SoK: Software Compartmentalization

Hugo Lefeuvre[†1], Nathan Dautenhahn[‡1], David Chisnall[∞], Pierre Olivier[▽]
[†]The University of British Columbia, [‡]Serenitix, [∞]SCI Semiconductor, [▽]The University of Manchester

***Abstract*—Decomposing large systems into smaller components with limited privileges has long been recognized as an effective means to minimize the impact of exploits. Despite historical roots, demonstrated benefits, and a plethora of research efforts in academia and industry, the compartmentalization of software is still not a mainstream practice. This paper investigates why, and how this status quo can be improved. Noting that existing approaches are fraught with inconsistencies in terminology and analytical methods, we propose a unified model for the systematic analysis, comparison, and directing of compartmentalization approaches. We use this model to review 211 research efforts and analyze 61 mainstream compartmentalized systems, confronting them to understand the limitations of both research and production works. Among others, our findings reveal that mainstream efforts largely rely on manual methods, custom abstractions, and legacy mechanisms, poles apart from recent research. We conclude with recommendations: compartmentalization should be solved holistically; progress is needed towards simplifying the definition of compartmentalization policies; towards better challenging our threat models in the light of confused deputies and hardware limitations; as well as towards bridging the gaps we pinpoint between research and mainstream needs. This paper not only maps the historical and current landscape of compartmentalization, but also sets forth a framework to foster their evolution and adoption.**

1. Introduction

Despite decades of effort, vulnerabilities still plague software, and thwarting them remains a game of cat and mouse. The *principle of least privilege* (PoLP) [209] is the last line of defense when protections fail or when flaws are unknown. By granting each entity only the privileges needed, the PoLP ensures that a compromise of one part will not imply that of the whole. *Software compartmentalization* is a prominent implementation of the PoLP, in which developers divide a large program into smaller, lesser privileged components, to reduce the impact of potential security breaches.

Software compartmentalization inherits from a large body of work, starting with processes [93]; including OS models such as microkernels [106, 126, 150, 164, 258], security and separation kernels [56, 58, 205, 206, 247], or capability OSes [83, 125]; all the way to fine-grain application compartmentalization in the 2000s [147, 181, 232, 238] following qmail [68], Postfix [121], or OpenSSH [198]. Its promises are plenty: containing memory safety issues [72, 202], untrusted third parties [54, 181], or unsafe parts of safe languages [62, 149, 174, 201, 228], isolating cryptographic secrets [167, 171, 232] or shadow stacks [74], thwarting supply-chain attacks [111, 235] and side-channels [137, 176, 182], or providing fault resilience [164, 184].

Despite longstanding recognition within the academic sphere and proven effectiveness in seminal industry projects, the adoption of compartmentalization techniques in mainstream software remains inconsistent: compartmentalizing is still far from being a common engineering practice. Take as example the isolation of cryptographic secrets, a practice long advocated by the community [65, 70, 73, 87, 115–117, 140, 158, 165, 167, 211, 226, 232] but without viable adoption by leading cryptography libraries. At a time of growing threats, we are missing out on the security benefits compartmentalization can bring. This paper investigates the reasons behind this status quo, and how to improve it.

Research speculated that this situation is due to a lack of automation [65, 253], limitations of mechanisms [190, 259], excessive performance overheads [188, 232], a lack of strong security guarantees [129, 160], among others. All are likely part of the problem, and through decades of research progress was made on every front. Today the community lacks a global overview of this progress. Designing for or retrofitting compartmentalization is often hampered by inconsistencies in the understanding and application of its concepts. Existing models and terminologies are numerous, ad-hoc, and often contradictory, leading to confusion and a growing body of work that cannot be compared. The lack of a systematic perspective leads to a mismatch between what software compartmentalization needs to progress towards the mainstream, and the focus and framing of research efforts: most do not tackle compartmentalization's key aspects as a whole and thus produce solutions that cannot be relevant to making it a mainstream practice.

Recognizing these challenges, we propose a unified model providing a consistent framework for defining, understanding, and implementing compartmentalization. This model comes with a comprehensive taxonomy that allows to classify compartmentalization strategies based on their policy definition methods, abstractions, and mechanisms, providing a basis for systematic evaluation and comparison.

We validate our model and taxonomy by systematizing 211 research and 61 mainstream software systems implementing compartmentalization. Doing so, this study provides unique insights into where mainstream efforts have

[1]This work was primarily done while Hugo Lefeuvre was affiliated with the University of Manchester and SCI Semiconductor, and Nathan Dautenhahn with Rice University (with minor contributions at Riverside Research).

fallen behind, and which problems research needs to focus on to reach the mainstream. We show how modern production software, if at all compartmentalizing, still vastly relies on manual separation, custom abstractions, and heavyweight legacy mechanisms, failing to adapt to the evolving security landscape. For future research, we stress the need for a more holistic approach to compartmentalization, the simplification of policy definition, stronger and more holistic threat models in the light of confused deputies and hardware limitations, as well as more attention to the gaps we highlight between research and production. Overall, this SoK contributes:

- A fundamental model and conceptual framework for compartmentalization (§2).
- A taxonomy for evaluating compartmentalization approaches and its systematic application to a wide set of research (§3) and mainstream (§4) efforts.
- Insights throughout §2, §3 and §4 and their consolidation into high-level challenges (§5) that compartmentalization research should focus on to foster mainstream adoption.

2. A Model of Software Compartmentalization

Early seminal works introduced models such as hierarchical layers and separation of concerns [59, 97, 100, 122], the object capability model [97, 157], the access control matrix [157], the principle of least privilege [208, 209], information hiding [193], or information-flow control [96, 214, 239]. Although these models deeply influenced modern compartmentalization, we observe that use-cases, practices, and enforcement means have evolved such that these seminal works have become non-trivial to map to modern compartmentalization approaches. Thus, to consistently characterize compartmentalization, we must first define and model it.

In the following, we adopt definitions of *subject*, *object*, and *permission* from Saltzer and Schroeder [209] and Miller [177], which we summarize as follows: a subject (or *principal* [209]) is a unit of computation (e.g., an assembly instruction, a thread of execution), an object is a unit of privilege enforcement (e.g., a byte of memory, a socket), and permissions are actions subjects may perform on objects (e.g., read, write). We refer to a maximal set of subjects sharing identical (sets of) permissions as a *protection domain*.[1] Finally, we define a *program* similarly to ISO/IEC [32] as a syntactic unit (conforming to the rules of one or more programming languages), composed of subjects and objects, needed to solve a certain function, task, or problem. This allows us to define software compartmentalization as:

Definition I: A *compartmentalization* of a program P is the set of (1) a policy to separate P into two or more protection domains (called *compartments*), and (2) the enforcement of this policy at runtime.

[1]Note that defining *protection domain* from the viewpoint of subjects and their permissions and not from that of objects like Saltzer [209] is deliberate to better match modern practices where domains are code-centric (see §3). Nevertheless, both definitions can be used interchangeably.

Compartmentalization can be applied to any program: applications, OS kernels (microkernels [164] are compartmentalized kernels), hypervisors [221], firmware [145], among others. Compartmentalization can be *retrofitted* into existing monolithic programs, or present in the initial design of a program. This definition is similar to Provos et al. [198]'s *privilege separation*, expanded to general program types and trust models,[2] and corresponds to the application of the principle of least privilege [209] within a single program.

A key problem of compartmentalization is validating all control and data flows at compartment boundaries [198]; we refer to this as ensuring *interface safety*. Improper validation results, among others, in confused deputy vulnerabilities [123] well-studied in previous works [84, 88, 129, 160, 181]. Interface safety is the instantiation in compartmentalization of the problem of information-flow control [96].

Evaluating Compartmentalization. Compartmentalization approaches strive to optimize some or all of four properties: the *security and safety* benefits of compartmentalization (properties enforced, interface safety guarantees); the *performance* of separated software (compared to a monolithic design); the *compatibility* of compartmentalization with existing software and programming idioms (e.g., to minimize the reimplementation effort needed by programmers to compartmentalize); and the *usability* of separated software, ensuring that non-expert end-users can correctly operate (e.g., configure, maintain, monitor) compartmentalized software.

Striking the right balance between these properties is key to match compartmentalization approaches with real-world uses. There is no silver bullet [101, 117, 161, 191, 192, 202]; all approaches discussed next adopt different points in the design space. These trade-offs are thus central to this SoK.

Scope of this SoK. As per Definition I, we refer to software compartmentalization as applied *within* one program (application, kernel, etc.), similarly to Provos [198]'s privilege separation. As we show throughout this paper, this covers a vast and coherent body of work. Other isolation techniques [222] *across* programs or groups of programs are out of the scope of this SoK: e.g., isolation of applications on a commodity OS, whole-application sandboxing [46, 61, 103, 112], separation between user and kernel [86, 127], between VMs [63] hosting separate programs, between stakeholders (confidential computing [207]), between users, or containers [175]. Though these isolation works share challenges with software compartmentalization, we focus on the latter for space reasons. Other works such as Shu [222] complement this SoK with a broader scope.

2.1. Key Differentiators

Three aspects are key to characterizing compartmentalizations: the *choice of subjects*, of *security properties*, and of *trust/threat models*. We detail them next in our model.

[2]The definition of privilege separation by Provos et al. [198] separates *applications* with a *sandbox* threat model, which constitutes a subset of compartmentalization as we define it in Definition I.

Subject Selection. The choice of subjects (§2) can be approached in three ways: *code-centric*, *data-centric*, and *hybrid*. In code-centric (or *spatial*) approaches, subjects are defined as program instructions, and protection domains constitute code regions. For instance, the *libjpeg* image processing library can be put in its own protection domain [181]. In data-centric (or *temporal*, *horizontal* [244]) approaches, subjects are defined as temporal units of execution, e.g., a thread or a process. Protection domains may contain one or more of these subjects. For example, worker processes in a modern web server (see §4) constitute data-centric domains, all executing the same packet-processing loop in isolation. These two strategies are not mutually exclusive: hybrid (or *object-oriented* [117]) approaches consider data-centric subjects bounded within code regions, e.g., a thread bounded to a specific library. Less popular than code- and data-centric approaches, hybrid approaches have been explored for multi-instance code-centric domains [172].

Insight 1: *Code-centric, data-centric, and hybrid subject selections suit different programs*. Code-centric is appropriate when distrust can be directed at a particular code unit (e.g., third-party code), or when secrets are associated to specific data structures (e.g., secret keys). Data-centric is appropriate for programs handling mutually distrusting information flows, particularly with per-flow secrets, e.g., a web server serving mutually-distrusting clients. Hybrid approaches can accommodate mixed characteristics.

Target Properties. Compartmentalization can enforce various properties, including *integrity*, *confidentiality*, and *availability*. In this context, Integrity guarantees that a subject cannot write out of its protection domain. Confidentiality guarantees that a subject cannot read out of its protection domain. Availability guarantees that a subject cannot prevent other protection domains from executing normally. Integrity is a prerequisite for all other properties: availability generally cannot be provided without integrity, and similar issues arise when enforcing confidentiality without integrity [248]. Compartmentalization approaches can also enforce additional properties, typically to raise the bar against cross-compartment attacks. Among them, *Cross-Compartment Control-Flow Integrity* [161, 211] (CC-CFI) enforces valid control-flow across compartments: cross-compartment call sites can only call compartment entry-points they would normally call according to the global Control-Flow Graph (CFG). *Runtime re-compartmentalization* [187, 191] enables the policy to change at runtime to achieve more suitable security or performance trade-offs, e.g., as the load evolves.

Trust & Threat Model(s). Compartmentalization can materialize different trust models by assigning subjects to domains, deciding for which domain to prioritize least privilege, and which properties to enforce. We observe that all can be expressed as a composition of three key trust models: *sandbox*, *safebox*, and *mutual distrust*.

