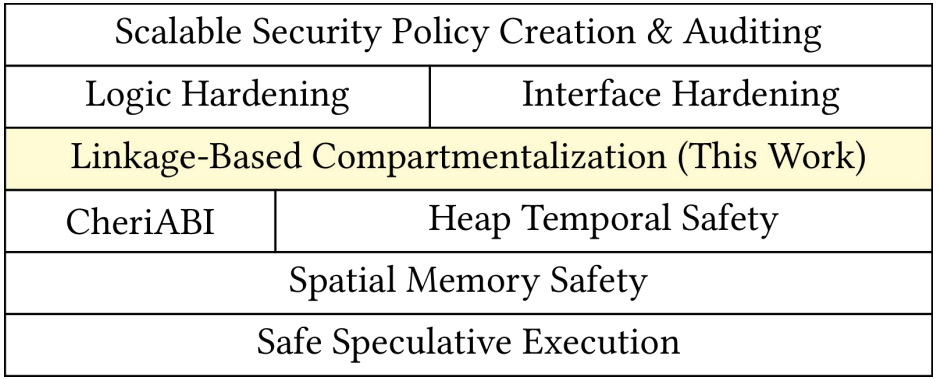
Efficient Linkage-Based Compartmentalization on CHERI

CheriTech‘26, Munich

Dapeng Gao
University of Cambridge

Background



Prior work:

- CHERI ISA enforces spatial memory safety
- CheriABI makes capability pointers first-class citizens on a POSIX operating system
- Heap temporal safety mitigates use-after-free

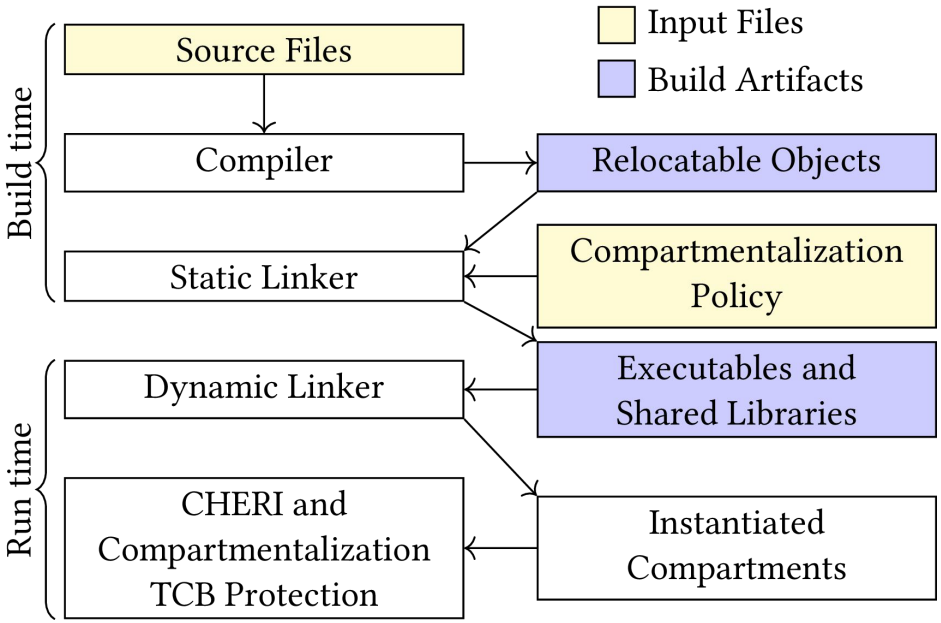
This work:

- Provides an efficient, fine-grained, and usable mechanism for compartmentalization

Future work:

- Static & dynamic analysis tools for logic and interface hardening
- Automatic policy generation and auditing

