeing that no capability to the function's frame exists after the function returns. This is an application of the classic Computer Science principle of optimising for the common case, with a slower path for handling the more difficult situations.

3.4 Performance Optimisations

Possible optimisations exist both at the scale of functions and individual lifetime checks.

- **Functions** can be optimised using escape analysis. A function that provably does not contain any escaping local variables need not waste time aligning its stack frame, nor will it ever need to test-and-revoke before returning.
- **Individual lifetime checks** can theoretically be elided if data-flow analysis shows them not to refer to any escaping values.

Additionally, both of these would be known at compile-time, so a compiler warning can be issued. Performance-conscious users willing to restructure awkward portions of their code can therefore obtain a guarantee of freedom from any runtime checks with a suitably powerful compiler.

3.5 Summary

The proposed scheme is essentially a hierarchy of opportunities to prove a store instruction safe, falling back to the slower next level only if analysis cannot prove the absence of any violations. The first two stages would be common to all realisations of CDL; the final software-driven stage can be tailored to the needs of the particular environment. Issuing a revocation sweep is just one of many ways of handling it.

1. The best-case scenario happens when no runtime check is required at all. If escape analysis, combined with a suitable data-flow analysis, can prove a store not to refer to any escaping values, then lifetime checks can be elided. Similarly, functions containing no escaping locals need not waste stack space aligning themselves.
2. Failing this, the compiler emits `ccsc` instructions to validate at runtime the lifetimes of possibly-dangerous pointer-type stores, and emits frame-alignment logic in the prologue of escaping functions. For typical, well-behaved code, no lifetime exception will be raised.

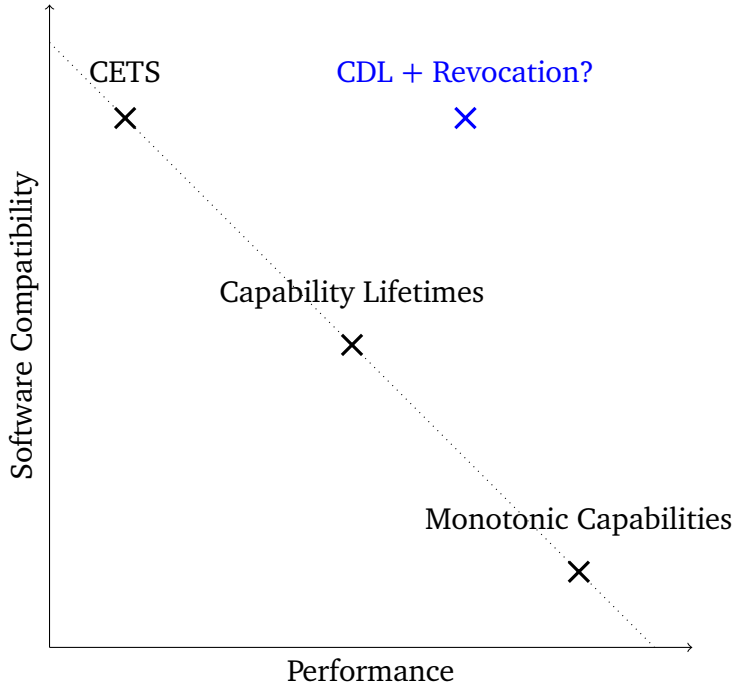


Figure 3.3: *Performance-compatibility trade-offs of stack temporal safety schemes.*

3. Finally, if dynamic lifetime comparisons still cannot prove a store to be safe, then the final level of protection is to revoke all capabilities to the stack frame. This is achieved by setting a bit during the exception's trap handler that is checked at the end of the function.

The prototype I implemented does not support the complete zero-overhead safe functions, but does elide conditional revocation sweeps for functions shown to be non-escaping by an escape-analysis pass. Finally, Figure 3.3 shows how it was hoped that the capability-derived lifetimes scheme might compare against alternatives in the performance-compatibility trade-off space.

Chapter 4

Proposed Architectural Changes

With a high-level overview of capability-derived lifetimes established, the next task was to decide on a realisation of the scheme for a real instruction set. I implemented my ISA changes atop the CHERI-RISC-V instruction set, one of the main architectures to which CHERI has been applied [7]. At a high level, I needed to:

1. Modify capabilities' metadata to add the new stack-frame size bits
2. Add a new "lifetime check" instruction to be emitted after pointer-type stores
3. Add new instructions for manipulating frame-size metadata
4. As a performance enhancement, add an additional instruction to accelerate derivation of *implied lifetimes*

4.1 Capability Metadata and Encoding

In Section 3, I explained the need for additional metadata in all capabilities to describe whether the capability points to the stack and, if so, how large the stack frame pointed to is. To determine an appropriate number of bits and range of sizes for them to span, I studied the static stack-frame sizes of three programs designed to showcase a range of code styles and workload types: Grep [24], a command-line tool for searching files for string patterns; `ocamlrun`, an interpreter for the OCaml programming language written in C [25]; and Clang [26] itself, an industry-standard C and C++ compiler. The tool I created used GLLVM [27] to translate each program's source code to LLVM IR, then summed the size of calls to `alloca()`, LLVM's instruction for stack allocation, for each function body. This unfortunately excludes any measurement of their dynamic size, such as if a C program calls *alloca()* for a dynamically-sized amount of memory or calls it at all within a loop, but

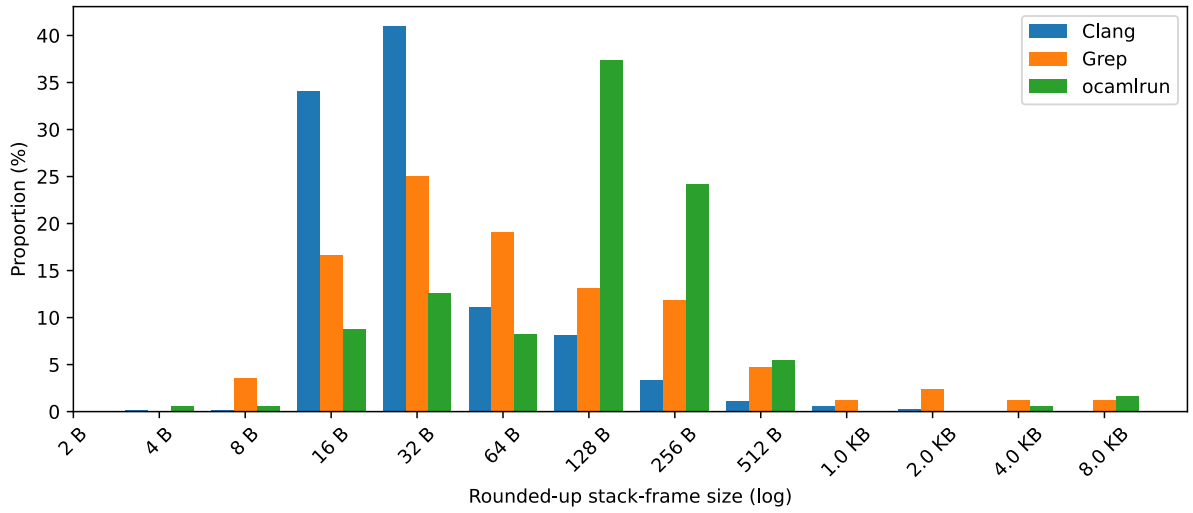


Figure 4.1: *Histograms of static stack-frame sizes in Clang, Grep and* `ocamlrun`. Estimated from LLVM IR representation by summing over statically-sized `alloca()` calls. This histogram does not include any dynamically-sized stack allocations, nor does it factor in the relative frequency of calls to each function.

such situations would be incompatible with capability-derived lifetimes anyway. The three programs were selected with the intention of analysing the stack usage of a short-running computation (Grep), a long-running computation (Clang), and a potentially indefinitely-running computation (`ocamlrun`). Figure 4.1 shows my findings.

While the stack-frame sizes in the three programs span too great a range of orders-of-magnitude to be described exactly using just seven states, I decided to use the range of 64 bytes to 4 KiB to cover all but the largest of frames. While this will cause a considerable number of 16 B and 32 B frames to be rounded up to 64 B, the small sizes involved should ensure that the overall wasted memory is low. I chose not to set 8 KiB as the maximum size because many of the large frames contained constructions that could easily be moved to the heap, such as stack-resident buffers for holding strings. Reserving three bits to describe sizes from 64 bytes to 4 KiB, plus an additional "not a stack capability" state, seemed an appropriate compromise between describing a representative range of frame sizes without reserving too much additional metadata. The huge range of orders of magnitude also affirmed my decision to encode frame sizes *logarithmically*.