The sandbox trust model reduces the privileges of an untrusted subject s_u to protect the rest of the system.[3] Least-privilege is enforced on s_u only. A popular use-case is protecting against vulnerable code, such as the request parser in a web server. Its inverse, the safebox model (or *vault* [217]), reduces the privileges of the whole system to protect a trusted subject s_t. Here, least privilege is applied on everything but s_t. A typical use-case is the protection of cryptographic secrets [232]. Finally, the mutual distrust model aims, for two disjoint sets of subjects s_1 and s_2, to eliminate the privileges of both parties on the other. Here least privilege is enforced on both sets. A typical use-case is distrust among sandboxes to increase fault isolation [69].

All compartmentalization approaches rely on a *Trusted Computing Base* (TCB) [205] to enforce compartmentalization. The TCB varies across approaches; typically included are the CPU package and a compartmentalization runtime (also called *reference monitor* [59]), but a compiler, OS kernel, or additional libraries, may also be included.

Compartmentalization is applied to wide range of threats, including isolating and recovering from adversarial and non-adversarial faults, thwarting supply-chain attacks, or protecting other security mechanisms such as shadow stacks. There is thus no one single threat model of compartmentalization.

Insight 2: Because compartmentalization approaches define subjects, properties, trust models, and TCBs differently, to thwart a many different threats, *there is no one unified "threat model of compartmentalization"*.

3. A Taxonomy of Compartmentalization

We propose to view compartmentalizing software as the combination of three key problems (P):

(P1) How to determine the right policy to enforce? Addressed by *policy definition methods*.
(P2) How to express the notion of compartmentalization policies in software, programming models, and idioms? Addressed by *compartmentalization abstractions*.
(P3) How to enforce policies at runtime? Addressed by *compartmentalization mechanisms*.

Nearly all works from the literature target a subset of these challenges: e.g., SOAAP [117] addresses (P1), SMVs [128] (P2), Donky [217] (P3). In fact, P1-P3 are rarely addressed all at once for scale reasons. We ground our systematization on this division of challenges by comprehensively systematizing each challenge (§3.1 - §3.3).

Selection Methodology. We manually filter the program of top-tier security and systems conferences for 2003-2023 (9,083 papers) through titles to obtain a list of potentially relevant works (923 papers). We choose 2003 as the point when application compartmentalization gained visibility in research with the seminal work of Provos [198] and Kilpatrick [147]. We then inspect abstracts to reduce this list

[3] *Sandbox* can also refer to whole-program privilege reduction [103], e.g., with seccomp [46], or language-based techniques [112]. This matches our sandbox model, but not our definition of compartmentalization (see scope).

to 96 relevant papers. To cover works prior to 2003 and from other venues, we apply *snowballing* [250] to each paper of our set to identify 66 further works, and complete the list with 49 more from our knowledge of the industry and literature, totaling 211 works. 106 of them, featured in Tables 1 to 3, address at least one of P1-P3. Our goal is not exhaustivity (this is not a survey), but to capture a representative set of influential compartmentalization works. We define our data points by extracting, for P1-P3, key characteristics to differentiate these solutions. This results in 7 differentiators for P1, 8 for P2, and 7 for P3 (Tables 1 to 3). Works spanning several categories (e.g., P1-P2), are studied independently from both perspectives. Interested readers can read more about our methodology in Appendix A.1.

3.1. Policy Definition Methods (P1)

Definition II: A *Policy Definition Method* (PDM) is a method to define a program privilege separation policy. PDMs identify subjects, objects, and permissions to enforce, and may be applied to existing and new codebases.

In this section we systematize PDMs on the basis of our taxonomy. It is designed to be read along with Table 1.

Automation. Automation is a key research problem in compartmentalization. It is structural for all other characteristics discussed in this section. Through automation, works seek to better restrict privileges or safeguard interfaces, and limit developer effort and performance overheads. We propose a taxonomy of four degrees of automation:

Manual methods (○ in Table 1) rely on the expert knowledge of developers to separate software. Developers must define a policy at the lowest level: *which* subject is given *what* permissions for *what* object. When determining permissions, manual approaches can be accurate, but are prone to human error, resulting in *false positives* (under-privileged compartments, hurting reliability) and *false negatives* (over-privileged compartments, weakening security properties). Similarly, performance and interface safety widely depend on expert knowledge and human error [117, 161]. Overall, it is not possible to guarantee quality or correctness with manual separation because of its reliance on human expertise and engineering effort. Still, much of the literature falls in the manual category (§3.2), and manually-separated software can achieve robustness and reliability (§4).

Guided manual methods (◔) such as SOAAP [117] are manual, but provide developers with tools to make the separation less tedious and error-prone. Often featuring a feedback loop [117, 181], they guide users to define and protect boundaries. Guided methods can bring firm guarantees to manually-separated software, e.g., eliminating classes of confused deputies [181] or information leaks [117].

Policy-refinement methods (◓) automatically separate provided a high-level policy from the developer. Such policies, written in a *policy language*, provide the PDM with semantic information (e.g., annotations associating objects with a confidentiality level [261]) and/or high-level instructions (pinpoint a library to sandbox [65]). Policy-refinement approaches automatically refine this information into *concrete, low-level* policies, greatly limiting the amount of expertise required from developers, but still requiring enough skills to provide a high-level policy.

Fully automated methods (●) automatically separate software without any policy input from the user. Instead of relying on semantic information on the software, which is hard to infer automatically [77, 260], fully automated approaches analyze the software for data dependencies, and apply least privilege on this basis [202, 253]. One drawback of relying on data dependencies only is that, due to the architecture of software, data-dependency relationships may exist between confidential data and untrusted parts, resulting in partitionings that feature weaker security properties [202].

Insight 3: The ability of PDMs to prevent confused deputies and information leaks is directly linked to their understanding of software semantics, which diminishes as automation increases. Thus, *policy refinement (◓) and full automation (●) trade off security for developer effort.*

Policy Languages. Policy languages allow developers to describe high-level policies to Guided manual (◔) or Policy-refinement (◓) PDMs. They are key to achieve intermediate degrees of automation, providing PDMs with an understanding of program semantics and trust relationships which is otherwise hard to extract automatically [260]. We distinguish between two types of policy languages: *annotations*, and *placement rules*. <u>Annotation</u>-based policies provide fine-grain semantics on subjects and objects, such as describing shared, confidential, or sensitive entities [73, 117, 161, 169–171, 261] (e.g., object `key` is confidential, function `parse()` is sensitive), past vulnerabilities [117], or performance goals and bottlenecks [117, 170]. Annotations are tightly coupled with program code such that they may be provided by external dependency vendors [117]. Conversely, <u>placement rule</u> languages provide coarse-grain, high-level descriptions of component trust relationships [65, 161, 261] and/or building rules [90, 170] (e.g., place libraries X and Y in separate domains). Placement rules are independent from program code, and are thus more commonly provided by system integrators. They can be expressed in many ways, including human-readable JSON files [65, 161], or integrated into the build-system [133]. Both classes are not mutually exclusive and may be combined by PDMs [161, 170].

Insight 4: Annotations and placement rules have *different expressivity*: annotations express local, low-level semantics, whereas placement rules express full-system properties. Both suit *different threat models*: since annotations are tightly coupled with program code, they may be vulnerable to supply-chain attacks, unlike placement rules. This provides strong incentives to combine the two approaches, which is not a common practice (cf. Table 1).

Separation Granularities. Many granularities have been explored (Table 1): functions [242], libraries [65], linkage

TABLE 1: *Taxonomy of Policy Definition Methods.* PDMs that also propose an abstraction are marked with ✻. Manual (○) and fully automated (●) policies do not leverage a policy language, thus the column features Not/Applicable.

Policy Definition Method	*Automation* ○◔◓●[1]	*Policy Language Type*		*Separation Granularity*	*Analysis Approach*	*Subject Selection*	*Language Specific*	*Additional Goals of Automation*	
		Annotations	*Placement Rules*					*Performance*	*Interface Safety*
Manual [128], [...]	○	N/A	N/A	Any	Manual	Any	○	N/A	N/A
Crowbar [70]	◔	○	○	Function	Dynamic	*Code*-centric	○	○	○
MPDs✻ [191, 192]	◔	○	○	Component	Hybrid	*Code*-centric	●	●	○
CubicleOS✻ [211]	◔	●	●	μLibrary	Static[2]	*Code*-centric	●	○	○
Google SAPI✻ [22]	◔	●	●	Function	Static	*Code*-centric	●	○	○
FlexOS✻ [143, 161]	◔	●	●	μLibrary	Dynamic	*Code*-centric	●	●	○
RLBox✻ [181]	◔	●	○	Function	Static	Any	●	○	●
SOAAP✻ [117]	◔	●	○	Any	Hybrid	Any	●	●	●
SeCage✻ [171]	◓	●	○	Function	Hybrid	*Code*-centric	●	○	○
PtrSplit✻ [169]	◓	●	○	Function	Static	*Code*-centric	●	○	○
PrivTrans✻ [73]	◓	●	○	Function	Hybrid	*Code*-centric	●	○	○
Glamdring✻ [165]	◓	●	○	Function	Static	*Code*-centric	●	○	●
Shreds✻ [87], CAPACITY✻ [105]	◓	●	○	Any	Static	*Code*-centric	○	○	●
DataShield✻ [77]	◓	●	○	Any	Static	*Code*-centric	○	○	●
Swift✻ [89]	◓	●	○	Any	Static	*Code*-centric	●	○	●
Jif✻ [261]	◓	●	●	Any	Static	*Code*-centric	●	○	●
PM [170]	◓	●	●	Function	Hybrid	*Code*-centric	●	●	○
KSplit✻ [133]	◓	●	●	Driver	Static	*Code*-centric	●	○	○
Cali✻ [65]	◓	○	●	Library	Static	*Code*-centric	○	○	○
CompartOS✻ [55]	◓	○	●	Linkage Unit	Static	*Code*-centric	○	○	○
Enclosure✻ [111]	◓	○	●	Package	Static	*Code*-centric	●	○	○
BreakApp✻ [235]	◓	○	●	Package	Static	*Code*-centric	●	○	○
CompARTist✻ [132]	◓	○	●	Library	Static	*Code*-centric	●	○	○
ACES✻ [90]	◓	○	●	Function	Any[3]	*Code*-centric	○	○	○
ProgramCutter [253]	●	N/A	N/A	Function	Dynamic	*Code*-centric	●	○	○
μSCOPE [202], SCALPEL [203]	●	N/A	N/A	Any	Dynamic	*Code*-centric	○	●	○

[1] ○ = manual, ◔ = guided manual, ◓ = policy refinement, ● = full automation. [2] Loader-based. [3] Implemented with static analysis, dynamic analysis possible [90].

units [55], drivers [133], software packages [111], etc. These choices are guided by design decisions. On the one hand, finer granularities of separation make it possible to better enforce least privilege, or reach boundaries more favorable to performance. Fine granularities may also be necessary to tackle certain vulnerabilities, e.g., Heartbleed [104]. Conversely, for some separation approaches, the threat model itself may be defined at a coarse granularity; larger components such as libraries or packages are valid units of trust in real-world scenarios such as supply-chain attacks [235]. Operating at coarser granularities may also reduce developer effort and expertise requirements: as granularities become finer, it becomes more complex to express policies and reason about them; higher-level boundaries such as libraries are more intuitive separation units than arbitrary internal functions. Finer granularities may also negatively impact interface safety: as boundaries are set at internal, less encapsulated software layers [202], interfaces are more exposed to confused deputies and harder to safeguard [161]. Finally, finer granularities pose technical challenges of state explosion [161], performance [134] and analysis/clustering [202].