Overview



During build time, source files are…

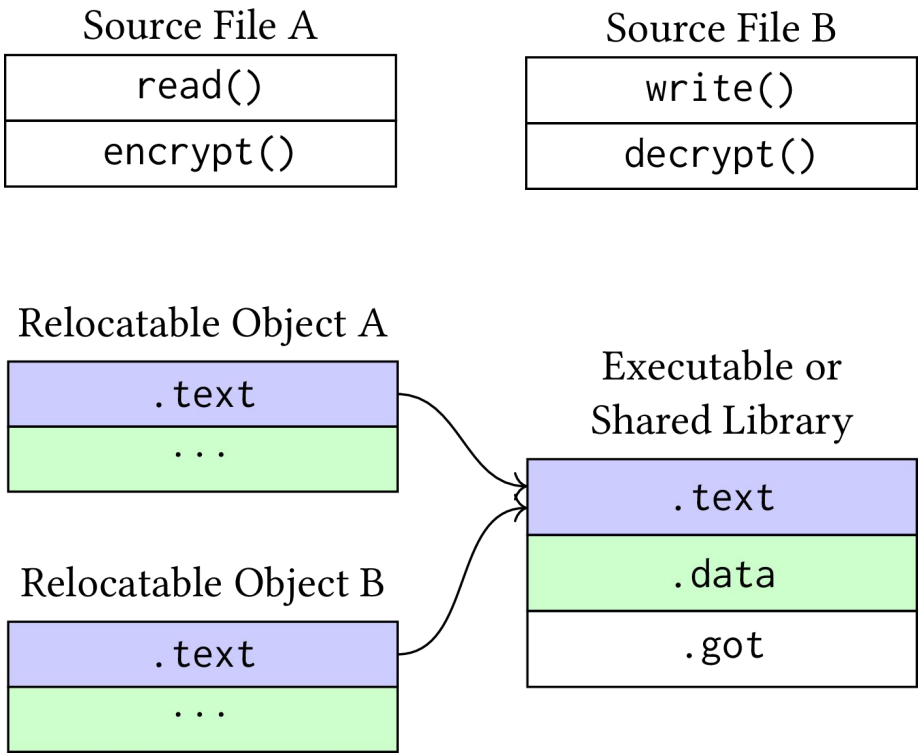
1. Compiled into relocatable objects
2. Linked into executables or shared libraries

(Optional) Compartmentalization policy dictates compartment composition and symbol accessibility in the output binaries.

During run time, these binaries are…

1. Loaded by the dynamic linker and relocated according to their policy-approved ELF section layouts and dynamic relocations
2. Executed under both standard CHERI memory safety protections and additional security guarantees provided by the compartmentalization TCB.

Compartmentalization Policy



Place symbols into compartments

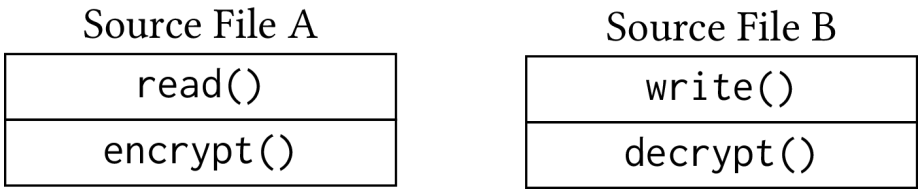
Example: policy separating io and compute

```
{"compartments": {
  "compute": {"symbols": ["encrypt", "decrypt"]},
  "io": {"symbols": ["read", "write"]}}}
```

However, conventional static linking amalgamates all source files of a library into one homogeneous binary blob.

So how does this work?

