Poster: A Systematic Review of Migration-Relevant Software Idioms and Their Interactions with Capability Systems

Fatih Durmaz
CISPA Helmholtz Center for Information Security
fatih.durmaz@cispa.de

Sven Bugiel
CISPA Helmholtz Center for Information Security
bugiel@cispa.de

***Abstract*—Software compartmentalization reduces vulnerabilities' impact by decomposing software into isolated, least-privilege components. Capability systems support this goal by replacing ambient authority with explicit, delegable rights and fine-grained protection. However, legacy software was usually not written for such models, so migration requires simultaneous reasoning about correctness, security, performance, and compatibility. Idioms that are harmless on conventional systems may become problematic once compartmentalization introduces stronger protection semantics. To date, there is no systematic overview of software idioms that become problematic under different compartmentalization semantics.**

To fill this gap, we plan to collect problematic idioms and patterns, with their corresponding compartmentalization semantics, and create a taxonomy. This poster presents our agenda for systematically identifying migration-relevant software idioms and their interactions with capability systems. We aim to build a bidirectional taxonomy linking recurring idioms, root causes, and interacting capability primitives. By synthesizing evidence from publications, repositories, and forums, the study will inform future compartmentalization systems and migration tools.

***Index Terms*—Compartmentalization, capability systems, software security**

I. Problem and Motivation

Software vulnerabilities remain a persistent, evolving problem, and disclosure processes can create periods in which defenders lag behind attackers [1]. Least-privilege design therefore remains a key strategy to reducing compromise impact. Software compartmentalization implements this principle by decomposing systems into isolated components, each granted only required privileges [2]. Instead of giving all code the same ambient authority, it introduces boundaries so that one compromised compartment need not compromise the whole system [3]–[5].

Although used in practice [6], [7], compartmentalization is still not an established engineering practice. Most existing systems were not built with compartmentalization—and even less object capabilities—in mind, while research systems that support it often require substantial manual effort to adapt legacy software [3]. Adapting existing software to compartmentalized designs forces developers to balance correctness, security, complexity, and performance. Existing tools are mostly semi-automated, rely on source-code annotations or high-level policy input, and still struggle with difficult conversions and real-world edge cases [3], [5].

Prior work shows that part of the difficulty in migrating existing software to compartmentalized systems stems from software idioms that are harmless under conventional architectures but become problematic under stronger protection semantics. Examples include pointer-related assumptions [8], [9], large monolithic functions [10], nested memory allocations [11], and function pointers or nested call structures that complicate partitioning [12].

We target object capability systems because they support compartmentalization by replacing ambient authority with explicit, delegable rights and fine-grained protection domains [9], [13], [14]. The difficulty is that capability systems represent and enforce capabilities differently. For example, In CHERI, the pointer value is itself a capability carrying protection metadata; in RV-CURE, the pointer remains a conventional machine word, while protection metadata is stored separately and reached through a tag in unused pointer bits. Thus, the same software idiom can have different consequences under the two models. CHERI capabilities are wider than ordinary integer pointers and include bounds, permissions, and a validity tag, so code assuming pointers fit conventional integer types may need changes (Listing 1). RV-CURE uses unused pointer bits and therefore avoids this pointer-size mismatch [15].

```c
#include <stdint.h>
uint64_t raw = (uint64_t)p; // - Legacy assumes
void *q = (void *)raw;  // ptr fits in 64 bits -
uintptr_t raw = (uintptr_t)p; // - CHERI preserve
void *q = (void *)raw; // ptr provenance -
```

Listing 1: Pointer-to-integer round trip in legacy C and CHERI-aware C.

What is still missing is a systematic account of migration-relevant software idioms, their underlying root causes, and the capability primitives with which they interact. Existing empirical studies and porting reports provide important evidence for this gap, but they usually describe blockers within a specific system, platform, or case study [16]. In this work, such reports are treated as input evidence. The contribution is a cross-system synthesis that maps recurring idioms to their root causes and to the capability-system primitives that make them difficult to migrate. This gap motivates the following

research questions:

RQ1. Which idioms across software stacks recur as obstacles during migration to capability systems?

RQ2. How do capability-system primitives shape these obstacles, and in which software-capability-system combinations is automation realistic?

The expected outcome of this systematization is a taxonomy that connects the characteristics of capability systems with the techniques and primitives used to realize them, and with the software idioms that become difficult to migrate under those design choices. The first part of the taxonomy will provide a cross-matching table between capability-system characteristics, their enabling primitives or mechanisms, and the recurring difficult idiom patterns observed in the collected corpus. Where space allows, representative examples will be included for the most prominent cases. The second part of the taxonomy will group the identified idioms by similarity and compare their migration difficulty using a preliminary set of criteria. These criteria are intended as a working definition and may evolve as the taxonomy is refined.

On one hand, this taxonomy can help characterize which classes of software are likely to be difficult to migrate, which idioms recur across software stacks, and where automation is realistic. On the other hand, it can help characterize capability primitives themselves in terms of the migration obstacles, incompatibilities, and adaptation burdens. In turn, this can guide both the design of future compartmentalization systems and auto-compartmentalization tools.