Analysis Techniques. Automated PDMs (◔◓●) employ a range of static, dynamic, and hybrid techniques which, through subtle trade-offs, strongly impact final properties.

Static methods. When determining permissions, approaches based on static analysis are *complete* but conservative: separated software is guaranteed to function correctly, but compartments may be left over-privileged due to the fundamental imprecision of static analysis [90, 170]. When applied to performance analysis, static approaches can provide useful [170] albeit imprecise performance metrics [117]. Applied to improve interface safety, static analysis approaches can detect potential issues at scale and systematically [181], but cannot yield precise impact metrics [129].

Insight 5: Though the problem of over-privilege in static analysis is well-known and its impact characterized [160], it has not been *quantified*, such that it is unclear to what extent static PDMs trade security for developer effort.

Dynamic methods. When determining permissions, dynamic methods guarantee that permissions granted to compartments are strictly necessary. However they are *incomplete*: due to their fundamental coverage problem [170, 202] domains may be left under-privileged, so that separated software may not function correctly anymore for all workloads. Similarly, dynamic methods can provide precise performance estimates [161, 202] but only on covered workloads. Applied to interface safety, they can detect vulnerabilities and their concrete impact, but not systematically [160]. Due to the problem of incompleteness, very few PDMs are dynamic (5 out of 27 in Table 1).

Hybrid methods. Static and dynamic methods do not compose well, as it is hard to utilize the delta between static and dynamic results. On the privilege detection side, some use this delta as the *suspicious subset* [171], authorizing but reporting uses, which poses usability questions. Others use dynamic results to optimize compartmentalizations [73], to predict performance accurately with an otherwise static approach [117], or to measure information flow [170].

Subject Selections. Policies can apply to different types of subjects: code-centric, data-centric, hybrid, each suiting different software characteristics (see §2.1). This choice strongly impacts the design of PDMs, particularly when targeting automation. Surprisingly all automated PDMs are code-centric, and non-code-centric PDMs all fall into the guided- or manual categories (cf. Table 1).

Insight 6: *Research in PDMs largely focuses on code-centric separation.* We speculate that this is caused by the lesser popularity of data-centric approaches (we repeat this observation in §3.2), but also to the greater complexity of data-centric separation: while many automated code-centric PDMs assume single-threaded programs, data-centric requires concurrent separation, complexifying the analysis [169]. This calls for more research in automated PDMs for data-centric and hybrid subjects.

Genericity. Most PDMs specialize on classes of programming languages (cf. Table 1), for several reasons. First, some focus on domain-specific problems or threat models: e.g., pointer aliasing in C [169], untrusted packages in modern languages [111]. Second, specializing on language classes allows PDMs to leverage language specificities: their type system to detect API sanitization needs [181]; their memory safety [89, 261] or interpreted nature [111, 235] to simplify boundary detection; or the overall system architecture [132] to make assumptions on boundaries. There is, for instance, a vast body of work (only partially covered in Table 1 for space reasons) specifically targeting 3rd-party Android library code [130, 219, 227], with some [132, 168, 194, 220, 263] specializing entirely on advertisement libraries. This is well-covered by Acar et al. [54].

3.2. Compartmentalization Abstractions (P2)

Definition III: A *compartmentalization abstraction* defines and implements primitives to express separation policies in a program. Depending on the semantics of these primitives, abstractions may be used to express different types of subjects, trust models, properties, etc.

We first contribute a model to characterize the core primitives of compartmentalization abstractions. We then leverage this model to systematize existing approaches. This section is designed to be read along with Table 2.

A Model of Compartmentalization Abstractions. Compartmentalization abstractions instantiate the notion of a *compartment*, defining the type of subjects compartments may isolate, properties they may enforce, and the trust models that may be implemented. They must also define five primitives: *`CREATE`* and *`DESTROY`* a compartment (defining the semantics of compartment lifetime management, the default permissions of new compartments, among others); *`CALL`* and *`RETURN`* from a compartment (defining cross-compartment control-flow semantics); and *`ASSIGN` privileges* (granting and revoking permissions across compartments, resource ownership). Abstractions achieve trade-offs by controlling the semantics of these primitives.

Not all five primitives must be exposed to developers; when they are exposed, we refer to them as having *explicit* semantics. Inversely, *implicit* primitives are handled automatically under the hood. Implicit semantics are common in abstractions that are coupled with the PDM (e.g., automatic `CREATE`/`DESTROY`, transparent `CALL`s, automatic `ASSIGN`). Additionally, abstractions may provide other primitives, e.g., to support fault tolerance. We now detail the core primitives, focusing on `CALL/RETURN` and `ASSIGN` for space reasons.

`CALL/RETURN`. Regardless of their implementation, `CALL` and `RETURN` must meet minimum safety requirements: 1) guaranteeing the validity of control-flow entry-points in compartments (compartments should not `CALL` or `RETURN` to arbitrary code in the context of other compartments); 2) switching call stacks and clearing registers appropriately to avoid unintentional leakages; but also 3) ensuring that the abstraction composes safely with other system interfaces.

Cross-compartment control flow can be approached *synchronously*, or *asynchronously*. In the <u>synchronous</u> case, `CALL` and `RETURN` are semantically similar to a local function call and thus transparent to separated programs. In the <u>asynchronous</u> case, `CALL` and `RETURN` abandon function-call semantics: the execution of the caller domain continues after the call, and the return is processed by the caller similarly to a separate message [157] (e.g., as part of an event loop). Asynchronous semantics are less popular in Table 2. This is likely due to the fact that they are more disruptive compatibility-wise: applications must be "structurally" aware of the separation, and redesigning for asynchronous behavior is non-trivial [244]. Still, asynchronous semantics can be beneficial to performance, as their non-blocking nature can mask boundary-crossing delays [223]. For certain target properties such as availability, `CALL` and `RETURN` ultimately need distantiating from function call semantics, as new error types appear: timeout, callee compartment failure, etc. These translate into asynchronous features in call semantics that may otherwise be synchronous [55].

`ASSIGN`. `ASSIGN` semantics are structured by two fundamental approaches to communicating data [60]: *message passing*, and *shared data* (or *message/object* systems [157]). With <u>message passing</u>, domains share data across boundaries via messages over a communication channel (e.g., POSIX sockets or pipes). This means that objects are not only systematically copied, but also marshalled, and as part of this, potentially serialized and checked. This makes message-based solutions rather disruptive compatibility- and performance-wise: they do not map to natural shared-memory semantics found in applications, and require copies. Still, systematic copying and checking greatly benefits security [160]. With <u>shared memory</u>, protection domains both have privileges over shared memory, and communicate via loads/stores. Although copies can still be made systematic by the abstraction [181], it is not the norm: copies are costly, it is thus enticing to avoid them whenever possible. This makes shared memory much less disruptive compatibility- and performance-wise, but potentially deceptive security-

TABLE 2: *Taxonomy of Compartmentalization Abstractions. Targets*: User, Kernel, HyperVisor. *Semantics*: $\mathcal{S}$ynchronous, $\mathcal{A}$synchronous, SHared Memory, MESsage passing. $\mathcal{S}$+$\mathcal{A}$: the abstraction exhibits both $\mathcal{S}$ and $\mathcal{A}$ semantics. *Properties*: Confidentiality, Integrity, Availability, Recompartmentalization. Mechanism-independent abstractions are labeled with ∅.

Class	*Target* U/K/HV	*Abstraction*	*Subject Selection*	*Semantics* CALL	ASSIGN	*Abstraction Granularity*	*Properties* C	I	A	R	*Interface Safety*	*Design Bound to Mechanism*
Mutual Distrust	U	Virtines [242]	*Code*-centric	$\mathcal{S}$	MES	Function	●	●	○	○	○	Virtual Machine (EPT)
		ACES [90]	*Code*-centric	$\mathcal{S}$	SHM	Function	●	●	○	○	○	∅[5]
		SeCage [171]	*Code*-centric	$\mathcal{S}$	SHM	Function	●	●	○	○	○	∅[5]
		HODOR [124]	*Code*-centric	$\mathcal{S}$	SHM	Library	●	●	○	○	○	∅
		CAPACITY [105]	*Code*-centric	$\mathcal{S}$	SHM	Any	●	●	○	○	○[4]	ARM PAC + MTE
		Jif [261]	*Code*-centric	$\mathcal{S}$+$\mathcal{A}$	MES	Any	●	●	○	○	●	∅
		Arbiter [241]	*Data*-centric	$\mathcal{S}$	SHM	Function[1]	●	●	○	○	○	∅[5]
		Secure Memory Views (SMVs) [128]	*Data*-centric	$\mathcal{S}$	SHM	Function[1]	●	●	○	○	○	∅[5]
		Salus [226]	*Data*-centric	$\mathcal{S}$	SHM	Function[1]	●	●	○	○	○	∅[5]
		Light-Weight Contexts (LwCs) [167]	Hybrid	$\mathcal{S}$	SHM	Function[1]	●	●	○	○	○	Page Table[2]
		POSIX Processes (and earlier instances) [93]	Hybrid	Any	Any	Any	●	●	○	○	○	Page Table
		SOAAP [117]	Hybrid	$\mathcal{S}$	SHM	Any	●	●	○	○	○[4]	∅
		libMPK [190]	Hybrid	$\mathcal{S}$	SHM	Any	●	●	○	○	○	Protection Keys
	U+K	CheriOS [108]	*Code*-centric	Any	Any	U/K-component	●	●	○	○	○	CHERI
		Microkernel Servers [106], [...]	*Code*-centric	Any	MES	U/K-component	●	●	○	○	○	∅[5]
		Mutable Protection Domains (MPDs) [191, 192]	*Code*-centric	$\mathcal{S}$	SHM	U/K-component	●	●	○	●	○	∅[5]
		RedLeaf [184]	*Code*-centric	$\mathcal{S}$	SHM	U/K-component	●	●	●	○	○	Safe Languages
		CubicleOS [211]	*Code*-centric	$\mathcal{S}$	SHM	μLibrary	●	●	○	○	○	Protection Keys
		FlexOS [161]	*Code*-centric	$\mathcal{S}$	SHM	μLibrary	●	●	○	○	○	∅
		xMP [196]	*Code*-centric	$\mathcal{S}$	SHM	Any	●	●	○	○	○	∅
		Monza [35]	Hybrid	$\mathcal{A}$	SHM	Function[1]	○	●	○	○	○	∅[5]
	K	VirtuOS [186]	*Code*-centric	$\mathcal{S}$+$\mathcal{A}$	SHM	K-component	●	●	●	○	○	Virtual Machine (EPT)
		HAKC [173]	*Code*-centric	$\mathcal{S}$	SHM	Function	●	●	○	○	○	ARM PAC + MTE
		LibrettOS [187]	*Code*-centric	$\mathcal{S}$	SHM	K-component	●	●	●	●	○	∅[5]
Sandbox	U	Cali [65]	*Code*-centric	$\mathcal{S}$	SHM	Library	●	●	○	○	○	∅[5]
		CompARTist [132]	*Code*-centric	$\mathcal{S}$	MES	Library	●	●	○	○	○	∅[5]
		Enclosure [111]	*Code*-centric	$\mathcal{S}$	SHM	Package	●	●	○	○	○	∅
		Google Sandboxed API (SAPI) [22]	*Code*-centric	$\mathcal{S}$	MES	Function	●	●	●	○	○	∅[5]
		RLBox / μSWITCH [181, 195]	Hybrid	$\mathcal{S}$	SHM	Function	●	●	○	○	●	∅
		Wedge [70]	Hybrid	$\mathcal{S}$	SHM	Function[1]	●	●	○	○	○	∅[5]
	U+K	CompartOS [55]	*Code*-centric	$\mathcal{S}$	SHM	Linkage Unit	●	●	●	○	○	CHERI
	K	LVDs / KSplit [133, 185]	*Code*-centric	$\mathcal{S}$	MES	K-component	●	●	○	○	○	∅[5]
		XFI/LXFI [107, 172]	Hybrid	$\mathcal{S}$	SHM	K-component	●	●	○	○	●	SFI
	HV	Nexen [221]	*Data*-centric	$\mathcal{S}$	SHM	Per-VM domain	●	●	○	○	○	Page Table[3]
Safebox	U	Shreds [87]	*Code*-centric	$\mathcal{S}$	SHM	Any	●	●	○	○	○[4]	∅[5]
Safebox / *Dual World*	U	Privman [147]	*Code*-centric	$\mathcal{S}$	MES	Function	●	●	○	○	○	Page Table[2]
		Privtrans [73]	*Code*-centric	$\mathcal{S}$	MES	Function	●	●	○	○	○	Page Table[2]
		Swift [89]	*Code*-centric	$\mathcal{S}$+$\mathcal{A}$	MES	Any	●	●	○	○	●	∅
		Glamdring [165]	*Code*-centric	$\mathcal{S}$	MES	Function	●	●	○	○	●	∅[5]
		PtrSplit / Program Mandering [169, 170]	*Code*-centric	$\mathcal{S}$	MES	Function	●	●	○	○	○[4]	∅[5]
		DataShield [77]	Hybrid	$\mathcal{S}$	SHM	Any	●	●	○	○	○[4]	Bounds Checking
		ERIM [232]	Hybrid	$\mathcal{S}$	SHM	Any	●	●	○	○	○	Protection Keys
	K	Nested Kernel [94]	*Code*-centric	$\mathcal{S}$	SHM	Function	●	●	○	○	○	Page Table