The three metadata bits needed to come from somewhere, and unfortunately the CHERI ISA at the time of writing leaves none of its 128 bits unused. My solution was to shrink the *OType* ("object type") field from 18 to 15 bits. The OType field is used for domain transitions between compartments: pairs of sealed code and data capabilities are object references whose invocation triggers a protection-domain switch [7]. Fortunately, current

compartmentalisation models do not make intensive use of the full OType range, meaning I could reduce the size of the field by three bits without too great a loss of functionality.

4.2 Frame-Size Metadata Inheritance

CHERI includes *monotonicity* properties relating to the bounds of capabilities [7]. There is intentionally no architectural method of expanding the bounds of a capability because including such an instruction would trivially enable circumvention of the architecture's spatial-safety properties. Derived capabilities may not include wider bounds than their parent.

Monotonicity properties do not map exactly to frame-size metadata – increasing the frame-size value of a capability does not change its bounds – but I needed a similar rule to ensure that the data are preserved in derived capabilities. I enforce the invariant that any sub-capability (e.g. a capability to a subset of a stack frame derived from the frame pointer) must inherit the frame-size metadata of the original. This rule prevents capabilities to subsets of the frame from spuriously having a different frame-size value, which could confuse lifetime checks. A convenient consequence is that the compiler only needs to set the bits for the stack pointer at the beginning of a function: the value will then propagate naturally to all derived capabilities.

4.3 New Instructions

The main new instruction required to realise capability-derived lifetimes in the CHERI-RISC-V architecture was of course an instruction to verify lifetimes of pointer-type stores. Additionally, the scheme requires a method of reading and writing the frame-size bits of a capability, for use in a trap handler and function prologue respectively, and I included a fourth instruction to optimise flag-checking in function epilogues.

4.3.1 The `ccsc` Instruction

In keeping with the design goal of designing new instructions in a manner that matches the rest of the architecture, I named this new instruction `ccsc` (Check Capability Store Capability), after the existing `csc` (Capability Store Capability) instruction. By design, the instruction takes the exact same operand format as a `csc`, making it easy for a compiler to emit the instructions alongside stores. This also makes it easy to migrate to a hypothetical future version of the ISA in which the changes were integrated into the main `csc`

instruction, since the two are likely to be emitted from the same place in any compiler implementation.

There are myriad possible means of signalling from the checking instruction that a violation has occurred. I chose to create a new CHERI exception called `StackLifetimeViolation`, rather than embedding any sort of integration with the revocation framework architecturally, because handling the exception in software means that an alternative operating system would be able to respond to it differently. The CheriBSD trap handler that I produced responds by setting the bit to signal that revocation is required, but a hypothetical real-time operating system would still have the freedom to adopt the stricter semantics of just killing the process without any architectural change. In other words, lifting the exact interpretation and handling of a `StackLifetimeViolation` out of the architectural specification of the ISA gives operating systems and programming-language runtimes the freedom to interpret the exception however best suits their programming model and desired semantics.

The instruction is designed to be used only with a `csc` instruction, which already performs tag- and bounds-checking on its operands, so to avoid redundant computation I decided to omit any such checks in `ccsc`. Architecturally the instruction may validly be emitted by a compiler before or after its corresponding `csc`, but in practice I chose to emit the `ccsc` *after* the `csc` to avoid needing to handle the difficult case of processing any `StackLifetimeViolations` triggered by invalid stores.

4.3.2 The `csfs` and `cgfs` Instructions

Adding new metadata to capabilities necessitated adding instructions to `set` and `get` that data. I added the instructions `csfs` (Capability Set Frame Size) and `cgfs` (Capability Get Frame Size) that set the frame-size bits to an immediate operand and read them to a general-purpose register respectively.

I did not include a method for setting the frame-size bits from a register operand because the expectation was that emitting the `csfs` instruction would be a part of the frame-lowering process, which typically happens *after* register allocation, at which stage the frame size is known statically. This was certainly the case in Clang, the prototype implementation. Investigating ways of determining and setting frame-size bits at runtime for dynamically-sized stack frames might prove an interesting direction for future research.

4.3.3 Architectural Optimisations

Computing the *implied lifetime* of a capability is not necessarily just required for lifetime checks: when I implemented my prototype, I used it to compute the address at which to store a flag for indicating whether a lifetime violation has happened in each stack frame, so that the function can determine whether it needs to perform a revocation sweep before it returns. The mask-and-shift operation therefore lies within the critical path of *returning from any function* for any function that has had its frame pointer elided, and the fact that the logic is already used in the `ccsc` instruction means the hardware to compute it must already be present in any microarchitectural implementation. Improving lifetime-determination seemed an appropriate place to focus optimisation efforts.

I added a final instruction, `cgetframebase`, that takes an arbitrary stack capability and produces a capability to the start of its frame – the address previously referred to as its "implied lifetime". This reduces the process of inferring the start of the stack frame using the stack pointer from four instructions (`cgetaddr`, `andi`, `addi`, `csetaddr`) to just one.

4.4 Formal Modelling

Architectural specifications written in natural language often contain ambiguous, misleading and erroneous statements, even in specifications as heavily reviewed as those used by ARM [28]. To specify the capability-derived lifetimes architectural changes a bit more rigorously, I extended the official CHERI-RISC-V formal model, written in Sail [29], a language designed specifically for writing formal models of instruction sets. A Sail model of an ISA is both an executable specification and an encoding suitable as the basis for formal proofs; the CHERI-RISC-V ISA is *defined* in Sail [7]. As an example, Figure 4.2 shows the formal definition I wrote for the `ccsc` instruction.

```
function clause execute (CheckCapStoreCap(cs2, cs1)) = {
  let cap_source = C(cs2);
  let cap_destination = C(cs1);
  let source_cap_frame_base = getStackFrameBase(cap_source);
  let destination_cap_frame_base = getStackFrameBase(cap_destination);
  let source_lifetime = source_cap_frame_base.address;
  let destination_lifetime = destination_cap_frame_base.address;
  if destination_lifetime <_u source_lifetime then {
    handle_cheri_cap_exception(CapEx_StackLifetimeViolation, 0b0 @ cs2);
    RETIRE_FAIL
  } else {
    RETIRE_SUCCESS
  }
}
```

Figure 4.2: *The formal semantics of the* `ccsc` *instruction (simplified)*. The instruction raises a `StackLifetimeViolation` exception if a store with incorrectly-oriented lifetimes happens, or acts as a no-op otherwise. Prepending a 0-bit to the capability register index is required just for internal register-encoding reasons. The *source* capability is blamed so that a software trap handler can identify it and the frame it escaped from.

Chapter 5

Prototype Implementation

Implementing the capability-derived lifetimes prototype required modifying a substantial cross-section of the CHERI stack and working on multiple large codebases. At a minimum, I needed to modify a processor to add the new instructions, an operating system to support the new `StackLifetimeViolation` exception, and a compiler to set up stack-frames with the new metadata and insert lifetime checks into the code. The primary artefacts I modified were Toooba, an out-of-order CHERI-RISC-V processor core [30]; the CHERI fork of Clang [26], an industrial-strength C and C++ compiler; and CheriBSD [31], a fork of the FreeBSD operating system [32] adding CHERI support. An additional and noteworthy component of the CHERI stack is a fork of QEMU, an ISA emulator. I began by applying the ISA changes to QEMU's CHERI-RISC-V mode so that I could work on the rest of the project without needing to wait to use shared FPGAs.

A common failure mode when working with such low-level systems – as I discovered first-hand! – is encoding misinterpretations between components, as might for example occur if a typo meant that the compiler and ISA emulator disagreed about the encoding of a `ccsc` instruction. To mitigate this as best as possible, I followed the principle employed by the CHERI-RISC-V team of treating the Sail model as the definitive source at all times.

5.1 ISA Emulation Support: QEMU

QEMU [33] is a C program, and unlike with actual microarchitectural implementations of an ISA, those working on the QEMU source code need not be quite so cautious about needing to hit timings along critical paths. This makes the implementation of self-contained new instructions largely straightforward.