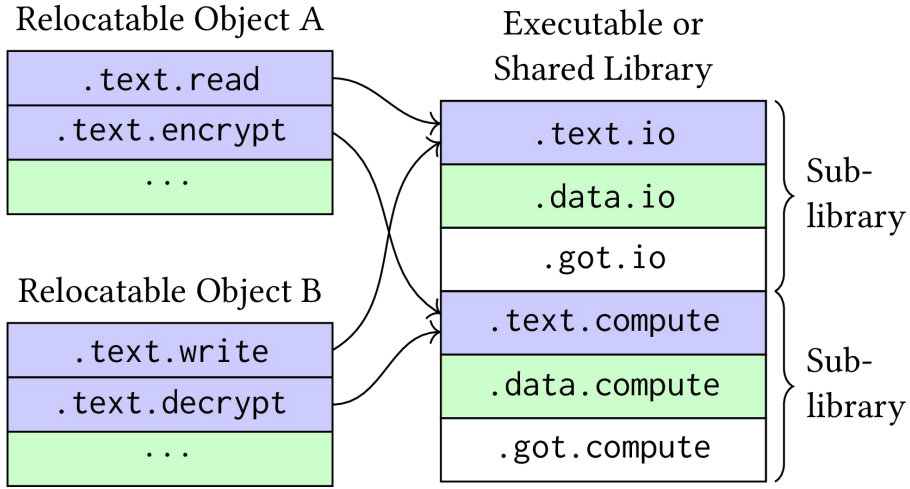
Compartmentalization Policy



Place symbols into compartments

Example: policy separating io and compute

```
{"compartments": {
  "compute": {"symbols": ["encrypt", "decrypt"]},
  "io": {"symbols": ["read", "write"]}}}
```



However, conventional static linking amalgamates all source files of a library into one homogeneous binary blob.

So how does this work?

Introducing *sub-libraries*—logically distinct libraries that live in the same library file.

Almost no disruption to build systems!

Programming Model

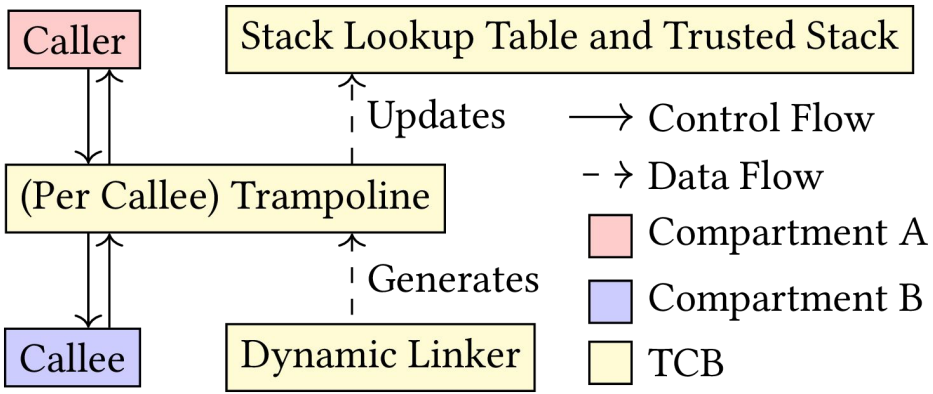
- Programs built with policy
  - Can be run without compartmentalization
- Programs built *without* a policy
  - Can be run *with* compartmentalization (using default per-library compartments)
- No extra compiler toolchain flag required
  - Apart from optionally specifying the policy

- Existing build systems can be reused
  - Easily toggle between building with and without policy
- Diagnosing compartmentalization issues is simplified
  - The same binaries also run without compartmentalization
- The same shared libraries can be used by processes with or without compartmentalization
  - Minimizing system bloat

Four Key Security Guarantees

- Stack Safety
  - No compartment has access to stack frames created by other compartments
- Register Integrity
  - Callee-saved registers are restored to their initial values after a callee compartment returns
- Register Confidentiality
  - Registers not used for holding arguments or return values are cleared
- Control Flow Integrity
  - Control flow is “well-bracketed”
  - Impossible, for example, for a callee compartment to stash a return pointer and reuse it later to enter another compartment unexpectedly