[1] Inherited from thread-like semantics, [2] from process-like semantics, [3] from the Nested Kernel, [4] The PDM does (to a certain extent).
[5] The abstraction could plug onto any intra-AS mechanism, though the paper or documentation claims reliance on a particular one.

wise [160]. Note that we describe exposed abstraction semantics here; under the hood shared-data can be implemented on top of message passing, and vice-versa [60].

Insight 7: Whereas compartmentalization abstraction semantics were historically asynchronous and message-passing based, new trends in retrofitting separation shifted the focus to synchronous and shared-memory approaches. *This comes at a non-trivial security and performance cost.*

Trust Models. Safebox, sandbox, and mutual distrust (§2.1) are all represented in Table 2. Sandbox and mutual distrust abstractions all support scenarios with an arbitrary number of compartments. Although Shreds [87] supports arbitrary safebox scenarios, all other safebox abstractions are limited to two compartments (trusted vs. untrusted, "*Dual World*" in Table 2). This shows a lesser interest in applying distrust among trusted entities. Though safebox and sandbox abstractions can both be implemented on top of arbitrary mutual distrust (and can thus be seen as special cases), the presence of safeboxes or sandboxes only, or of a fixed number of compartments, considerably simplifies their semantics. Each presents trade-offs. Mutual distrust abstractions can express true least privilege. However, this comes at a performance cost, as they must enforce integrity and other properties on all compartments, whereas one-sided distrust models (safebox, sandbox) must only enforce them for the trusted side. Mutual distrust also makes interface hardening generally more challenging and costly in performance [160].

Enforcing More or Fewer Properties. All abstractions in Table 2 enforce integrity, a prerequisite for other properties (§2.1). Most target confidentiality, and some target availability, runtime re-compartmentalization, or interface safety.

Confidentiality. Since nearly all abstractions provide confidentiality, we focus on the impact of *not* doing so. Not providing confidentiality does not cause structural changes in abstractions [35]. It may benefit performance, as it enables zero-cost read-only data sharing, i.e., fewer copies at compartment boundaries. It also simplifies separation: only write-shared data require explicit sharing. On the downside, not providing confidentiality vastly reduces the abstraction's ability to counter information leakages: the only remaining barrier is the compartments' ability to limit data exfiltration vectors. It may also be detrimental from an interface-safety viewpoint, defeating randomization (thus easing cross-compartment exploits). Lastly, avoiding copies at boundaries may make the system more prone to shared-memory TOCTOU [160]. Overall, except for legitimate integrity-only use-cases (e.g., shadow-stack protection [155]), giving up confidentiality trades security for performance and compatibility.

Availability. Abstractions may go beyond fault isolation and target cross-compartment fault tolerance. This brings many well-known challenges from the fault-tolerance and distributed systems fields [234]: e.g., avoiding, detecting, recovering from resource exhaustion, and from various failures (omission, Byzantine). Particularly relevant to compartmentalization are resource ownership problems [184] (when restarting a domain, can shared resources be released?), and state coherence issues [71] (the state of restarted domains may be incoherent with that of other components), as compartmentalized components, particularly when retrofitting, are typically less encapsulated [160]. Overall, availability differs from properties such as integrity and confidentiality in that it is a property of the whole system, not a compartment-local property: to achieve availability with compartmentalization, the interaction (and failure) of all domains must be considered at once, instead of separately.

To tackle these challenges, abstractions take a vast variety of approaches, whose exhaustive listing outreaches the scope of this paper: bounded execution in resources [186] or time [55, 117], and generally performance isolation [118] to tackle resource exhaustion; leveraging type systems for resource ownership [184]; proposing manually-designed interface wrappers [184, 229], per-component fault-handlers [55], careful TCB and interface designs to store state outside of domains [187], or recursive restarting of relevant components [186] for state coherence. Because the problem is hard, all require expert understanding of fault domains, rely on a variable amount of manual engineering, and not all approaches are complete; e.g., Google SAPI [22] automatically re-iterates failed calls and restarts faulty components, but does not consider state coherence problems.

Insight 8: *Very few compartmentalization abstractions target availability*. We speculate that this is due to the additional complexity of fault tolerance, as discussed above. This is problematic: as we show in §4 availability is a common need of real-world deployments. *This calls for more work on fault-tolerant compartment abstractions.*

Runtime Re-Compartmentalization. Some abstractions also support changing the policy enforced at *runtime* [187, 191, 192], e.g., to adapt policies to evolving requirements in performance and fault isolation. Beyond technical challenges of achieving transparent, fast re-compartmentalization, we observe that this poses non-trivial security challenges in adversarial contexts: assuming attackers can wait for the weakest fault-isolation profile to be instantiated, or influence workloads to trigger such profiles (e.g., generate more, or different network traffic), then the overall security properties are that of the *weakest* profile achievable at any point in time. This may be true even when attackers cannot wait; since component states are preserved across policy changes, any undetected corruption triggered by an attacker during a "strong" profile will eventually reach other components when weaker profiles trigger, similar to delay attacks [255]. This limits applications of re-compartmentalization to non-adversarial scenarios.

Compartment-Interface Hardening. Although interface safety is a key problem of compartmentalization, most abstractions consider it orthogonal to their mission (Table 2). Works often transfer the responsibility to PDMs by assuming a safe separation policy, or to downstream developers by assuming hardened components. Yet, though the purpose of compartmentalization abstractions is not to help users defining security policies (this is the role of PDMs), they can contribute to interface safety by *making it harder to implement unsafe policies*, thus ensuring that compartment interfaces are free of certain classes of confused deputies. They may do so at various levels, e.g., by *enforcing restrictions on interface definitions*, such as restricting interface-crossing types [172, 181] or enforcing points-to ranges for interface-crossing pointers [146, 162, 165, 172, 181]; by *forcing the presence of checks on interface-crossing data*, forcing users to write checks [181], or checking automatically when possible [165, 172, 181]; or by *enforcing restrictions on cross-compartment control-flow*, providing primitives to specify and enforce API call ordering [181], and enforcing additional properties such as CC-CFI to raise the bar for cross-compartment attacks [161, 211]. These measures are not orthogonal to the core mission of compartmentalization abstractions, as they generally cannot be implemented independently: without knowledge of compartments mappings, it may be impossible to verify pointers and indexes; to check reference types or call ordering; to implement CC-CFI; etc. Lefeuvre [160] and Hu [129] provide deeper coverage of the topic of interface safety in compartmentalization.

Insight 9: There is a widespread misconception that interface safety is orthogonal to the mission of compartmentalization abstractions. *More work is needed on abstractions that contribute to compartment-interface hardening*.

Composing P1, P2, and P3.

Subject Selection. Similarly to PDMs, most abstractions specialize towards certain subject types (Table 2). Abstractions focusing on code-centric models assume a direct mapping between code and compartment. As a result, CREATE and DESTROY are implicit: developers are not provided with explicit controls to manage compartment lifetime, and the abstraction is *static*, i.e., the number and content of compartments is known at compile time. Conversely, those focusing on data-centric models feature explicit CREATE and DESTROY, and are *dynamic*, i.e., the number and content of compartments may not be known at compile time and depend on the control flow taken at runtime. Abstractions that are hybrid are similar to data-centric abstractions but allow restricting the code available to a compartment; they can be used to implement both code- and data-centric separations.

Granularities. Abstractions may also specialize towards specific domain granularities. These decisions are embodied in CREATE and CALL semantics which define the granularity at which compartments may be created and entered. This specialization comes with various goals: target properties may imply a granularity (e.g., fault resilience implies coarser grains), threat models may imply a granularity (library sandboxing → library granularity), or abstractions may be coupled together with a PDM that itself restricts granularity.

Mechanisms. Many abstractions are tightly coupled with a specific mechanism (Table 2). This is typically done to better leverage mechanism-specific properties such as safe copy-less sharing for capabilities, or strong typing, points-to knowledge, and provable termination for safe languages. Tightly coupling with mechanisms has drawbacks: beyond making abstractions unusable without their related mechanism, this creates a mechanism dependency in downstream programs, which curbs efforts to roll out new mechanisms (e.g., `fork()` makes it hard to replace the page table [66]). These limitations incited a recent trend towards mechanism-agnostic abstractions [155, 161, 181].

Insight 10: Design decisions on either of P1, P2, and P3 have implications across the stack. The split of P1-P3 has its limitations and *all three problems must eventually be considered together to achieve harmonious solutions.*

Composing with Other Abstractions.

Threads. Compartmentalization abstractions must define a threading model for compartments. We classify threading models as either *orthogonal* or *coupled*. In the orthogonal case [161, 172, 232], threads cross protection domains as they CALL and RETURN. To ensure safety, these operations must guarantee that the underlying thread state is updated to reflect the crossing, and carefully define the behavior of thread local storage. By definition, orthogonal threading models exclusively suit *code-centric* approaches. In the coupled case [70, 128, 241], threads are immutably assigned to a protection domain at their creation, and CALL spawns a new thread in the desired protection domain. Coupled threading models suit any subject selection.

CPU Privilege Levels (Rings). In userland, semantics of compartmentalization abstractions are heavily influenced by the presence of the user/kernel interface. Compartmentalization abstractions may [128] or may not [232] be exposed as part of the user/kernel boundary for performance, security, or compatibility reasons. As we discuss next, user-mode abstractions must harmoniously compose with kernel interfaces such as processes, signals, or system calls. Different factors shape kernel and hypervisor mode compartmentalization abstractions. Kernel and hypervisor codebases are often designed assuming ambient privilege on hardware and user data, making the TCB (and retrofitting) less obvious than in userland. The need to integrate with low-level events such as interrupts brings even more specific challenges that make it necessary for abstractions to integrate deeper in kernel and hypervisor designs [94, 161, 185], encompassing boot, scheduling, memory management, or interrupt handling.