Easily the most complicated part of the process was updating the central "capability" data

structures. QEMU uses two encodings: one, a standard C struct with meaningful representations of capability metadata (such as flags, bounds information, and of course, stack-frame size) stored in separate fields for easy access; and a second *compressed* representation that packs that information into 128 bits plus a tag bit for writing to memory. Adding a `stack_frame_size` metadata field required more than just extending both of these structs: I also needed to recompute manually the value that a *null capability* should take. Only 64 of the bits in a 128-bit capability are the address it points to, and even a zero-width, zero-permissions capability has nonzero bits for some of this metadata. I initially failed to do this, causing curious CheriBSD to fail to boot, panicking with error messages complaining about syntax errors in initialisation scripts. Additionally, the actual structs holding capability information are generated by macros to produce implementations for both 32- and 64-bit RISC-V automatically. I therefore needed to specify nonsense offsets for the `stack_frame_size` metadata in a 64-bit capability (corresponding to 32-bit CHERI-RISC-V) to make QEMU and its static assertions compile correctly, even though I had no intention of completing a full 32-bit RISC-V implementation.

Once this was complete, implementing the new instructions entailed simply adding the new instruction encodings to the CHERI-RISC-V-specific table and setting up various functions to dispatch instructions to their implementing functions. The instructions' implementations are essentially just translations of their Sail semantics into equivalent C code, minding QEMU's internal translations between compressed and expanded capability representations.

5.2 Compiler Support: LLVM

The reference CHERI software stack uses Clang/LLVM is its compiler [7]: it was therefore the natural choice for implementing the required compiler-side changes to support capability-derived lifetimes. The LLVM project defines an intermediate language called *LLVM IR* (Intermediate Representation), and provides the LLVM library that contains a plethora of compiler-related routines like code generation and optimisation passes. Clang, also part of the LLVM project, is a C and C++ front-end built on top of LLVM. My modifications were all applied to the CHERI-RISC-V back-end of the LLVM library.

At a high level, four main modifications were needed:

- Adding code-generation support for the new instructions in the CHERI-RISC-V back-end and corresponding *intrinsics* for each so that they can be referenced programmatically in target-independent passes.

- Extending the frame-lowering code in the RISC-V backend to insert stack-alignment logic into escaping functions' prologues and prepare the "did a lifetime exception happen?" flag, as well as saving and restoring the caller's stack pointer and frame pointer (if used) so that their frame-size bits are preserved upon return to the caller.
- Adding a compiler pass to insert lifetime checks before any pointer-type stores.
- Adding a conditional `caprevoke_stack()` system call (see Section 5.3) to be executed in functions' epilogues if a lifetime violation happened for a capability pointing to that frame. I implemented this in the same pass as the one that inserts lifetime checks.

5.2.1 New Intrinsics

Compiler *intrinsics* are a way to represent specific functions or instructions that can be lowered in a target-specific way. For example, LLVM defines an intrinsic to encode the `memcpy` function so that analysis passes can identify and reason about such a common and well-defined subroutine more precisely than just an arbitrary function call. All other CHERI instructions are implemented as LLVM intrinsics, and at the time of writing the LLVM documentation recommends creating a new intrinsic for anyone looking to add support for a new instruction [34], so that was the approach I used.

LLVM internally uses a custom language called TableGen to define tabular information like intrinsics and instruction encodings. TableGen files are automatically used to generate C++ data structures corresponding to the intrinsics and instructions as part of the build process. Conveniently, this means that fiddly details about opcodes and operand formats are all kept in a single place. The CHERI LLVM fork adds a TableGen file listing all CHERI-specific intrincs, and a similar file exists for the code-generation engine. I added my four new instructions to each, using existing patterns where appropriate.

5.2.2 Call-Frame Setup

Modifying LLVM's RISC-V-specific frame-lowering logic was easily the trickiest part of the whole prototype implementation. LLVM and CheriBSD's interpretations of memory layouts needed to match exactly to avoid garbage data being written and read, and the epilogue needed to restore the stack to precisely the same state as it was in before the prologue. Additionally, there were separate code paths for functions needing conditional revocation, functions needing unconditional revocation because they were incompatible with CDL, and "safe" functions requiring no revocation (see Section 5.2.5). Various bugs during development manifested through variables in the generated binary holding strange values read

from the wrong place, rather than just producing invalid code, and debugging was challenging given that my frame-layout changes and additional instructions broke compatibility with GDB [35], a popular debugging tool.

The memory layout I ultimately used is shown in Figure 5.1. It isn't the most memory-efficient representation, but separating the three components reduced the number of instructions needed to store and recover them that would be required for a more complicated packing method. My compiler omits saving the frame pointer for functions that do not use one, because such a function is not going to clobber any value stored in that register. I used a highly inefficient capability-sized flag slot for tracking whether a `StackLifetimeViolation` had happened, even though only one bit is needed, because the capabilities above it require that amount of memory alignment anyway, and putting the flag above the saved capabilities would have complicated the `StackLifetimeViolation` trap handler's implementation since it would then need to know whether the saved frame pointer is present.

Saving the stack pointer on the stack might appear strange: to understand why this is necessary, recognise that aligning the stack pointer *discards information*. I chose to store both the original stack and frame pointers, rather than the common delta that they were both shifted by, because this way the original pointers' frame-size metadata could be restored "for free" in the process.

Unfortunately, time constraints meant I was unable to implement support for functions that need to spill arguments onto the call stack. Arguments are placed by the caller, unlike local variables which are placed by the callee; aligning the stack pointer during functions' prologues essentially splits the stack frame into two, causing any references to stack-resident function arguments to point to garbage memory. I had hoped to solve this either by reserving a virtual register to use as a second frame pointer for affected functions to use to access their parameters, or alternatively by performing a `memcpy` of arguments during the alignment process into their corresponding location in the aligned frame.

5.2.3 Inserting Lifetime Checks

LLVM provides a library of *passes*, operations that can be applied to programs at the scale of functions or modules. These correspond to the `FunctionPass` and `ModulePass` interfaces respectively. Additionally, most passes can be categorised as being either an *analysis* or a *transformation* depending on whether it modifies the object it is applied to. Passes may operate at different levels in the lowering process: for example, a `FunctionPass` operates on target-independent IR functions, while a `MachineFunctionPass` operates on target-