II. Approach

Our planned work has three stages: corpus construction, evidence extraction, and creating the taxonomy.

a) Corpus construction: We construct an evidence corpus from three source types. The first is academic publications and technical reports, which describe capability-system designs, compartmentalization mechanisms, and known migration limitations, and may therefore mention problematic idioms. For example, Gudka et al. [5] report that "The dlopen() call will fail when in the sandbox," illustrating how lazy initialization and dynamic library loading can become compartmentalization obstacles. For this source type, we handpicked 36 papers and technical reports as a seed covering different capability-system characteristics.

We do not start from a fixed list of idioms or restrict the study to a single application family. Instead, idioms are treated as open-ended findings to be discovered during evidence extraction. Our initial seed set is therefore selected to cover a diverse range of capability-system characteristics, so that the resulting taxonomy can capture how different protection primitives and semantics expose different migration obstacles. Starting from the seed papers, we applied two rounds of backward and forward snowballing [17] using OpenAlex [18] to retrieve publication metadata and citation graphs, yielding 12,621 papers. Keyword filtering removed clearly irrelevant papers and left 7,525 papers. We will next apply context-aware

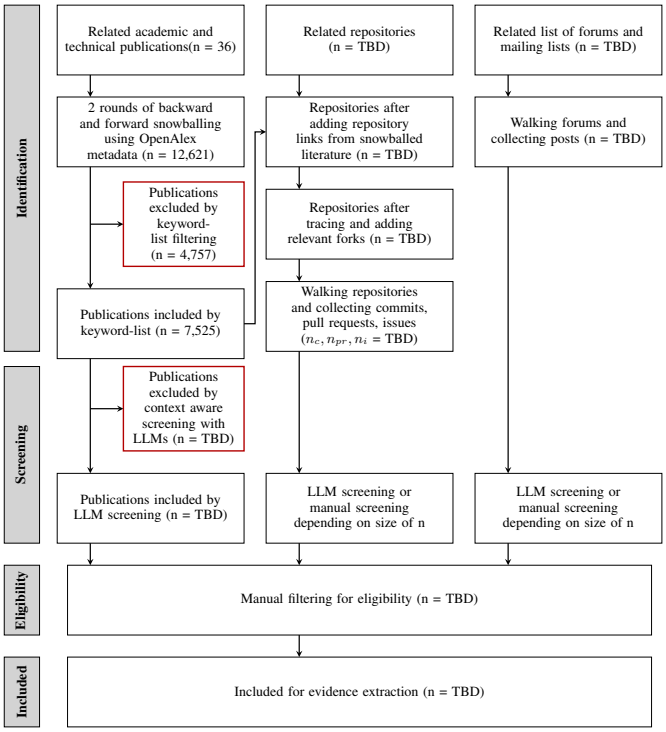

Fig. 1. PRISMA Flow Diagram for our Corpus Construction (§II-0a) and Evidence Extraction (§II-0b)

LLM screening to titles and abstracts, then manually decide whether the remaining resources contain relevant idioms.

Challenge 1: Evidence Identification

Relevant idioms may appear in many parts of a publication, from evaluation to future work, and not necessarily as focal point of the publication.

The second source type consists of open-source repositories, their commit histories, and issue trackers of software ported to capability systems. These provide concrete examples of porting problems. For example, one CHERI BSD issue [19] points to an `INT2PTR` cast, illustrating how repository discussions can expose migration failures such as pointer-integer conversions that violate CHERI's capability semantics [20]. Starting from the repositories of capability systems, we will collect pull requests, issues, and commit messages. Depending on the corpus size, we will apply LLM screening to decide whether the collected resources are relevant. We will then apply manual filtering to determine eligibility.

Challenge 2: Repository Selection and False Positives

Not all capability-system repositories, provide useful migration evidence; screening must distinguish idiom constraints from ordinary engineering noise.

The third source type consists of forums, mailing lists, and other developer-discussion platforms, which can reveal practical problems and informal workarounds absent from papers and documentation. For example, one FreeBSD mailing-list post [21] reports: "Do not try to build `cloudabi32` for `pc98`. Should unbreak tinderbox." Even brief build reports

like this can expose capability-related integration problems that are not written up elsewhere . We start from selected capability-system and systems-software discussion venues. Projects have to be mature enough to have active forums or mailing lists. For such projects from the handpicked initial list, we will scrape these venues to collect posts. Similar to the repositories, we might apply LLM screening if the corpus size explodes. Then we will manually inspect the remaining posts for eligibility.