Processes. Though processes are themselves a compartmentalization abstraction, they are also used by programs for reasons other than protection [66]. Several prior works showed that composing compartmentalization abstractions with processes is error-prone [91, 216]. User-mode compartmentalization abstractions must thus take special care defining how they compose with processes. To ensure safety, approaches may intercept and forbid attempts to spawn new processes [216], at the expense of application compatibility.

System Calls & Other System Interfaces. We discussed earlier (Insight 9) how compartmentalization abstractions can contribute to safeguarding intra-program compartment interfaces. Yet, interfaces with the *rest of the system* are also major interface safety weak spots [91, 160, 213, 237]. These include, in user-mode, system calls, other kernel abstractions (pseudo file-systems [48, 49], files, sockets), but also interfaces exposed by other applications on the system [85]. Safeguarding these interfaces is non-trivial: mechanisms come with different protection needs (e.g., protection keys with "PKU pitfalls" [91]); protection needs are ABI-bound and thus vary across OSes, configurations, and architectures; protection must be maintained as these ABIs evolve; the protection effort itself comes with application compatibility problems (e.g., precisely detecting the OS features required by individual application components is hard [76, 95]); and this protection often results in noticeable performance overheads [91]. Compartmentalization abstractions either attempt to solve this problem through careful composition with the kernel [111, 117, 167, 181, 195, 216], or scope it out as a separate problem [124, 169, 232] addressed by other solutions [43, 46, 197, 243]. Still, there is a growing consensus that user-mode compartmentalization abstractions should be designed hand in hand with kernel and broader system interfaces to maximize interface safety [91, 195, 216].

3.3. Compartmentalization Mechanisms (P3)

Definition IV: A *Compartmentalization mechanism* enforces separation, as defined by PDMs and implemented through compartmentalization abstractions, at runtime.

Next, we concretize this definition by modeling the fundamental primitives a compartmentalization mechanism must provide. Then, we show how mechanisms approach these primitives to reach trade-offs, going through Table 3.

TABLE 3: *Taxonomy of Compartmentalization Mechanisms.* Page-Table = PT; Permissions: Read, Write, Execute, Address (*create pointers to*), ● = supported, ◐ = supported by some, ○ = unsupported; Overhead: *free* = ○ < ◔ < ◓ < ◕ < ● = *very costly*.

	Mechanism Class	*Conditioned*	*Trust Model*	*TCB*	*Permissions* R	W	X	A	*Granularity*	*№ of Domains*	*Domain Switch Cost (Versus Non-Separated)*
Hardware	Physical Separation [205]	○	Mutual	Full	●	●	●	○	Physical Mem.	№ of machines	◓ – ● (link latency)
Hardware	PT: Access Bits [25], EPT / `vmfunc` [26]	◐[1]	Mutual	Full	●[2]	●[2]	●[2]	○	Page	∞	● (PT switch + [5])
Hardware	PT: Supervisor Bit [25, 158, 159]	○	Single	Full	●[2]	●[2]	●[2]	○	Page	2 (kernel/user)	◕ (interrupt + [5])
Hardware	PT: Mondrian Memory Protection (MMP) [249]	○	Mutual	Full	●[2]	●[2]	●[2]	○	Word	∞	◓ (MMP hardware + [5])
Hardware	PT: Protection Keys [13, 27, 53, 217, 259]	●	Mutual	Full	●	●	◐	○	Page	8-1024 [13, 217][4]	◔ (special register flip + [5])
Hardware	Segmentation-like Hardware [109, 178]	○	Single	Full	●	●	○	○	Byte - Page [178]	2 (safe/unsafe)	◔ ([5])
Hardware	TEE: Enclaves [28, 92]	○	Mutual	TEE	●	●	●	○	Page	∞	● (enclave call, incl. [5])
Hardware	TEE: Confidential VMs [10, 29, 30]	○	Mutual	TEE	●	●	●	○	Page	∞	● (> EPT switch)
Hardware	TEE: World Separation [9, 14]	○	Single	TEE	●	●	●	○	Page	2 (trusted/rest)	● (world switch, incl. [5])
Hardware	Hardware Capabilities [57, 78, 180, 236, 244]	○	Mutual	Full	●	●	●	●	Byte	∞	◓ (special instr. + [5])
Hardware	Bounds-Checking Hardware [47, 98, 148, 155, 212]	●	Mutual	Full	●[2]	●[2]	○	○	Byte	∞	◓ (bounds hardware + [5])
Hardware	(Other) Tagged Architectures [12, 21, 99, 131, 138, 204, 224, 246]	●	Mutual	Full	●[2]	●[2]	○	○	Byte - Words[3]	16[4] - ∞ [138]	◓ (tagging hardware + [5])
Software	Software Capabilities [83, 125]	○	Mutual	Full	●	●	●	●	Byte	∞	◓ – ● (impl. dep., incl. [5])
Software	Bounds-Checking Software [225]	●	Mutual	Full	●	●	◐	○	Byte	∞	◓ – ● (impl. dep., incl. [5])
Software	Safe Languages [50] / Software Verification [154, 163]	○	Single	Full	●	●	●	●	Byte	2 (safe/unsafe)	○ (function call)
Software	Software Fault Isolation [79, 119, 141, 142, 179, 238, 257, 262]	○	Single	Full	●	●	●	●	Byte	∞	◔ ([5])
Software	Memory Encryption / AES-NI [155]	○	Mutual	Full	●	●	○	○	128 bits	∞	◕ (copy key + encrypt + [5])

[1] In Ring 0. [2] Not all combinations of R/W/X supported. [3] Covers many granularities [138]. [4] Some works [113, 173, 190] increase it. [5] Register saving/scrubbing, stack switch.

A Model of Compartmentalization Mechanisms. At the core, a mechanism must define at least two primitives: a *protection domain* primitive, and a *communication* primitive. The former enforces isolation across protection domains and must at least guarantee integrity (§2.1). The latter must be able to transmit bits bidirectionally across compartments, and enforce compartment control-flow entry points. Communication primitives can be implemented in many ways [157]: message passing, shared memory, specialized control-flow operations (e.g., cross-compartment call). Still, a simple message passing primitive suffices for compartmentalization. `CALL`, `RETURN`, and `ASSIGN` (§3.2) can all be implemented on top of it, and `CREATE`/`DESTROY` can be left implicit. If we expand the mechanism with a third primitive to `CREATE` domains, we obtain a general compartmentalization mechanism; `DESTROY`, and other mechanism-specific primitives, may also be supported.

A mechanism may not fulfill all properties necessary to be suitable for compartmentalization, in which case we call it *conditioned*. For instance, PKU [27] is conditioned since compartment entry points cannot be enforced in without additional measures to monitor key-editing instructions [124, 232]. Other conditioned cases include PT-based protection in kernel mode [94], or bounds-checking [47, 155]. Not all mechanisms discussed next were specifically designed for compartmentalization (e.g., TEEs, bounds-checking). Still, all are either suitable or conditioned, and have been leveraged for compartmentalization in practice.

Insight 11: Many mechanisms used for compartmentalization were not designed for that aim, causing them to be conditioned. This *has important implications on how we evaluate their security or performance cost.*

Trust Models & TCB. Mechanisms themselves are designed for a given trust model (*TM* in Table 3). Here it is sufficient to distinguish between *single* distrust (covering both safebox and sandbox), and *mutual* distrust. For instance, the PT supervisor bit enforces single distrust by protecting one subject (the kernel) from others (users), whereas the page-table enforces mutual distrust by separating arbitrary sets of subjects. This constrains the trust models which abstractions can implement on top of these mechanisms (§3.2).

Mechanisms also influence the content of the TCB. In the general case (*Full* in Table 3), TCBs of compartmentalized systems include (part of) the workload, compiler, loader, system software, firmware, CPU package, and physical environment. TEEs exclude firmware and physical environment from the TCB [110, 231]. From a compartmentalization view, the TCB can be the only difference between otherwise similar mechanisms, e.g., confidential VMs vs. EPT.

SW / HW. Mechanisms can be implemented in hardware (ASICs, FPGAs, simulators) or in software. Compartmentalization advances have historically been driven by progress in hardware, which enabled separation granularities and security properties previously unreachable in software. Still, hardware is not a silver bullet: hardware takes time to reach end-devices, can be cost-prohibitive, and, as we discuss below, tends to be heterogeneous due to the lack of industry standard. Thus, in recent years, software-based approaches building on commodity hardware (MMU) have been particularly successful in popularizing compartmentalization practices (e.g., SFI with WebAssembly [119]).

Permissions Enforced. Mechanisms enforce different combinations of four primitive permissions: *read*, *write*, *execute*, and *address* (create pointers to). Subsets are common: most protection key approaches [13, 27, 53, 217] do not protect instruction fetch; PT approaches do not support all combinations of R/W/X [25]; and capabilities [78, 125] also protect addressing, which few other mechanism classes enforce. These specificities are the result of trade-offs. Security benefits from more expressiveness in permissions, and properties such as addressing also benefit interface safety by thwarting certain classes of confused deputies by construction [160]. On the other hand performance is sensitive to implementation constraints which may require to trade off security: PT entries are limited in size, and using more bits to represent more permissions requires separate tables, degrading performance; capabilities often trade off performance (e.g., through cache pressure [251]). This poses non-trivial problems to abstractions, which must map diverse levels of permission expressiveness to the previously described high-level properties (§3.2).

Enforcement Granularity. Mechanisms enforce permissions at varying memory granularities. At the extremes, enforcement may be done at the granularity of the entire physical memory [205], or at byte granularity [244]. Others (Table 3) operate on pages, words, 128-bit chunks, etc. Granularity choices too trade off performance, security, and compatibility. Whereas finer granularities approach least privilege more closely, performance and memory footprint benefit from coarser granularities due to implementation constraints: supporting finer grains implies storing more permission information, potentially increases the complexity of permission checks, or of protection domain instantiation.

Number of Domains. The domain creation primitive may restrict the maximum number of domains: protection keys support, depending on the implementation, a few [13] to thousands of [217] domains. The domain creation primitive may also not exist at all: physical separation [205] and TrustZone [14] rely on a fixed number of physically separated domains. Other mechanisms may not limit the number of domains, but scalability in performance and resource usage limit it in practice. These decisions too are trade-offs: for protection keys, the number of domains can be increased at a high performance cost [113, 190, 259].

Insight 12: Mechanisms feature very heterogeneous properties (such as the permissions enforced, the enforcement granularity, or domain count). This reinforces Insight 10 on the need to approach compartmentalization holistically.

Performance. Mechanisms impact performance in many ways: latencies of compartment switching, creation, modification, and destruction; locality and cache effects; domain-crossing sanitization costs (which mechanisms may accelerate); scalability (growing costs with domain count or size); and other mechanism-specific runtime overheads (such as memory access cost or generated code size) [136, 155]. The performance profile of mechanisms varies widely: PKU domain switches are unprivileged and thus fast, but domain creation and modification requires a costly trap [27]. With software memory encryption, domain switches are expensive (compartment encryption and decryption), but their creation and modification is an unprivileged bookkeeping operation.

There is a focus on domain switch latencies in the literature, as they often (yet not always) dominate compartmentalization performance overheads [154]. Domain switch costs result from design decisions leading to security, performance, and compatibility trade-offs: Is the switch a *privileged primitive*, i.e., do cross-domain switches require trampolines or traps to the TCB with elevated privileges? Should domain switches require trampolines at all, or should they be encoded with separate load and store instructions [109, 178]? Should switches be made faster at the expense of, e.g., compartment creation costs? How deeply can compatibility (with existing compilers, OS kernels, and programs) be broken? How generic (granularity, domain count) must the mechanism be? For these reasons, comparing mechanism domain-switch costs can be deceptive: conditioned mechanisms such as PKU [27] are fast but insufficient by themselves to guarantee safety, requiring combination with additional software techniques [113, 190] which come with costs and trade-offs of their own.