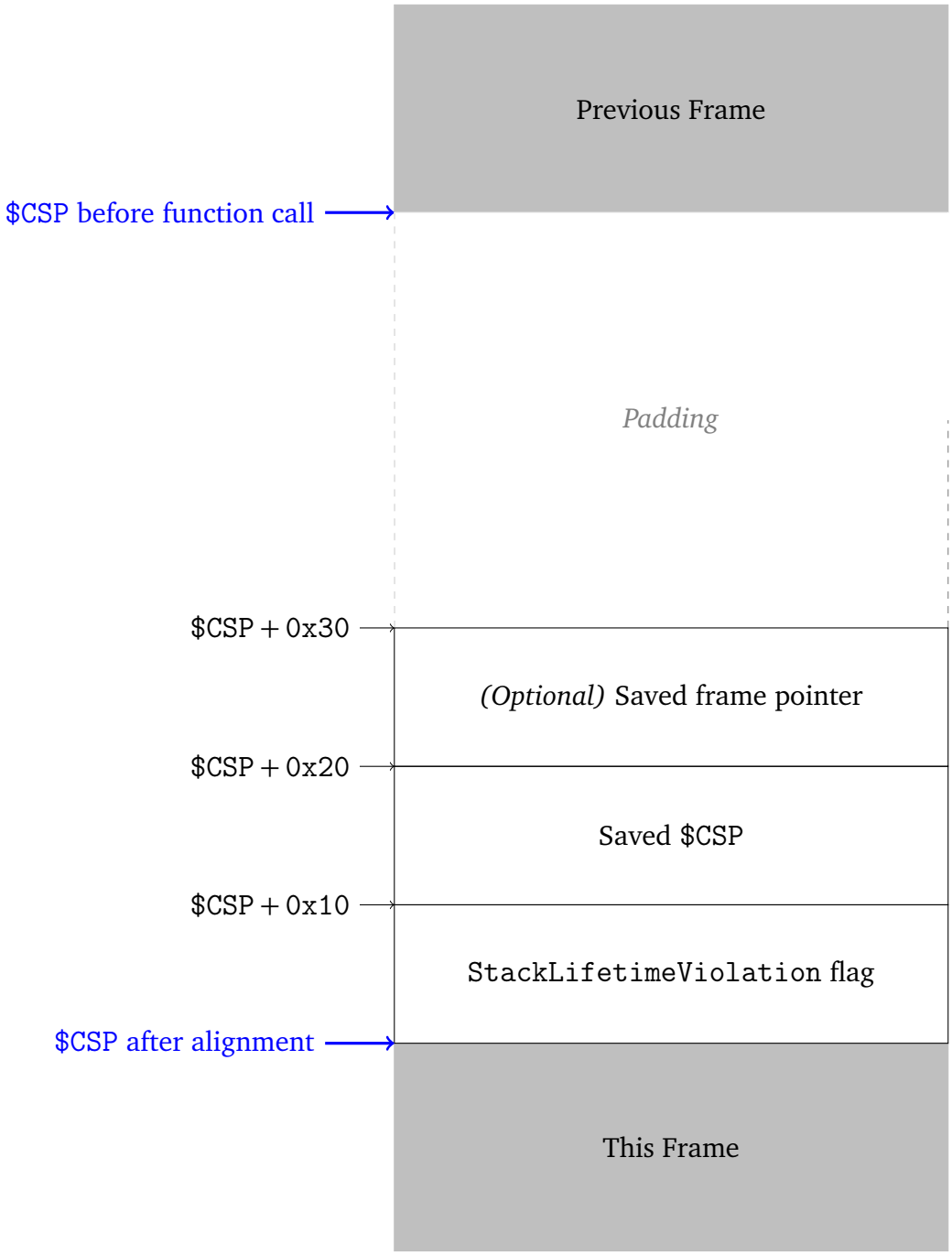


Figure 5.1: *Lowered stack-frame layout.* Saved values and the `StackLifetimeViolation` flag are placed *above* the aligned stack-pointer value to avoid detracting from the space available for local variables, which may cause a larger architectural frame size to be used and memory wasted as a result. For example, without doing this, a frame architecturally reported as being 64 bytes in size could have as few as 16 bytes of usable space. None of the saved values will ever be used directly by the body of the function, so the fact that they reside outside the aligned stack frame will not break the capability-derived lifetimes model.

specific representations of functions that have already been lowered to instructions in the target architecture's instruction set.

I created an IR-level transformation pass called `CheriCapDerivedLifetimesPass`, though implemented it in the `CodeGen` sub-component because its invocation is currently specific to the RISC-V backend and no other backend contains mappings for the intrinsics the pass inserts. Although the pass only operates on each function separately, optimisations described in Section 5.2.5 meant that implementing it as a `ModulePass` should produce better-performing code. Running the pass late during the LTO stage also meant I did not need to worry about the effects of function inlining on frame sizes.

I combined the insertions of lifetime checks and conditional revocation sweeps into a single compiler pass because the prototype I developed used a revocation sweep at the end of a function to resolve failed lifetime checks. If someone wanted failed lifetime checks to be fatal for a process, for example if they were developing a real-time system and could tolerate stricter memory-safety semantics, then they would need to separate the pass into two and disable the conditional revocation part.

Being an intentionally simple load-store architecure, LLVM IR contains only a single `store` instruction that is used to store pointer and non-pointer data alike, so scanning the entire sequence for `csc` (Capability Store Capability)-like instructions was not a possibility. Fortunately, however, LLVM IR is strongly-typed and preserves higher-level information like whether a value is of pointer-type. My pass inserts lifetime checks simply by looping over all instructions, inserting the intrinsic corresponding to `ccsc` after any store whose value being stored was a pointer.

Accessing the flag to determine whether a lifetime violation happened required a method of calculating the start of the stack frame, including for functions without a frame pointer. Fortunately, I had already prepared the `cgetframebase` instruction to accelerate this exact process. I created another intrinsic to give direct access to the stack pointer, applied `cgetframebase` to it, and issued an IR `load` instruction to read the value stored there.

5.2.4 Handling Incompatible Functions

The capability-derived lifetimes scheme, as implemented in my prototype, has some limitations in the functions it supports. Functions whose stack frames are larger than 4 KiB, or that use an `alloca()` call to allocate stack memory dynamically at runtime, are incompatible because their sizes cannot be encoded into the three bits of metadata available.

The solution I used to handle such cases was simply to issue an *unconditional* revocation

sweep at the end of the function that happens regardless of whether an unsafe store occurred. For this to work, the frame-size metadata of the stack pointer must be set to the "not a stack capability" value in the function's prologue to ensure that no lifetime check ever tries to set a nonexistent "a lifetime violation happened" flag in that frame. The unconditional revocation sweep at the end of the function is inconvenient but necessary because disabling the checks means we lose the ability to determine whether any capabilities to the frame may have escaped.

5.2.5 Optimisation: Escape Analysis

Escape analysis, as explained in Section 2.1.1, is a compiler analysis that decides whether a reference to a particular stack-resident object may leave its stack frame. If a function is provably safe at compile-time then we know that no dangerous stack lifetime violations will ever occur, meaning that we can elide the flag and conditional revocation entirely. To reduce development time, I integrated an escape analysis pass written by Lawrence Esswood as part of his work for an as-yet unpublished PhD thesis [36].

One mistake I made while initially integrating the pass, and later corrected, was to believe that a function containing no escaping locals needed no lifetime checks at all: this is not true! Demonstrating that a particular function contains no escaping locals means that no capabilities to *that function's frame* will escape and cause a `StackLifetimeViolation`, but says nothing about capabilities pointing to *other functions' frames* causing an exception during the execution of this function. Figure 5.2 shows an example of how a "safe" non-escaping function can still cause a violation to occur between capabilities pointing to other frames. Because of this, lifetime checks still need to be inserted for non-escaping functions; eliding them would require an additional data-flow analysis tracking specifically the prevalence of any escaping allocations.

Lawrence's escape-analysis pass was designed to operate during the LTO (link-time optimisation) stage of compilation so that it can track capabilities across translation-unit boundaries. Unfortunately, at the time of writing, CHERI-Clang's `-flto` flag is broken for the CHERI-RISC-V backend, so I was forced to invoke it per-translation unit rather than per-module. Its accuracy will therefore have been reduced because of the necessity of making pessimistic assumptions about the treatment of any capability passed to a callee in another translation unit. If one were to pass a capability as an argument to a function whose implementation is unknown, as would be the case if the only available information were its prototype from a header file, then there would be no guarantee against that function writing the capability to the heap, for example. Fortunately, function inlining remained safe: the lack of LTO meant there would also be no inter-translation-unit inlines applied, and

```
void outer() {
    int *xp;
    middle(&xp);
}

void middle(int **xp) {
    int y;
    inner(xp, &y);
}

void inner(int **xp, int *y) {
    *xp = y;
}
```

Figure 5.2: *A stack lifetime violation in a non-escaping function.* The stack-allocated local variable y resides in a shorter-lifetime stack frame than the pointer xp declared in `outer()`, so storing a pointer to y in xp should trigger a lifetime exception. However, the actual store does not happen until execution enters `inner()`, a function with no escaping locals. Excluding lifetime checks from stores in non-escaping functions would allow functions containing lifetime-invalid stores to succeed simply by deferring the store to a callee.

running the pass late in the target-specific code-generation phase meant that the generic intra-translation-unit inlining optimisation pass had already been applied.

5.3 Operating System Support: CheriBSD

Operating systems can largely be thought of as *services* made available to user processes. Execution enters the kernel in response to an interrupt or system call, and returns to a user process once handling is complete. The two CheriBSD kernel entry points that my capability-derived lifetimes prototype needed to interact with were the CHERI exception trap handler, for detecting and handling `StackLifetimeViolation` exceptions, and a new system call for revoking capabilities to stack frames. Other changes were largely uninteresting, such as adding the exception to an internal enum and adding case-branches to pretty-print error messages accordingly.

5.3.1 Handling `StackLifetimeViolation` Exceptions

Trap handling is an architecture-specific process because it requires intimate manipulation of register contents for context switching. CHERI-RISC-V uses status code 34 to indicate CHERI exceptions, and the default trap-handling behaviour was just to terminate any process that incurs one. I added logic inside the CHERI exception handler to check the sub-code and break out for `StackLifetimeViolation` exceptions, vectoring execution instead to a separate handler that sets the "a lifetime violation happened" flag in the appropriate stack

frame and advances the program counter to the next instruction, rather than just terminating the process as all other CHERI exception handlers do. Figure 5.3a shows graphically the control-flow path when a `StackLifetimeViolation` occurs in CheriBSD.