Trampolines Interpose Compartment Transitions



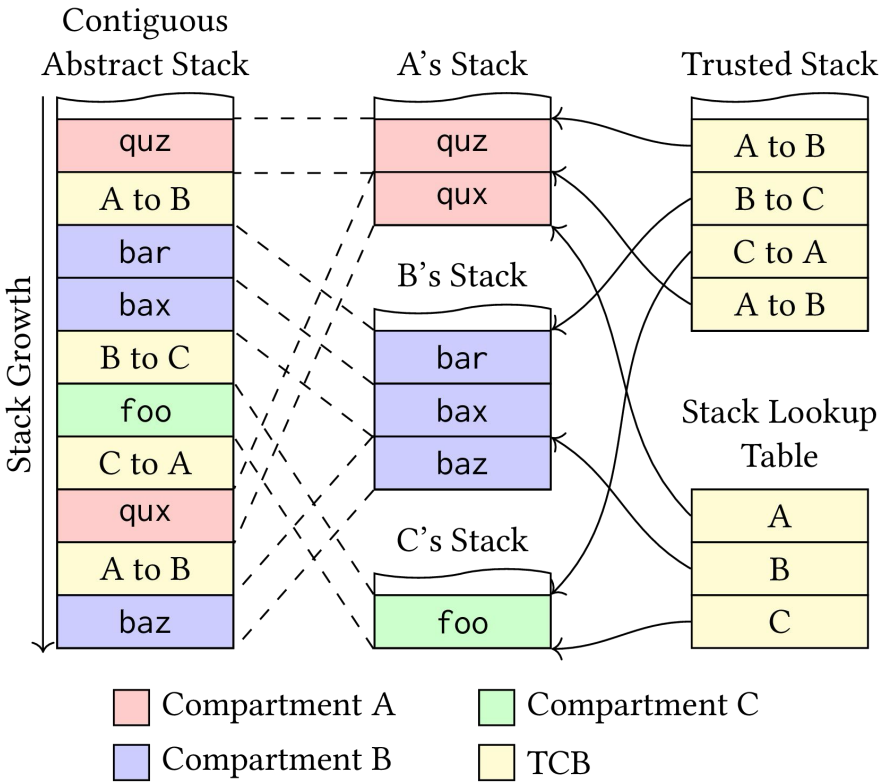
Basic facts:

- Managed by the TCB
- One instance per callee
- ~100 instructions long
- Packed in dynamically-mapped memory

What does a trampoline do?

- Switch the execution stack
- Save callee-saved argument registers
- Clear unused argument/return value registers
- Record the control flow
- Jump to the callee!
  - Or return to the caller

Backbone: Trusted Stack and Stack Lookup Table



Each compartment now has its own stack, but the programming model is a contiguous abstract stack

The *Trusted Stack* and *Stack Lookup Table* help to reconstruct this continuous abstract stack

Example:

- Calls through `bar`, `bax`, `foo`, `qux`, and `baz`
- Each compartment's stack has a few frames
- The Trusted Stack holds 'trusted frames' recording compartment transitions
- The Stack Lookup Table records, per stack, the analogy of a frame pointer
- The contiguous abstract stack changes dynamically as the program executes

Implementation and Evaluation

Addressing compatibility challenges:

- Signals
- Tail-calls
- setjmp/longjmp
- C++ exceptions
- System calls
- Thread-local storage

Case studies:

- Compartmentalising tcpdump
- Adapting the V8 JavaScript engine for Chromium
  - ~2MLoc
  - Has JIT compilation and garbage collection
  - Changed ~300LoC to work with compartmentalization

Benchmarks:

- Compartment transition micro-benchmark
- Macro-benchmark
  - FFmpeg
  - SPDK
  - gRPC

Case Study: tcpdump

```
#0  http_print
#1  <print-ip> -> <print-http>
#2  ip_print
#3  <print-ether> -> <print-ip>
#4  ethertype_print
#5  ether_common_print
#6  ether_print
#7  ether_if_print
#8  <netdissect> -> <print-ether>
#9  pretty_print_packet
#10 <tcpdump> -> <netdissect>
#11 print_packet
#12 <pcap> -> <tcpdump>
#13 pcapint_offline_read
#14 pcap_loop
#15 <tcpdump> -> <pcap>
#16 main
```

Background:

- Prints each protocol layer of packets
- Each layer printed by a dedicated *dissector*
- Next layer forwarded to the next dissector
- 150+ dissectors in total

We put each dissector in a separate compartment

- Resulting in 180 compartments

We feed the program an HTTP packet and use compartment-aware GDB to print a stack trace

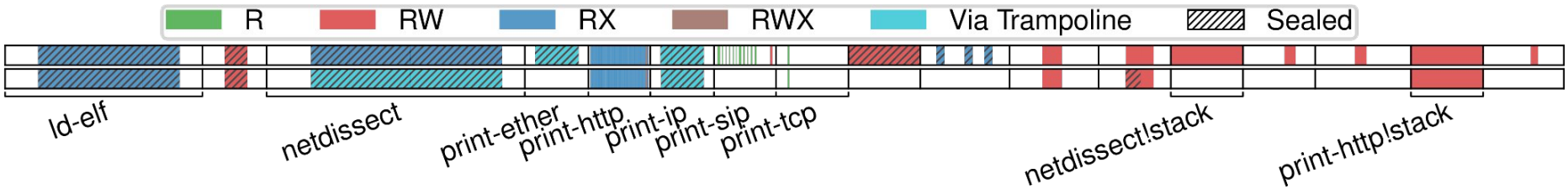
- Can identify the packet structure from the sequence of compartment transitions!