Insight 13: There is a strong focus on domain-switch latencies when evaluating mechanism performance. Yet, this disregards many other cost factors relevant in real-world deployments. Domain-switch cost comparisons are also commonly done between conditioned and unconditioned mechanisms (e.g., PKU vs. CHERI), which is unsound.

Side Channels. All software-exploitable side channels are relevant to compartmentalization threat models.

Transient execution side channels, starting with Spectre [151] and Meltdown [166], demonstrated the ability to break the confidentiality and integrity (and thus availability) properties of mechanisms such as the page table [75], Intel PKU [75], Intel MPX [75], ARM PAC [199], or ARM TrustZone [82]. For instance, Meltdown-PK [75] allows attackers to fully bypass Intel PKU. Software-based compartmentalization mechanisms are equally vulnerable [81, 139, 182], enabling attackers to bypass the confidentiality and integrity properties of WebAssembly, eBPF, and others [81].

Side channels with sequential execution semantics, such as timing side channels [152] or cache-timing side channels [256], as well as software-exploitable power consumption side channels [153], can also be leveraged to bypass the confidentiality properties of compartmentalization.

Many side-channel mitigations can be applied at the mechanism-level (P3). In the general case, this may be done through sharing less hardware resources [209]. For transient execution side channels, solutions partly consist in microcode updates to existing hardware [75], fixing existing hardware designs [75], or designing entirely new hardware mechanisms resilient to side channels [153, 183]. Some mitigations to transient execution side channels, e.g., Spectre attacks, cannot be achieved in hardware. Approaches employ ad-hoc compiler-based mitigations [215], pointer masking [139, 182], or combinations of software and hardware mechanisms [182]. All come at a sizable performance cost [67] which should be factored into broader mechanism performance considerations (Insight 13).

Still, not all mitigations to software-exploitable side channels can be done at the level of the mechanism. Other mitigations include designing applications themselves to break the correlation between side channels and secrets with constant-time programming techniques [64, 135, 245] made robust to transient execution attacks [80], or limiting attackers' measurement abilities through, e.g., reducing the resolution of timers or designing APIs that impede profilinglike behavior [189]. These too come at a performance cost and must be implemented at the level of P1 and P2.

Insight 14: The problem of side channels spans the entire P1-P3 stack. Counter-measures in compartmentalization remain in their infancy: new mechanisms commonly scope out side channels [57, 113, 138, 216, 259], and the problem is widely considered as orthogonal to P1 and P2 [105, 133, 145, 195]. This calls for more research on combined compartmentalization and side-channels topics to reach side-channel resilience throughout P1, P2, and P3.

4. Deployed Compartmentalized Software

How does the vast state of the art systematized in §3 translate in practice? To answer this, we discuss a corpus of 61 mainstream compartmentalized programs (Table 4). We constitute the corpus via a systematic search in the Debian archive [1] (56 apps), completed with previous works (§3, 3 programs), and our own expertise (2 programs). For the former, we manually triage all Debian packages with >1K installations [2] (1,520 apps), whose source-code feature privilege-separation keywords. For clarity we group programs in 13 classes, characterized using our taxonomy (§3). Interested readers can find the full list of keywords and programs in Appendix A.2. We present our insights next.

Software compartmentalization is (still) not the norm. As Table 4 shows, compartmentalized designs started gaining mainstream awareness in the mid-2000s with programs such as qmail [68, 120], OpenSSH [198], or Postfix [121]. Compartmentalization progressed since then, driven by the challenges of the web, as well as by the OpenBSD and academic communities. Still, today compartmentalized designs remain a minority (<56 out of 1,520 apps), tied to security-aware vendors (8 / 13 classes in Table 4 are authored by academics or security professionals). *Non-expert developers, even of popular software, still do not commonly compartmentalize.*

With skills and time, retrofitting is realistic. Partly due to the OpenBSD community, cases where separation was retrofitted outnumber those architected with separation in mind (Table 4). This shows that retrofitting is realistic even in service-critical, long-established codebases such as OpenSSH, V8, or Firefox. In cases, the deployment of compartmentalization schemes caused functional regressions, e.g., due to overly restrictive policies [8, 42]. Described in §3.1, these issues are a concern when deploying policies obtained manually or dynamically, *which is accepted by practitioners in Table 4*. Dunlap [103] covers this in details.

Performance matters. Hardware historically imposed a heavy performance tax to compartmentalized software. This explains why most of Table 4 came together with faster hardware in the 2000s. A textbook case, the Windows NT 3.x kernel compartmentalized its graphics stack in the 90s under the influence of microkernels, but soon reverted this in 4.0 due to performance [37]. Still today, overheads are decisive when shipping compartmentalization schemes to production [45]. This is reflected in research, with a majority of performance-focused works throughout §3.

Separation is effective but vulnerable. Concrete impact on bug exploitability has been documented where compartmentalization was pushed to production – most of Table 4. In the OpenBSD userland, the OpenSSH and slaacd sandbox compartments successfully mitigated code-execution flaws [3, 4, 19, 41]. Similar observations were made for Nginx workers [5, 6], and web browser site isolation. Still, the protection is not limitless: interface safety vulnerabilities were reported against OpenSSH [17, 39, 40], Firefox [144, 218], Nginx [7], Chrome and generally site isolation [16]. These observations are more or less direct effects of the ad-hoc nature of deployed compartmentalization approaches. Although some works in §3 are concerned with the problem of interface safety, most scope it out to focus on performance. This constitutes a gap between mainstream needs and research trends, which is important: should compartmentalization become widespread, interface safety flaws will be the main vulnerabilities of tomorrow.

PDMs are vastly manual. For all of Table 4, separation boundaries are manually identified and maintained organically over time, a process costly in expertise and efforts [15, 44]. As discussed in §3.1, the engineering cost of manual separation limits achievable separation granularities (for nearly all 61 apps, separation is coarse with less than two to three domains), and makes separation less approachable by the mainstream. Firefox library sandboxing, stemming from a research project, is the only case of a non-fully manual PDM [181]. The fact that so few leverage automated methods may also result from a mismatch between research goals and mainstream needs, as research does not offer automated PDMs for data-centric separations (cf. §3.1).

Diverse abstractions & Focus on interfaces. Abstractions show a clear tendency towards sandboxing of untrusted code (vs. other models from §2.1). For the rest, abstractions feature an heterogeneous mix of code- and data-centric designs, synchronous and asynchronous semantics, and message-passing-based and shared-memory-based communication. Overall great attention is dedicated towards interfaces. For instance OpenSSH leverages a custom protocol with fully serialized and checked objects [198], RLBox leverages type data to systematically check and copy objects, and Nginx leverages a very thin interface with nearly no communication from the untrusted to the trusted world. The importance of interfaces is characterized by the dominance of message-based approaches, which ease the checking of interface-crossing data and control flows.

Importance of availability. Most designs in Table 4 target a degree of availability. This comes in contrast with research, which generally considers availability out of scope (§3.2). This may make it difficult to deploy many of the previously described approaches under real-world expectations.

Mechanisms are page-table-centric. Only two programs from Table 4 do not rely on the page table: Google V8, which leverages SFI, and Firefox library sandboxing, a product of modern research [181] which is mechanism-agnostic. All other designs strongly depend on the page table, an effect of their building atop `fork()` semantics. Regrettably, this dependency is hard to break [66], making it difficult to

TABLE 4: *Characteristics of Mainstream Compartmentalized Software.* Abbreviations are the same as in Tables 1 to 3.

Software Class (Full list in Appendix A.2)	*Author Profile*[1]	*PDM* (§3.1)	*Abstraction (§3.2)* *Name*	*Trust Model*	*Subjects*	*Semantics*	*Properties*	*Granularity*	*Mechanism* (§3.3)
Google V8 [23]	Sec. Pro.	○[3]	Custom	Sandbox	*Code*-centric	$\mathcal{S}$, SHM	CI	Coarse (High-level components)	*Fixed*: SFI
Browser Site Isolation [85, 200, 240]	Sec. Pro.		Site Isolation		*Data*-centric	$\mathcal{A}$, *both*	CIA		*Fixed*: PT (Due to a dependency to `fork()` semantics)
OpenBSD Privilege-Separated Userland (>40 apps[2] incl. OpenSSH) [38]	Sec. Pro.		Custom (Process-Based)	(*mainly*) Sandbox	(*mainly*) *Code*-centric	$\mathcal{S}$, MES	CI(A)		
IRSSI [31]	Academic			Safebox	*Code*-centric	$\mathcal{S}$, MES	CI		
Debian `man` [34]	Other			Sandbox	*Code*-centric	$\mathcal{S}$, MES	CI		
Wireshark [52]	Sec. Pro.				*Code*-centric	$\mathcal{A}$, MES	CI		
DHCPCD [18]	Other				*Data*-centric	$\mathcal{A}$, MES	CIA		
VSFTPD [51]	Sec. Pro.	○			*Data*-centric	$\mathcal{S}$, MES	CI		
qmail [68, 120], Postfix [121], djbdns [252]	Sec. Pro.				*Code*-centric	$\mathcal{A}$, MES	CIA		
Dovecot [20]	Other				*Code*-centric	$\mathcal{A}$, MES	CIA		
Web Servers [11, 24, 33]	Other				*Data*-centric	$\mathcal{A}$, *both*	CIA		
Microkernels [106, 126, 150, 164, 258] [...]	Academic		Microkernel	Mutual Distrust	*Code*-centric	*both*, MES	CIA		*Fixed*: PT[5]
Firefox Library Sandboxing [36]	Academic	◔[3,4]	RLBox	Sandbox	*Code*-centric	$\mathcal{S}$, SHM	CIA	Finer (Libraries)	*Flexible*

[1]Sec. Pro. = Security Professional, [2]Including OpenSSH/NTPd/SMTPd, and others – see Appendix A.2, [3]Separation was retrofitted, [4]RLBox, [5]Alternatives in research [114].

reap the benefits of the modern mechanisms we discuss in §3.3: to leverage an intra-address-space compartmentalization mechanism such as Intel MPK, program code and data structures have to be redesigned to eliminate reliance on `fork()`'s transparent address-space copy semantics. Taking the example of OpenSSH (cf. Table 4), which forks an unprivileged process to perform its pre-authentication phase, this may require substantial changes, e.g., identifying which data is required by the unprivileged child, copying it explicitly, and refactoring the pre-authentication phase code to function with the explicit copies.

5. Outstanding Challenges

We conclude by consolidating the insights gained throughout this paper into high-level challenges which we believe should be solved to mainstream modern developments in compartmentalization and foster adoption.

Challenge 1: solving P1-P3 in isolation results in unsuited solutions. Because software compartmentalization is such a large and complex problem, P1-P3 are rarely, if ever, solved as a whole. Unfortunately, as we show throughout this paper, this creates friction across the compartmentalization stack as approaches are seldom composable: designing policies (or tools to generate policies) which cannot be represented efficiently with existing abstractions, abstractions which do not map to enforcement methods, or hardware which does not efficiently enforce typical partitioning needs.

For example, most abstractions used in the mainstream are strongly tied to processes and `fork()` semantics (§4). This is a problem as these semantics 1) do not compose, making it hard to use them in conjunction with new compartmentalization abstractions, and 2) hinder the ability to leverage new mechanisms, which are released at a fast pace (§3.3). Yet most *new* abstractions proposed in research still do not compose, and specialize towards specific mechanisms (§3.2). Will we repeat the mistake of `fork()` [66], requiring each codebase to implement several compartmentalization approaches? We need more concerted efforts across the stack towards generic abstractions that compose and map to the ecosystem of existing and future mechanisms.