Adding an if-else block to a code section as critical as a trap handler might seem like an excessive performance burden, but since it only applied to the case-branch for handling CHERI exceptions, the only process it might slow down is that of terminating a process after a memory-safety fault. Adding a few nanoseconds to the amount of time required to kill a failed process is an overhead I was willing to accept.

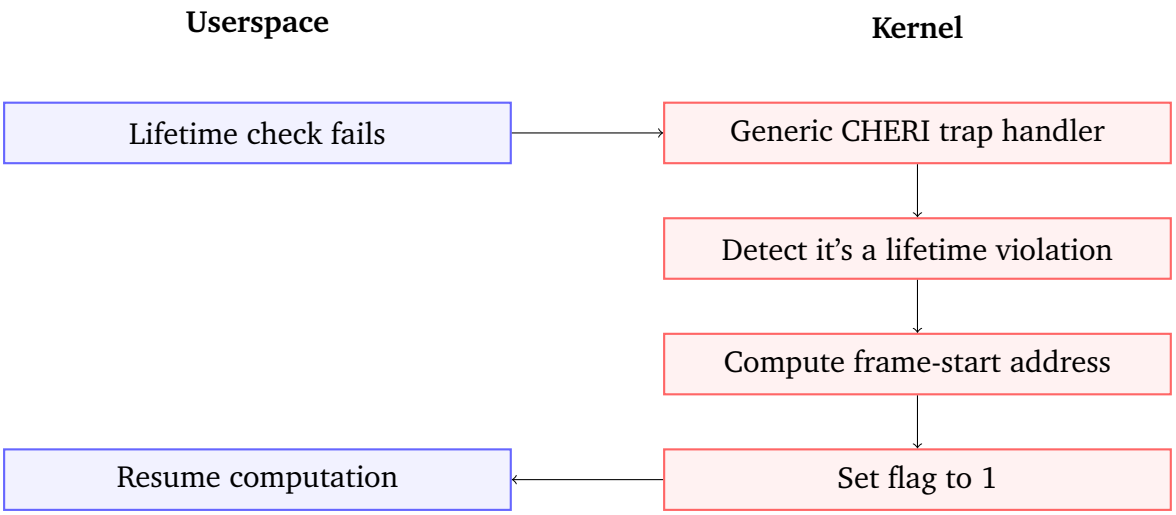
5.3.2 The `caprevoke_stack()` System Call

While I could have implemented stack-frame revocation without an additional system call, I decided to keep the userspace code simple and spare user processes the burden of needing to include the whole procedure of painting the shadow map in the epilogues of all possibly-escaping functions.

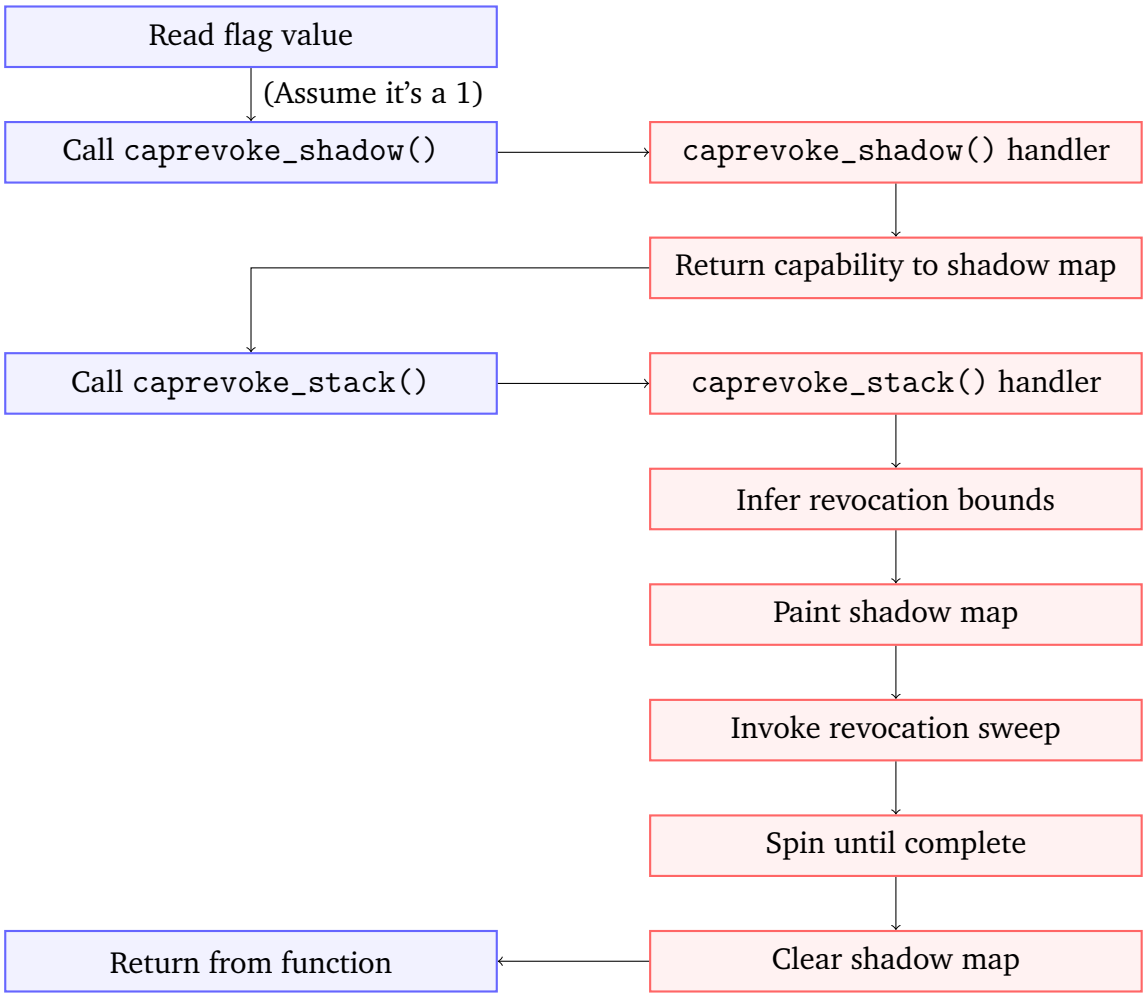
The current revocation framework exposes two system calls: `caprevoke_shadow()`, which takes a capability as an argument and returns a capability to the corresponding subset of the *shadow map*; and `caprevoke()`, which performs the actual revocation. The shadow map is a buffer that userspace code *paints* to indicate which addresses need to be revoked. I designed the new system call, `caprevoke_stack()`, to behave much like the main `caprevoke()` system call by taking the shadow-map capability as a parameter alongside a capability to the stack frame needing revocation.

To revoke all capabilities pointing to a stack frame, Clang first emits a call to `caprevoke_shadow()`, passing in the stack pointer as an argument since its bounds cover the entire stack. It then passes the resulting shadow-map capability to a subsequent call to `caprevoke_stack()`, along with the stack pointer again, to invoke the actual revocation process. In retrospect, designing `caprevoke_stack()` to obtain the shadow-map capability internally from the argument might have been a cleaner interface. I needed to add a wrapper function to CheriBSD's `libc` library to expose the system call to userspace applications.

Once inside the system call, the kernel uses the address of the supplied capability and corresponding frame-size metadata to derive the bounds of the region to be revoked. If the frame-size metadata bits are nonzero, then this means revoking just a single stack frame, the bounds of which can be derived using a binary and-masks and additions, just like are used elsewhere for lifetime-checking operations. This corresponds to a *conditional* revocation. For *unconditional* revocations, as are needed for functions whose stack frames cannot



(a) Lifetime checks



(b) Revocation sweeps

Figure 5.3: *Control-flow paths between the userspace process and CheriBSD kernel.*

be deterministically aligned in the way that capability-derived lifetimes require (see Section 5.2.4), the system call instead interprets the supplied capability as a stack pointer pointing to the beginning of the stack frame (since the call is made during the function epilogue, after the local variables have been popped). As explained in Section 5.2.4, Clang sets the frame-size metadata of capabilities pointing to incompatible stack frames to a special "not a stack capability" state to force lifetime checks not to write to a flag slot that does not exist because the frame could not be aligned. The fact that this makes it easy for the `caprevoke_stack()` system call to disambiguate the conditional and unconditional revocation cases is a convenient bonus. For unconditional revocations, it revokes all capabilities pointing anywhere between the start of the frame under revocation and the end of the stack, inferred by inspecting the passed stack pointer's address and lower bound respectively. I decided against making this behaviour the default for both cases since it would cause unneeded extra revocation work for CDL-compatible functions. Figure 5.3b shows the control-flow path for issuing revocation sweeps.