Challenge 3: Terminology Mismatch

Related migration problems and methodologies may be described using different terminology across papers, repositories, discussions, and toolchains; therefore, keyword-based search alone may miss relevant evidence.

b) Evidence extraction: We will extract concrete observations that indicate difficult idioms. In papers, these observations typically appear in porting reports, case studies, evaluation sections, and discussions of limitations. We will look for descriptions of source-code changes, compatibility problems, failed assumptions, or constructs that required special handling under a capability system. For repositories, we will inspect commits, pull requests, issues, and code comments related to capability-system support. Developer discussions will be treated similarly: we will extract reports of errors, workarounds, and explanations that connect a concrete programming pattern to a capability-system constraint. Each extracted item will record the source, the affected capability system, the observed idiom, and the reason it appears to be difficult. When available, we will also record the patch or workaround used by developers. The result of this stage will be a set of evidence-backed candidate idioms that will later be merged, refined, and categorized.

c) Creating taxonomy: The taxonomy will be based on two dimensions in our collected data. The first dimension is the characteristics of difficult idioms. We will group idioms based on common properties, such as the assumptions they rely on, the source-code constructs involved, and the migration difficulties they create. We expect many idioms to come from C and C++, because many capability systems target low-level systems software. However, we do not restrict the taxonomy to these languages. Some systems target managed languages and use language-level mechanisms for compartmentalization. We will include such idioms when they are clearly related to migration difficulty. Language will be recorded as an attribute of each idiom. Language-specific patterns will form separate categories when needed.

The second dimension is the characteristics of capability systems. For this dimension, we will use characteristics identified by prior work [3] to group capability systems according to their primitives and protection semantics. We will then examine the overlaps between both dimensions and use them to define taxonomy categories. This will help show which idioms are hard to migrate under which capability based compartmentalization systems. Eventually, this taxonomy can guide the future compartmentalization system design and prioritization of practical migration and auto-compartmentalization tools

III. Current Project State and Poster Takeaway

The project is currently in the corpus-construction stage. We have identified the main source types for the study, selected an initial set of capability and compartmentalization systems from the literature, and implemented tooling to collect publication metadata and citation graphs. Using this tooling, we have begun building a publication corpus through backward and forward snowballing. As an initial result, we derived a keyword filter from capability- and compartmentalization-related terms. Full-text processing, systematic screening, evidence extraction, and taxonomy construction, have not yet been completed.

This early stage makes the poster especially timely, because feedback from researchers can still shape the study before its methodology and scope are fixed. Discussions with the community can help us identify which types of evidence are most useful and which aspects of migration difficulty deserve closer attention. In turn, this feedback can improve the resulting taxonomy and make the study more useful for future work on migration support and automated compartmentalization.

References

[1] P. Rotella, "Software security vulnerabilities: baselining and benchmarking," in *SEAD-ICSE*, 2018.

[2] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, 1975.

[3] H. Lefeuvre *et al.*, "Sok: Software compartmentalization," in *IEEE Symposium on Security and Privacy*, 2025.

[4] R. Khan, K. Ghanem, and F. Coffele, "Digital security by design: A review of combined hardware-software-based cybersecurity with compartmentalization," in *TechDefense*, 2023.

[5] K. Gudka *et al.*, "Clean application compartmentalization with soaap," in *ACM CCS*, 2015.

[6] Development/PrivilegeSeparation - Wireshark Wiki. [Online]. Available: https://wiki.wireshark.org/Development/PrivilegeSeparation

[7] Privilege Separated OpenSSH. [Online]. Available: http://www.citi.umich.edu/u/provos/ssh/privsep.html

[8] D. Chisnall *et al.*, "Beyond the pdp-11: Architectural support for a memory-safe c abstract machine," in *ASPLOS*, 2015.

[9] R. N. M. Watson *et al.*, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *IEEE S&P*, 2015.

[10] Y. Wu *et al.*, "Automatically partition software into least privilege components using dynamic data dependency analysis," in *ASE*, 2013.

[11] T. Palit *et al.*, "Dynpta: Combining static and dynamic analysis for practical selective data protection," in *IEEE S&P*, 2021.

[12] J. Lind *et al.*, "Glamdring: Automatic Application Partitioning for Intel SGX," in *USENIX ATC*, 2017.

[13] R. N. Watson *et al.*, "Capsicum: Practical capabilities for UNIX," in *USENIX Security*, 2010.

[14] A. Dawoud and S. Bugiel, "Droidcap: Os support for capability-based permissions in android," in *NDSS*, 2019.

[15] Y. Kim *et al.*, "Rv-cure: A risc-v capability architecture for full memory safety," *IEEE Transactions on Computers*, 2025.

[16] S. Faily. Biting the CHERI bullet: Blockers, Enablers and Security Implications of CHERI in Defence.

[17] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *EASE*, 2014.

[18] J. Priem, H. Piwowar, and R. Orr, "OpenAlex: A fully-open index of scholarly works, authors, venues, institutions, and concepts," 2022.

[19] Cheribsd. Official project website. [Online]. Available: https://www.CHERIbsd.org/

[20] gcjenkinson, "lang/perl5.32: Capability tag fault," 2022. [Online]. Available: https://github.com/CTSRD-CHERI/CHERIbsd-ports/issues/75

[21] jenkins-admin, "Freebsd build #1503 failure," 2016, freebsd-current mailing list.