Another instance of this problem is the performance cost of compartmentalization. It remains exceedingly hard to quantify the real performance overhead of compartmentalization because research approaches it narrowly: the focus on domain-switching latencies causes other less studied costs such as those induced by mechanisms (§3.3), the runtime costs of interface protection (§3.2) and system call shielding (§3.2), or of allocator hardening when heaps are shared, to be left aside. This calls for more efforts to better characterize the performance costs of compartmentalization, and towards techniques and tooling to support developers in estimating, diagnosing, and optimizing compartmentalization performance costs early on and across P1, P2, and P3.

Challenge 2: creating and maintaining safe compartmentalization policies is still too hard. The skills required to design safe policies are very specific: ensuring interface safety, reasoning about the performance of compartmentalized software, maintaining compartmentalizations over time; all constituting an art mastered by trial and error. Compartmentalization cannot go mainstream expecting non-expert developers to acquire this art. As we show, this makes many approaches described in §3.2 rather unsuitable for that purpose. In fact, even when developers posses the required skills, compartmentalization policies are still overly error-prone. For instance, most of the works described in §3.2 leave the job of securing internal and external interfaces, a complex and particularly error-prone task, entirely to the developer. As we observe in §4, some software projects have the skills, time, and budget to do so, but this is not the case of the vast majority of the software ecosystem.

This calls for concerted efforts in two directions. First, more work is needed *on approaches that do not require a policy from application developers*. This can be approached with a focus on third-party dependencies that have the skills and community, and where costs get amortized. Shared library and software package APIs, for instance, should be designed from the ground up to be transparently distrusted, following the example of, e.g., V8 (§4). This can also be approached through more works on automated, generic compartmentalization, trading off security for deployability (§3.1). Second, more work is needed on *supporting the policy development process, for developers who have the skills to do so*. There is a need for PDMs which can understand application and boundary semantics; for more tooling to integrate compartmentalization into long-term maintenance workflows to ensure safe separation over time; for more automated interface safety checking methods; and for more fuzzing targeted at the needs of compartmentalization.

Challenge 3: threat models are insufficiently challenged. Compartmentalization works largely make assumptions about interfaces (adequately hardened, free of high-level logic bugs), compilers (no correctness-security gap [102, 254]), shared components (libc, threading libraries, memory allocators, are bug-free when shared), the kernel (no confused deputies), or the hardware (the requirements of conditioned mechanisms are satisfied, no side-channels). *These assumptions do not hold in practice.* Interfaces are porous and abstractions must be involved to ensure safety by construction [91, 160, 213] (§3.2). Compilers [141, 142, 233], shared components and kernels [91, 237], hardware [75, 151, 166, 189] (§3.3) are all prone to separation-threatening flaws. We need more offensive research exploring gaps in compartmentalization threat models, characterizing and quantifying their impact, and defensive works with holistic threat models to achieve truly systematic counter-measures.

Challenge 4: compartmentalization research deviates from the needs of the mainstream. From analyzing the gap between research (§3) and practice (§4), we observe clear discrepancies between the focus of compartmentalization in research and what practitioners run in production. There are, for instance, very few works on availability in the compartmentalization literature, yet this is what most deployed compartmentalized programs target. A similar observation can be made with subject selections: there is no research work on PDMs for data-centric compartmentalization, although it is popular among practitioners. Research on compartmentalization abstractions also developed a strong focus synchronous semantics and shared memory in recent years, but practitioners take much more diverse approaches mixing synchronous and asynchronous semantics, message passing, and shared memory. Lastly, research explores many hardware-specific solutions, but practitioners need approaches that run on commodity hardware as well, indicating that more efforts should be put into compartmentalization schemes across P1-P3 that can deal with hardware heterogeneity. This broad gap between research and practical compartmentalization concludes this SoK motivating for many research avenues in these under-explored areas.

6. Related Works

Shu at al. [222] surveys general isolation, and Acar at al. [54] systematizes general Android security research, including application compartmentalization. Both have a wider scope than this SoK (cf. §2) and thus do not cover compartmentalization as systematically. Both also predate many recent works: most of Tables 1/2 appeared post-2016.

Other works cover vulnerability classes mitigated by compartmentalization such as memory safety [230], side-channels [81], and supply-chain [156] attacks, as well as mechanisms suitable for compartmentalization [82, 131, 138]. These works are orthogonal to this paper. Others [85, 129, 146, 160, 162] classify interface safety issues and mitigations. These complement this SoK, which draws a bigger picture of compartmentalization challenges. Sammler et al. [210] models sandboxing to prove safety properties. These efforts motivate this SoK, confirming both the validity as well as the limitations of their model.

7. Conclusion

Despite its benefits and decades of research and industry works, compartmentalization remains a niche software engineering practice. Through a systematic study of 211 software compartmentalization works and 61 deployed approaches, this paper sheds a light on the strengths and limitations of current compartmentalization knowledge. We stress that popularizing software compartmentalization will require progress towards a more holistic approach to compartmentalization; towards facilitating the definition of compartmentalization policies; towards stronger and more holistic threat models in the light of confused deputies and hardware limitations; as well as towards more attention to the gaps between research and production approaches.

Acknowledgments

We are grateful to the anonymous reviewers and to our anonymous shepherd, as well as to Peter Pietzuch, Mikel Luján, Shravan Narayan, Sid Agrawal, and Aastha Mehta for their insightful feedback. This work was partly funded by a studentship from NEC Labs Europe, a Microsoft Research PhD Fellowship, the EPSRC grants EP/V012134/1 (UniFaaS), EP/V000225/1 (SCorCH), and EP/X015610/1 (FlexCap), and the NSF CNS grants #2008867, #2146537.

References

[1] https://packages.debian.org. Viewed 24/03/23.
[2] https://popcon.debian.org/. Viewed 24/03/23.
[3] https://nvd.nist.gov/vuln/detail/CVE-2022-27882.
[4] https://nvd.nist.gov/vuln/detail/CVE-2022-27881.
[5] https://nvd.nist.gov/vuln/detail/CVE-2021-23017.
[6] https://nvd.nist.gov/vuln/detail/CVE-2022-41743.
[7] https://nvd.nist.gov/vuln/detail/CVE-2016-1247.
[8] Allow writev in sandbox. https://github.com/openssh/openssh-portable/commit/6283f4bd83ee. Viewed 23/03/23.
[9] AMD® PRO Security: AMD Secure Processor. https://www.amd.com/en/technologies/pro-security. Viewed 08/02/23.
[10] AMD® Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/. Viewed 08/02/23.
[11] Apache MPM worker. https://httpd.apache.org/docs/2.4/mod/worker.html. Viewed 18/03/23.
[12] Application Data Integrity (ADI). https://www.kernel.org/doc/Documentation/sparc/adi.rst. Viewed 06/04/23.
[13] ARM® Cortex-R4/R4F MPU. https://developer.arm.com/documentation/ddi0363/g/Memory-Protection-Unit/About-the-MPU. Viewed 08/02/23.
[14] ARM® TrustZone for Cortex-A. https://www.arm.com/technologies/trustzone-for-cortex-a. Viewed 08/02/23.
[15] BSDCan 2018: slaacd(8). https://www.openbsd.org/papers/florian_slaacd_bsdcan2018.pdf. Viewed 24/03/23.
[16] Chrome: Site Isolation bypass via NavigationPreloadRequest. https://bugs.chromium.org/p/project-zero/issues/detail?id=2239. Viewed 24/09/23.
[17] CVE-2006-5794: Sandbox escape via auth bypass in OpenSSH. https://nvd.nist.gov/vuln/detail/CVE-2006-5794. Viewed: 25/09/23.

[18] DHCPCD: DHCP and DHCPv6 client. https://roy.marples.name/projects/dhcpcd/. Viewed 24/03/23.

[19] double-free vulnerability in OpenSSH (CVE-2023-25136). https://seclists.org/oss-sec/2023/q1/92. Viewed 24/03/23.

[20] Dovecot processes. https://doc.dovecot.org/developer_manual/design/processes/#dovecot-processes. Viewed 25/09/23.

[21] Enhanced security through MTE. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhanced-security-through-mte. Viewed: 06/04/2023.

[22] Google: Code Sandboxing: SAPI. https://developers.google.com/code-sandboxing/sandboxed-api. Viewed 22/02/23.

[23] Google V8 untrusted code mitigations. https://v8.dev/docs/untrusted-code-mitigations. Viewed 25/09/23.

[24] Inside NGINX. https://nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale/. Viewed 18/03/23.

[25] Intel® 64 and IA-32 Arch. SDM. Vol. 3A, Sec. 4.6.1.: ACCESS RIGHTS: Determination of Access Rights. Version: December 2022.

[26] Intel® 64 and IA-32 Arch. SDM. Vol. 3C, Sec. 26.5.6: FEATURES SPECIFIC TO VMX NON-ROOT OPERATION: VM Functions. Version: December 2022.

[27] Intel® 64 and IA-32 Arch. SDM. Vol. 3A, Sec. 4.6.2.: ACCESS RIGHTS: Protection Keys. Version: December 2022.

[28] Intel® SGX. https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html. Viewed 08/02/23.

[29] Intel® TDX. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html. Viewed 08/02/23.

[30] Introducing Arm CCA V1. https://developer.arm.com/documentation/den0125/0100/Arm-CCA-extensions. Viewed 08/02/23.

[31] Irssi Chat Client. https://irssi.org/. Viewed 24/03/23.

[32] ISO/IEC 2382:2015 (entry #2121372): *program*. https://www.iso.org/obp/ui/en/#iso:std:iso-iec:2382:ed-1:v2:en. Viewed 24/01/24.

[33] Lighttpd Angel Configuration. https://doc.lighttpd.net/lighttpd2/core_config_angel.html. Viewed 18/03/23.

[34] man-db. https://man-db.nongnu.org/. Viewed 24/03/23.

[35] Microsoft Monza. https://github.com/microsoft/monza. Viewed 22/02/23.

[36] Mozilla RLBox. https://github.com/mozilla/gecko-dev/tree/master/third_party/rlbox. Viewed 18/03/23.

[37] MS Windows NT kernel-mode user and GDI white paper. https://learn.microsoft.com/en-us/previous-versions//cc750820(v=technet.10). Viewed 25/09/23.

[38] OpenBSD Innovations. https://www.openbsd.org/innovations.html. Viewed 24/03/23.

[39] OpenSSH 7.0 Fixing Sandbox Escape. https://www.openssh.com/txt/release-7.0. Viewed 24/03/23.

[40] OpenSSH 7.4 Fixing Sandbox Escape. https://www.openwall.com/lists/oss-security/2016/12/19/2. Viewed 24/03/23.

[41] OpenSSH 9.0: near miss in sshd(8). https://www.openbsd.org/71.html. Viewed 24/09/23.

[42] OpenSSH breaking due to overly restrictive syscall policy. https://github.com/NixOS/nixpkgs/issues/157451. Viewed 23/10/03.

[43] pledge. https://man.openbsd.org/pledge.2. Viewed 24/03/23.

[44] Privilege Separation. https://www.openbsd.org/papers/auug04/mgp00030.html. Viewed 24/03/23.

[45] Sandbox libexpat using RLBox (incl. performance discussion). https://bugzilla.mozilla.org/show_bug.cgi?id=1688452. Viewed 25/09/23.

[46] Seccomp BPF. https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html. Viewed 18/03/23.

[47] Support for Intel® MPX. https://www.intel.com/content/www/us/en/support/articles/000059823/processors.html. Viewed 08/02/23.

[48] sysfs: filesystem for exporting kernel objects. https://www.kernel.org/doc/html/latest/filesystems/sysfs.html. Viewed 18/03/23.