5.4 Microarchitectural Support: Toooba

Toooba [30] is a superscalar, out-of-order CHERI-RISC-V processor based on the RISC-V core with the same name developed by Bluespec [37] that is in turn based on MIT's RiscyOO core [38]. Toooba is written in Bluespec SystemVerilog, a rule-based language where hardware is described as object-oriented models [39]. I chose Toooba as the base for adding capability-derived lifetimes support because it is the highest-performance CHERI-RISC-V processor available at the time of writing.

Implementing the required ISA changes in Toooba was not substantially different to implementing them in Sail or QEMU. Being a hardware implementation, concerns about critical paths and timings were far more pertinent, especially since only the slowest path determines the overall clock frequency. The source code internally uses various tricks to reduce path lengths: for example, the decode stage simultaneously decodes all possible register and immediate operands for all possible instruction formats, packing only the needed subset into a data structure for the execute stage once the instruction format is known. Fortunately, my modifications did not extend the critical-path length, meaning the synthesis output still achieved the same 25 MHz frequency on FPGA as before.

I implemented the metadata-modification instructions (`csfs` and `cgfs`) exactly as expected. Toooba shares much of the logic for the various set and get instructions for CHERI metadata, so adding `csfs` and `cgfs` support merely entailed modifying internal "capability" data structures and inserting additional case-arms into large switch statements. Supporting

`cgetframebase` was similarly mundane, just requiring additional branches in the decode and execute stages and a direct translation of the address-computation logic.

By far the most interesting modification was adding `ccsc`, the main lifetime-checking instruction. Designing `ccsc` to mimic the operand format of `csc`, the capability-store instruction, made it easy for the compiler to insert them, but made the microarchitectural implementation more challenging because Toooba decodes and executes store-type instructions in a different location than that in which it decodes ALU-using ones. To manage the various store-type operations and their corresponding checks, Toooba decodes them into internal micro-ops and sets flags for each "check" that the operation requires, such as "the capability in the destination register is within its bounds" or "the capability is not sealed".

I avoided creating an additional internal micro-op for the lifetime check instruction by instead decoding `ccsc` instructions to "store the specified capability register in the null register" – that is, a no-op. I added another flag to indicate that lifetime checks are required for a given store-type micro-op and implemented the logic in the appropriate validation part of the execute stage. Avoiding adding another micro-op undoubtedly helped to reduce chip area overheads.

Chapter 6

Evaluation

This chapter evaluates the capability-derived lifetimes model and my prototype of it in terms of performance overheads, microarchitectural overheads, security impact and software compatibility.

All performance and microarchitectural measurements were taken using Toooba synthesised for a Xilinx VCU118 FPGA. An existing bug in CheriBSD's virtual-memory system meant I had to disable superpages to make revocation work; for fairness, I therefore disabled superpages for all baseline measurements too.

6.1 Time Overheads

As explained in Section 5.2.2, time constraints meant I was unable to implement stack-passed arguments, preventing Clang from being able to compile the SPEC benchmark suite [40]. Instead, I constructed a suite of microbenchmarks to reveal particular types of performance overhead. The three main benchmarks I used were:

1. Calling a function that does nothing and returns 1,000,000 times
2. Calling a function that performs a *safe* store 1,000,000 times
3. Calling a function that performs an *unsafe* store 1,000,000 times

Microbenchmark results are shown in Figure 6.1. Function-calling overheads are noticeable but not substantial, adding a 5.9% execution-time overhead to the microbenchmark. Importantly, the more work a function does during its execution, the more this overhead will be suppressed.

Because I implemented the changes in Toooba, a superscalar processor capable of out-of-

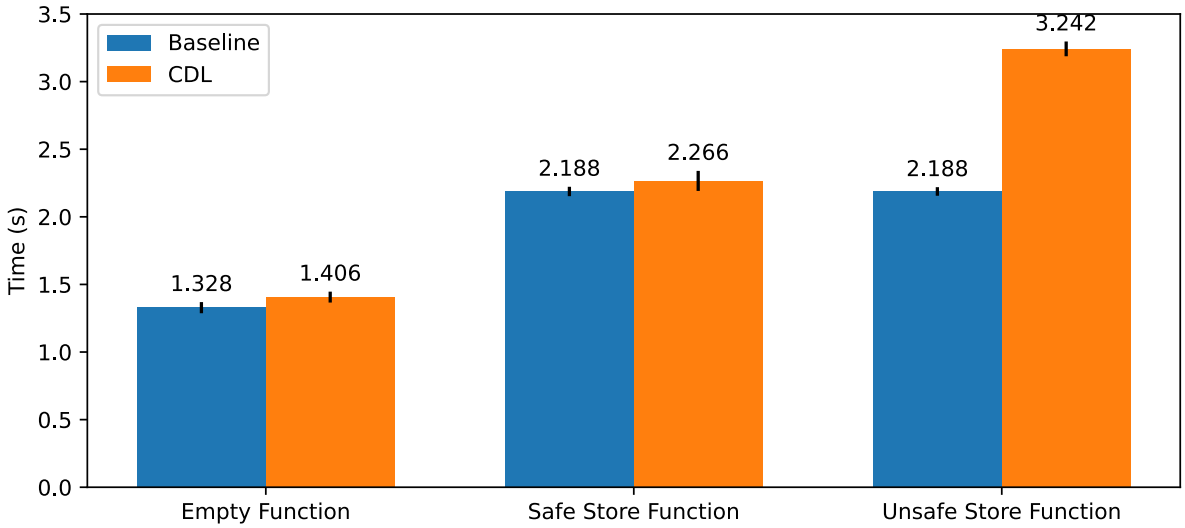


Figure 6.1: *General overheads of capability-derived lifetimes.* Bar heights indicate median values ($N = 10$) and error bars indicate standard deviations. Timing measurements were collected by recording the time at the start and end of `main()`; the store microbenchmarks therefore also incur a function-alignment time overhead compared with the baseline. See Appendix B for all microbenchmarks' source code.

order dispatch, adding `ccsc` instructions after valid stores added *no cycle-count overhead* at all, since the check could be dispatched in parallel to its store. As expected, an overhead was observed for unsafe stores, since it triggers a trap handler and a revocation sweep. Unsafe stores suffered a 48.2% overhead in the unsafe stores microbenchmark.

Figure 6.2 gives a more detailed breakdown of the overheads. I produced these numbers by compiling binaries with appropriate compiler modifications enabled or disabled, timing specifically the feature in question rather than execution of the whole microbenchmark as was the case for Figure 6.1. I found the 6.5% stack alignment overhead to be surprisingly high, and the revocation overhead of 8.6% to be surprisingly low[1].

One problem in the results is that I remain unable to explain why the unsafe CCSC instruction's time overhead of 0.5% is so low. I manually inspected the binary and can confirm the instruction is present; I even inserted print statements into the kernel and can confirm that the trap handler was being hit. The true overhead is likely higher than that, because processing an unsafe store involves a lifetime exception being thrown and context-switch into the kernel.

Overall, however, the microbenchmark results show promising performance, especially

[1]The reason for the discrepancy in revocation overheads between Figures 6.1 and 6.2 is that the unsafe stores microbenchmark endures all overheads at once, including stack alignment and the exception trap handler, while the numbers in Figure 6.2 show specifically each component on its own.