Case Study: tcpdump

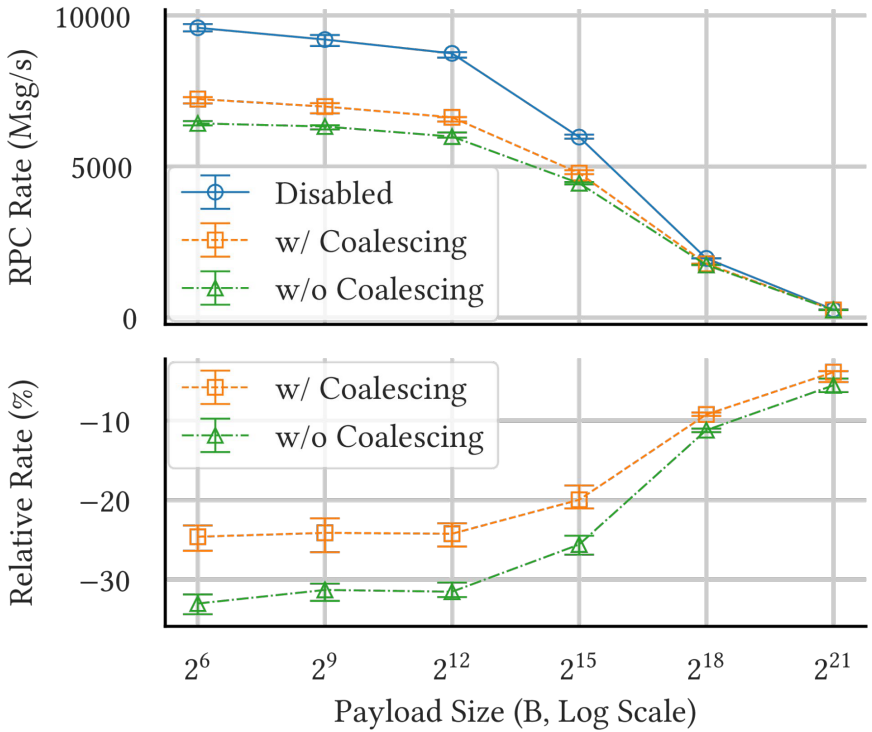
CheriTree: a compartment visualization tool

- Traverses the capability graph at any given point in execution
- Maps out reachable memory as intervals on the address-space “spectrum”
  - Colored by the union of the permissions of all capabilities covering each interval
- Traversal is exact because all valid capabilities can be precisely identified in memory by their tags, and their bounds enable an exhaustive graph-search

Upper plot: Catching a leak via a `jmp_buf`; Lower plot: After fixing it



Macro-benchmark: gRPC



High-performance C/C++ RPC framework

- Many distinct but tightly coupled libraries that belong in the same security domain
  - Bloats the numbers of compartment transitions
- Extensive use of C++ virtual functions
  - Causes many compartment self-transitions

Our solutions for better scalability:

- Allow policy to coalesce groups of libraries into the same compartment at run-time
  - Reducing run-time compartments from 82 to 18
- Add fast paths to trampolines to detect self-transitions and skip them
  - Reducing number of transitions by 2/3

Conclusion

We demonstrate that, underpinned by CHERI memory protection, compartmentalization can be significantly more efficient, fine-grained, and usable

- User-friendly policy framework
  - Just pass a JSON file to the static linker—no further changes needed
- Scales to hundreds of compartments per process
  - And 10K+ compartments for the whole system
- Compartment transitions much closer to the speed of function calls
  - And just as transparent as function calls
- Integrated debugging and visualization tools
  - How did I reach this compartment? What memory can I transitively reach?