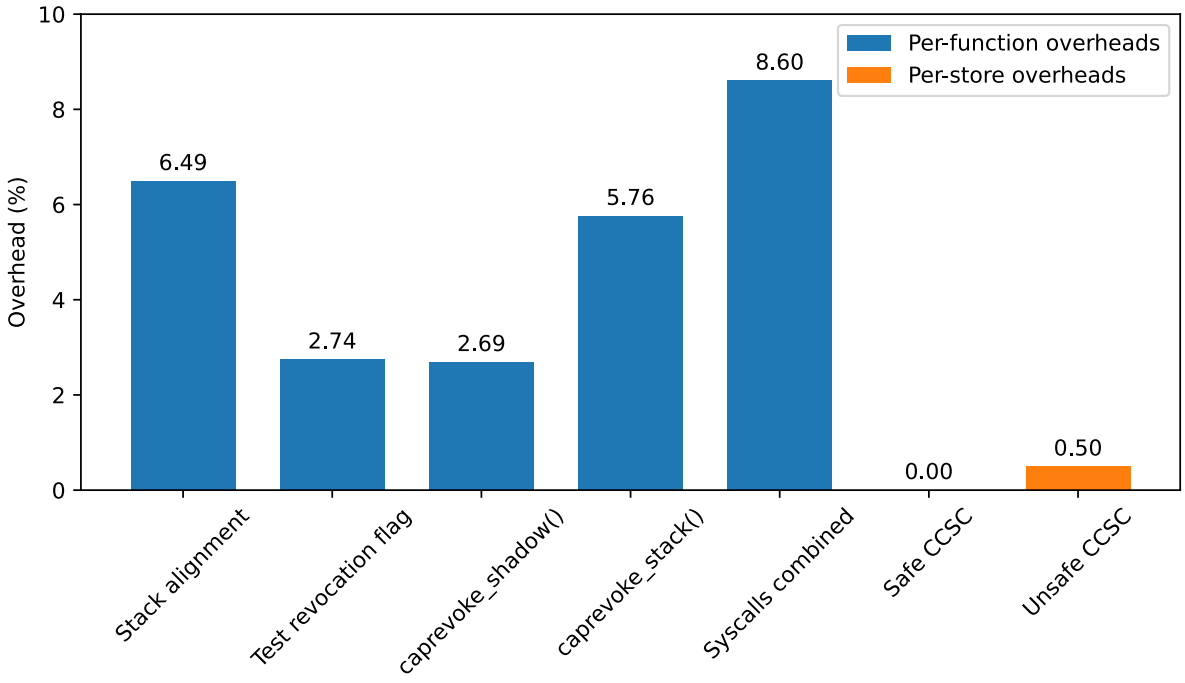


Figure 6.2: *Breakdown of capability-derived lifetime overhead sources.* Plotted values are ratios between median values of the two microbenchmark runs ($N = 10$). Timing measurements span just the sub-overhead in question, not the execution of the whole process.

given that they essentially reveal worst-case scenarios. Alignment costs and revocation sweeps are incurred per-function call: a function with a long-running while-loop would occur practically zero. Similarly, store overheads would also be amortised in functions that contain instructions other than stores, especially if wide superscalar designs allow the instruction to be executed in parallel.

6.2 Memory Overheads

Because I was unable to run the SPEC benchmark suite, I was not able to measure the true memory overhead of capability-derived lifetimes. However, we can still compute in advance that the additional memory used for a possibly-escaping stack frame is:

$$\text{memory usage with CDL} \quad \leq \quad 2 \times \text{memory usage without CDL} \; + \; 48 \text{ bytes}$$

Padding can at most double the amount of space needed for a frame, and the additional upto 48 bytes are from the saved SP, FP and the "did a violation happen" flag. Note that padding behaves unpredictably: for example, a function with a small frame that calls a function with a big frame will likely waste a lot of memory because of the increase in the

	Baseline Toooba	Toooba with CDL	Overhead
Frequency	25 MHZ	25 MHz	0%
CLB LUTs	690,255	685,621	-0.67%
CLB Registers	327,402	327,364	-0.01%
Static power draw	0.494 W	0.514 W	4.05%

Table 6.1: *PPA overheads of capability-derived lifetimes in Toooba.*

number of bits' alignment needed, while a function with a large frame that calls a function with a small frame would waste very little. To understand why padding is used *before* a function, refer back to Figure 3.2.

6.3 Microarchitectural Overheads

Outputs from the FPGA synthesis tool show that the static PPA (Power, Performance and Area) overheads range from small to negligible. I synthesised the baseline CHERI Toooba core and the version extended to support capability-derived lifetimes both for a Xilinx VCU118 FPGA: Table 6.1 shows the PPA overheads of the changes.

The Configurable Logic Block (CLB) is the main resource for implementing general-purpose combinatorial and sequential circuits in the VCU118 [41]. The number of LUTs (Look-Up Tables) and Registers therefore provide estimates of the overall increase in the amount of logic and storage used.

Adding capability-derived lifetimes support to Toooba did not affect the chip's static frequency, nor did it impose any substantial area overhead. The overall *decrease* in LUT usage is probably an artefact of an imperfect place-and-route step, since the problem being solved is NP-complete, as opposed to a true decrease in complexity; the inference we *can* make, however, is that any area overheads appear to be negligible. A static power draw increase of 4% is slightly disappointing, but the exact effect on dynamic power draw would depend on the workload, not just changes in the underlying microarchitecture.

6.4 Security Impact

For functions that are compatible with capability-derived lifetimes (frames no larger than 4 KiB and containing no dynamic stack allocations), the security provided is equivalent to those of the *capability lifetimes* scheme proposed by Tsampas *et al.* [13]. CDL emulates the semantics of capability lifetimes, throwing a hardware exception whenever a capability escapes to a potentially-longer lifetime region. Tsampas *et al.* proved that capability lifetimes

prevent the creation of a temporally-invalid stack capability: my scheme therefore does the same.

My particular prototype, using stack-frame revocation to handle incompatible functions and functions that incur a lifetime exception for any of their local variables, preserves this level of security. Revoking capabilities to a stack frame before it returns means that, assuming the revocation process is correct, capabilities to that frame *cannot exist* once the function returns.

As a result, my capability-derived lifetimes model makes *use-after-free* and *use-after-reallocation* violations become completely impossible for single-threaded programs – assuming the compiler is correct in its decisions about when to include lifetime checks. My CDL prototype successfully mitigated the exploits presented in Section 1.2.

Processes with multiple userspace threads are sadly not supported because the scheme has no way to disambiguate stack capabilities to multiple stacks. Investigating how to extend the scheme for multithreaded applications might be an interesting direction for future research.

6.5 Software Compatibility

The basic capability-derived lifetimes scheme does place considerable burdens on programs' source code: stack frames cannot exceed 4 KiB and must be statically-sized. However, if one applies a strategy such as reverting to an unconditional revocation sweep at the end of any incompatible function, one regains 100% code compatibility.

As stated previously, time constraints meant I was unable to implement stack-passed function arguments, impeding Clang's ability to compile real-world software. Every single CHERI-compatible SPEC benchmark eventually crashed, and inspections revealed that garbage accesses to stack-passed arguments were to blame. If that were fixed, then my prototype *would* in theory be compatible with *all correct C and C++ code*.

The separate lifetime-checking instruction makes capability-derived lifetimes *opt-in*, and code is interoperable with non-CDL code, making gradual roll-out across a large codebase far easier.

Chapter 7

Summary and Conclusions

CHERI is an extremely promising ISA extension for memory safety, but currently lacks protections against certain stack temporal-safety vulnerabilities. While there may not yet be a large corpus of high-profile stack temporal-safety exploits in popular software, the fact that new schemes like Cornucopia essentially eliminate use-after-reallocation bugs on the heap means we should expect to see more attention than ever before on stack memory in the pursuit of new vulnerabilities to exploit.

Capability-derived lifetimes take a simple idea – that the nature of stack memory means we can deem certain stores "safe" and others "unsafe" – and uses it to construct an efficient scheme for detecting the possible escape of pointers to local variables. Relegating the handling of lifetime violations to a software-level handler gives users the freedom to respond in a means appropriate to the application and its environment, without compromising safety guarantees.

I hope my work will open the door to many new avenues of research, and more importantly I hope that the CHERI project leads computer architecture to a safer, more reliable future. The project was challenging but extremely rewarding. Connecting so many low-level systems can be daunting, and bugs can be fiendishly difficult to solve, but capability-derived lifetimes, like CHERI itself, are an example of the power of joint hardware and software co-design. Let's hope the next century of computing is safer than the last.

Appendices

Appendix A

A Believable Vulnerability

Following is an obfuscated version of the example given in Figure 1.2. The dangerous store happens in `UserManagerUtils::createAndProcessExampleUser`. Here, the programmer made the mistake of not realising that the `UserProcessor`'s active user would be dereferenced again. This longer example was constructed to emphasise how easily such vulnerabilities can emerge when separate programmers misunderstand the usage of or guarantees provided by each others' code.

```
#include <cassert>
#include <cstring>
#include <iostream>
#include <string>

#define NAME_LENGTH 32

/**
 * Contains typical attributes for the user of some system.
 */
struct User {
    char name[NAME_LENGTH];
    // Example: uint64_t id;
};

/**
 * A stateful class that performs important actions for users.
 */
class UserProcessor {
private:
    User *activeUser;
public:
    void setActiveUser(User *user);
```

```
    User *getActiveUser();
    void doSomethingMeaningfulWithActiveUser();
};

/**
 * A collection of utilities for working with users.
 */
struct UserManagerUtils {
public:
    static void createAndProcessExampleUser(UserProcessor *processor);
    static void updateUsername(User *user);
    static bool userIsAdmin(User *user);
    static void performLogin(User *user);
};

void UserProcessor::setActiveUser(User *user) {
    activeUser = user;
}

User *UserProcessor::getActiveUser() {
    return activeUser;
}

void UserProcessor::doSomethingMeaningfulWithActiveUser() {
    std::cout << "Doing something meaningful with user: "
              << activeUser->name << std::endl;
}

void UserManagerUtils::createAndProcessExampleUser(UserProcessor *processor) {
    User newUser{"Mr Anonymous Student"};
    processor->setActiveUser(&newUser);
    processor->doSomethingMeaningfulWithActiveUser();
}

void UserManagerUtils::updateUsername(User *user) {
    // Pretend this loaded a user-controlled value, e.g. from a UI textbox or
    // from a file.
    memset(&user->name, 1, NAME_LENGTH);  // No spatial violation!
}

bool UserManagerUtils::userIsAdmin(User *user) {
    // Pretend this is looked up the from a database or something like that.
    return false;
}

void UserManagerUtils::performLogin(User *user) {
    bool isAdmin = userIsAdmin(user);
    assert(!isAdmin);
    updateUsername(user);
```

```
    if (isAdmin) {
        std::cout << "User is admin" << std::endl;
    } else {
        std::cout << "User is not an admin" << std::endl;
    }
}

int main(int argc, char **argv) {

    // Earlier:
    UserProcessor uploader;
    UserManagerUtils::createAndProcessExampleUser(&uploader);

    // Then, later on:
    User *lastUploaded = uploader.getActiveUser();
    UserManagerUtils::performLogin(lastUploaded);

    return 0;
}
```

As expected, the output from my laptop is `"User is admin"`, showing that the exploit successfully manipulated control-flow to bypass a permissions check.

Appendix B

Microbenchmarks Source Code

Function calling

```
__attribute__((noinline))
void functionToCall() {
    return;
}

int main(int argc, char **argv) {
    long int N = atol(argv[1]);
    TIME_AND_RUN_N_TIMES(functionToCall(), N);
    return 0;
}
```

Safe stores

```
typedef struct demo {
    struct demo *ptr;
} demo_t;

__attribute__((noinline))
void inner(demo_t *outer_obj) {
    demo_t inner_obj;
    inner_obj.ptr = outer_obj;
}

__attribute__((noinline))
void outer() {
    demo_t outer_obj;
```

```
    inner(&outer_obj);
}

int main(int argc, char **argv) {
    long int N = atol(argv[1]);
    TIME_AND_RUN_N_TIMES(outer(), N);
    return 0;
}
```

Unsafe stores

```
typedef struct demo {
    struct demo *ptr;
} demo_t;

__attribute__((noinline))
void inner(demo_t *outer_obj) {
    demo_t inner_obj;
    outer_obj->ptr = &inner_obj;
}

__attribute__((noinline))
void outer() {
    demo_t outer_obj;
    inner(&outer_obj);
}

int main(int argc, char **argv) {
    long int N = atol(argv[1]);
    TIME_AND_RUN_N_TIMES(outer(), N);
    return 0;
}
```

For the breakdowns in Figure 6.2 I instead timed iterations of a while-loop for the store microbenchmarks, to avoid measuring function-calling overheads.

Bibliography

[1] Catalin Cimpanu. Chrome: 70% of all security bugs are memory safety issues.

[2] Catalin Cimpanu. Microsoft: 70 percent of all security bugs are memory safety issues.

[3] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62.

[4] Kelly Jackson Higgins. The Morris Worm Turns 30.

[5] Waleed Alraddadi and Harshini Sarvotham. A Comprehensive Analysis of WannaCry: Technical Analysis, Reverse Engineering, and Motivation.

[6] Jonathan Berr. "WannaCry" ransomware attack losses could reach $4 billion.

[7] R. Watson, S. W. Moore, Peter Sewell, and P Neumann. An Introduction to CHERI.

[8] R. Watson, Alexander Richardson, Brooks Davis, John Baldwin, D. Chisnall, Jessica Clarke, N. Filardo, S. Moore, E. Napierala, Peter Sewell, and P. Neumann. CHERI C/C++ Programming Guide. 947.

[9] Alexander Richardson. Complete spatial safety for C and C++ using CHERI capabilities.

[10] N. Wesley Filardo, B. F. Gutstein, J. Woodruff, S. Ainsworth, L. Paul-Trifu, B. Davis, H. Xia, E. Tomasz Napierala, A. Richardson, J. Baldwin, D. Chisnall, J. Clarke, K. Gudka, A. Joannou, A. Theodore Markettos, A. Mazzinghi, R. M. Norton, M. Roe, P. Sewell, S. Son, T. M. Jones, S. W. Moore, P. G. Neumann, and R. N. M. Watson. Cornucopia: Temporal Safety for CHERI Heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 608–625.

[11] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, pages 545–557. Association for Computing Machinery.

[12] Chrysanthus Forcha. Object Lifetime and Storage Duration in C++.

[13] 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)*, pages 243–24312.

[14] Aleph One. Smashing The Stack For Fun And Profit.

[15] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. 2(4):20–27.

[16] Mark Dowd, John McDonald, and Justin Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Pearson Education.

[17] Security Analysis of CHERI ISA – Microsoft Security Response Center. https://msrc-blog.microsoft.com/2020/10/14/security-analysis-of-cheri-isa/.

[18] CMake. https://cmake.org/.

[19] Thomas Kotzmann and Hanspeter Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 111–120. Association for Computing Machinery.

[20] Thurston H. Y. Dang, Petros Maniatis, and David Wagner. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. pages 815–832.

[21] Günter Obiltschnig. *C++ for Safety-Critical Systems*.

[22] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler enforced temporal safety for C. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 31–40. Association for Computing Machinery.

[23] Aina Linn Georges, Armaël Guéneau, Alix Trieu, and Lars Birkedal. Toward Complete Stack Safety for Capability Machines (PriSC 2021) - POPL 2021.

[24] Grep - GNU Project - Free Software Foundation. https://www.gnu.org/software/grep/.

[25] The OCaml Programming Language. https://ocaml.org/.

[26] Clang C Language Family Frontend for LLVM.

[27] SRI-CSL/gllvm. https://github.com/SRI-CSL/gllvm.

[28] Alastair Reid. Who guards the guards? formal validation of the Arm v8-m architecture specification. 1:88:1–88:24.

[29] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS. 3:1–31.

[30] CTSRD-CHERI/Toooba. https://github.com/CTSRD-CHERI/Toooba.

[31] CTSRD-CHERI/cheribsd. https://github.com/CTSRD-CHERI/cheribsd.

[32] The FreeBSD Project. https://www.freebsd.org/.

[33] QEMU: The FAST! processor emulator. https://www.qemu.org/.

[34] Extending LLVM: Adding instructions, intrinsics, types, etc. — LLVM 12 documentation. https://llvm.org/docs/ExtendingLLVM.html.

[35] GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/.

[36] Lawrence Esswood. CheriOS: Designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor. Unpublished Ph.D Thesis.

[37] Bluespec/Toooba. https://github.com/bluespec/Toooba.

[38] Csail-csg/riscy-OOO. https://github.com/csail-csg/riscy-OOO.

[39] Oriol Arcas-Abella and Nehir Sonmez. Bluespec SystemVerilog. In Dirk Koch, Frank Hannig, and Daniel Ziener, editors, *FPGAs for Software Programmers*, pages 165–172. Springer International Publishing.

[40] SPEC - Standard Performance Evaluation Corporation. https://www.spec.org/.

[41] UltraScale Architecture Configurable Logic Block User Guide (UG574).