Upcoming Publication in ACM CCS 2026



Efficient Linkage-Based Compartmentalization on CHERI

Dapeng Gao
University of Cambridge
Cambridge, UK
dapeng.gao@cl.cam.ac.uk

John Baldwin
Ararat River Consulting
Ashland, VA, USA
john@ararat river.co

Jessica Clarke
University of Cambridge
Cambridge, UK
jessica.clarke@cl.cam.ac.uk

Nicholas C. Connolly
Arm Limited
Cambridge, UK
nick.connolly@arm.com

Brooks Davis*
Capabilities Limited
Llanwrda, UK
brooks@capabilitieslimited.co.uk

Franz A. Fuchs
University of Cambridge
Cambridge, UK
franz.fuchs@cl.cam.ac.uk

Alfredo Mazzinghi†
University of Cambridge
Cambridge, UK
alfredo.mazzinghi@cl.cam.ac.uk

Daniel Moghimi
Google
Mountain View, CA, USA
danielmm@google.com

Peter Rugg†
University of Cambridge
Cambridge, UK
peter.rugg@cl.cam.ac.uk

Domagoj Stolfa‡
University of Cambridge
Cambridge, UK
domagoj.stolfa@cl.cam.ac.uk

Konrad Witaszczyk
University of Cambridge
Cambridge, UK
konrad.witaszczyk@cl.cam.ac.uk

Simon W. Moore
University of Cambridge
Cambridge, UK
simon.moore@cl.cam.ac.uk

Robert N. M. Watson†
University of Cambridge
Cambridge, UK
robert.watson@cl.cam.ac.uk

Abstract

We present an efficient *linkage-based* model for in-process compartmentalization built on CHERI memory safety, which enables fine-grained compartmentalization of the entire UNIX user-space, scaling to 10K+ compartments on desktop systems. The model's "push-button" compartmentalization along existing library boundaries regularly hosts 500+ compartments per process for large applications such as Chromium, far exceeding the number of concurrently available protection domains supported by other mechanisms (e.g., up to 16 for Intel MPK). Custom policies can further subdivide libraries. Of the thousands of C/C++ programs tested, only the V8 JavaScript engine required source-level adaptation (<300 lines of changed code concerning garbage collection and JIT compilation).

We implement the model for CHERI-extended versions of Armv8-A and RISC-V through support in the compiler toolchain and operating system. Case studies illustrate the smooth delegation of memory between compartments, compartment-aware debugging and visualization, as well as extensibility to a complex managed language runtime, demonstrating the benefits of our single-address-space model. We evaluate using multiple processors, including Arm's superscalar Morello and, notably, the first commercial CHERI-enabled RISC-V application core—Codasip's in-order dual-issue X730.

CCS Concepts

• **Security and privacy** → **Software and application security**; **Systems security**; *Malware and its mitigation.*

Keywords

Security; Memory Safety; CHERI; Compartmentalization; Linking

ACM Reference Format:

Dapeng Gao, John Baldwin, Jessica Clarke, Nicholas C. Connolly, Brooks Davis, Franz A. Fuchs, Alfredo Mazzinghi, Daniel Moghimi, Peter Rugg, Domagoj Stolfa, Konrad Witaszczyk, Simon W. Moore, and Robert N. M. Watson. 2026. Efficient Linkage-Based Compartmentalization on CHERI. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3846528

* Also with University of Cambridge.
† Also with Capabilities Limited.
‡ Also with NSB Associates.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

CCS '26, The Hague, Netherlands

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2871-9/2026/11
https://doi.org/10.1145/3830454.3846528

1 Introduction

Compartmentalization solutions are fundamentally constrained by the available architectural primitives. Techniques designed for commodity hardware often rely on features that were not originally intended for security, leading to programming models that

Co-authored by Dapeng Gao, John Baldwin, Jessica Clarke, Nicholas C. Connolly, Brooks Davis, Franz A. Fuchs, Alfredo Mazzinghi, Daniel Moghimi, Peter Rugg, Domagoj Stolfa, Konrad Witaszczyk, Simon W. Moore, Robert N. M. Watson

Pre-print available on arXiv: https://arxiv.org/abs/2609.36731