[49] The /proc Filesystem. https://www.kernel.org/doc/html/latest/filesystems/proc.html. Viewed 18/03/23.

[50] The Rust language. www.rust-lang.org/. Viewed 22/02/23.

[51] vsftpd: secure and fast FTP server for UNIX-like systems. https://security.appspot.com/vsftpd.html. Viewed 24/03/23.

[52] Wireshark privilege separation. https://wiki.wireshark.org/Development/PrivilegeSeparation. Viewed 23/05/24.

[53] z/OS concepts: What is storage protection? https://www.ibm.com/docs/en/zos-basic-skills?topic=storage-what-is-protection. Viewed 08/02/23.

[54] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith. SoK: Lessons learned from Android security research for appified software platforms. S&P'16.

[55] H. Almatary, M. Dodson, J. Clarke, P. Rugg, I. Gomes, M. Podhradsky, P. G. Neumann, S. W. Moore, and R. N. M. Watson. CompartOS: CHERI Compartmentalization for Embedded Systems. *arXiv preprint arXiv:2206.02852*, 2022.

[56] J. Alves-Foss, P. Oman, C. Taylor, and S. Harrison. The MILS architecture for high-assurance embedded systems. *Int. J. Embed. Syst.*, 2, 2006.

[57] S. Amar, D. Chisnall, T. Chen, N. W. Filardo, B. Laurie, K. Liu, R. Norton, S. W. Moore, Y. Tao, R. N. M. Watson, and H. Xia. CHERIoT: Complete memory safety for embedded devices. MICRO'23.

[58] S. J. Ames, R. Schell, and M. Gasser. Security kernel design and implementation: An introduction. *Computer*, 16(07), 1983.

[59] J. P. Anderson. Computer security technology planning study, 1972.

[60] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing Memory Robustly in Message-Passing Systems. *Journal of the ACM*, 42(1), 1995.

[61] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock Android. USENIX Security'15.

[62] I. Bang, M. Kayondo, H. Moon, and Y. Paek. TRUST: A compilation framework for in-process isolation to protect safe Rust against untrusted code. USENIX Security'23.

[63] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. SOSP'03.

[64] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie. System-level non-interference for constant-time cryptography. CCS'14.

[65] M. Bauer and C. Rossow. Cali: Compiler assisted library isolation. AsiaCCS'21.

[66] A. Baumann, J. Appavoo, O. Krieger, and T. Roscoe. A fork() in the road. HotOS'19.

[67] J. Behrens, A. Belay, and M. F. Kaashoek. Performance evolution of mitigating transient execution attacks. EuroSys'22.

[68] D. J. Bernstein. Some thoughts on security after ten years of qmail 1.0. CSAW'07.

[69] S. Biggs, D. Lee, and G. Heiser. The jury is in: Monolithic OS design is flawed: Microkernel-based designs improve security. APSys'18.

[70] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting applications into reduced-privilege compartments. NSDI'08.

[71] K. Boos, E. D. Vecchio, and L. Zhong. A characterization of state spill in modern operating systems. EuroSys'17.

[72] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in Linux. ATC'10.

[73] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. USENIX Security'04.

[74] N. Burow, X. Zhang, and M. Payer. SoK: Shining light on shadow stacks. S&P'19.

[75] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A systematic evaluation of transient execution attacks and defenses. USENIX Security'19.

[76] C. Canella, M. Werner, D. Gruss, and M. Schwarz. Automating seccomp filter generation for Linux applications. CCSW'21.

[77] S. A. Carr and M. Payer. Datashield: Configurable data confidentiality and integrity. AsiaCCS'17.

[78] N. P. Carter, S. W. Keckler, and W. J. Dally. Hardware support for fast capability-based addressing. ASPLOS'94.

[79] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. SOSP'09.

[80] S. Cauligi, C. Disselkoen, K. v. Gleissenthall, D. Tullsen, D. Stefan, T. Rezk, and G. Barthe. Constant-time foundations for the new Spectre era. PLDI'20.

[81] S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, and D. Stefan. SoK: Practical foundations for software Spectre defenses. S&P'22.

[82] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto. SoK: Understanding the prevailing security vulnerabilities in TrustZone-assisted TEE systems. S&P'20.

[83] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst.*, 12(4), 1994.

[84] S. Checkoway and H. Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. ASPLOS'13.

[85] S. Chen, D. Ross, and Y.-M. Wang. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. CCS'07.

[86] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. ASPLOS'08.

[87] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu. Shreds: Fine-grained execution units with private memory. S&P'16.

[88] Y. Chien, V.-A. Bădoiu, Y. Yang, Y. Huo, K. Kaoudis, H. Lefeuvre, P. Olivier, and N. Dautenhahn. CIVSCOPE: Analyzing potential memory corruption bugs in compartment interfaces. KISV'23.

[89] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. SOSP'07.

[90] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer. ACES: Automatic compartments for embedded systems. USENIX Security'18.

[91] R. J. Connor, T. McDaniel, J. M. Smith, and M. Schuchard. PKU pitfalls: Attacks on PKU-based memory isolation systems. USENIX Security'20.

[92] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. USENIX Security'16.

[93] R. C. Daley and J. B. Dennis. Virtual memory, processes, and sharing in Multics. *Commun. ACM*, 11(5), 1968.

[94] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. ASPLOS'15.

[95] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis. sysfilter: Automated system call filtering for commodity software. RAID'20.

[96] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5), 1976.

[97] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3), 1966.

[98] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: Architectural support for spatial safety of the C programming language. ASPLOS'08.

[99] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, B. C. Pierce, and A. DeHon. Architectural support for software-defined metadata processing. ASPLOS'15.

[100] E. W. Dijkstra. The structure of the "THE"-multiprogramming system. *Commun. ACM*, 11(5), 1968.

[101] X. Dong, H. Hu, P. Saxena, and Z. Liang. A quantitative evaluation of privilege separation in web browser designs. ESORICS'13.

[102] V. D'Silva, M. Payer, and D. Song. The correctness-security gap in compiler optimization. IEEE SPW'15.

[103] T. Dunlap, W. Enck, and B. Reaves. A study of application sandbox policies in Linux. SACMAT'22.

[104] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman. The matter of heartbleed. IMC'14.

[105] K. D. Duy, K. Cho, T. Noh, and H. Lee. Capacity: Cryptographically-enforced in-process capabilities for modern ARM architectures. CCS'23.

[106] K. Elphinstone and G. Heiser. From L3 to SeL4 what have we learnt in 20 years of L4 microkernels? SOSP'13.

[107] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. OSDI'06.

[108] L. G. Esswood. *CheriOS: designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor*. PhD thesis, U. of Cambridge, 2021.

[109] T. Frassetto, P. Jauernig, C. Liebchen, and A.-R. Sadeghi. IMIX: In-process memory isolation extension. USENIX Security'18.

[110] X. Ge, H.-C. Kuo, and W. Cui. Hecate: Lifting and shifting on-premises workloads to an untrusted cloud. CCS'22.

[111] A. Ghosn, M. Kogias, M. Payer, J. R. Larus, and E. Bugnion. Enclosure: Language-based restriction of untrusted libraries. ASPLOS'21.

[112] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2. USITS'97.

[113] J. Gu, H. Li, W. Li, Y. Xia, and H. Chen. EPK: Scalable and efficient memory protection keys. ATC'22.

[114] J. Gu, X. Wu, W. Li, N. Liu, Z. Mi, Y. Xia, and H. Chen. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. ATC'20.

[115] J. Gu, B. Zhu, M. Li, W. Li, Y. Xia, and H. Chen. A hardware-software co-design for efficient intra-enclave isolation. USENIX Security'22.

[116] L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. S&P'15.

[117] K. Gudka, R. N. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson. Clean application compartmentalization with SOAAP. CCS'15.

[118] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in Xen. Middleware'06.

[119] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with WebAssembly. PLDI'17.

[120] M. Hafiz, R. Johnson, and R. Afandi. The security architecture of qmail. PLoP'04.

[121] M. Hafiz and R. E. Johnson. Evolution of the MTA architecture: the impact of security. *Software: Practice and Experience*, 38(15), 2008.

[122] P. B. Hansen. The nucleus of a multiprogramming system. *Commun. ACM*, 13(4), 1970.

[123] N. Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4), 1988.

[124] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. ATC'19.

[125] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke. The Mungi single-address-space operating system. *Software: Practice and Experience*, 28(9), 1999.

[126] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev.*, 40(3), 2006.

[127] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: Secure applications on an untrusted operating system. ASPLOS'13.

[128] T. C.-H. Hsu, K. Hoffman, P. Eugster, and M. Payer. Enforcing least privilege memory views for multithreaded applications. CCS'16.

[129] H. Hu, Z. L. Chua, Z. Liang, and P. Saxena. Identifying arbitrary memory access vulnerabilities in privilege-separated software. ESORICS'15.

[130] J. Hu, A. Iosifescu, and R. LiKamWa. LensCap: Split-process framework for fine-grained visual privacy control for augmented reality apps. MobiSys'21.

[131] W. Hu, A. Ardeshiricham, and R. Kastner. Hardware information flow tracking. *ACM Comput. Surv.*, 54(4), may 2021.

[132] J. Huang, O. Schranz, S. Bugiel, and M. Backes. The ART of app compartmentalization: Compiler-based library privilege separation on stock Android. CCS'17.

[133] Y. Huang, V. Narayanan, D. Detweiler, K. Huang, G. Tan, T. Jaeger, and A. Burtsev. KSplit: Automating device driver isolation. OSDI'22.

[134] Z. Huang, T. Jaeger, and G. Tan. Fine-grained program partitioning for security. EuroSec'21.

[135] J. Jancar, M. Fourné, D. D. A. Braga, M. Sabt, P. Schwabe, G. Barthe, P.-A. Fouque, and Y. Acar. "they're not that hard to mitigate": What cryptographic library developers think about timing attacks. S&P'22.

[136] A. Jangda, B. Powers, E. D. Berger, and A. Guha. Not so fast: Analyzing the performance of Webassembly vs. native code. ATC'19.

[137] I. R. Jenkins, P. Anantharaman, R. Shapiro, J. P. Brady, S. Bratus, and S. W. Smith. Ghostbusting: Mitigating Spectre with intraprocess memory isolation. HotSoS'20.

[138] S. Jero, N. Burow, B. Ward, R. Skowyra, R. Khazan, H. Shrobe, and H. Okhravi. TAG: Tagged architecture guide. *ACM Comput. Surv.*, 55(6), dec 2022.
[139] D. Jin, A. J. Gaidis, and V. P. Kemerlis. BeeBox: Hardening BPF against transient execution attacks. USENIX Security'24.
[140] X. Jin, X. Xiao, S. Jia, W. Gao, D. Gu, H. Zhang, S. Ma, Z. Qian, and J. Li. Annotating, tracking, and protecting cryptographic secrets with CryptoMPK. S&P'22.
[141] E. Johnson, E. Laufer, Z. Zhao, D. Gohman, S. Narayan, S. Savage, D. Stefan, and F. Brown. WaVe: a verifiably secure WebAssembly sandboxing runtime. S&P'23.
[142] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan. SFI safety for native-compiled Wasm. NDSS'21.
[143] A. Jung, H. Lefeuvre, C. Rotsos, P. Olivier, D. Oñoro Rubio, F. Huici, and M. Niepert. Wayfinder: Towards automatically deriving optimal OS configurations. APSys'21.
[144] C. Kerschbaumer and C. Holler. Effectively fuzzing the IPC layer in Firefox. https://blog.mozilla.org/attack-and-defense/2021/01/27/effectively-fuzzing-the-ipc-layer-in-firefox/. Viewed 25/09/23.
[145] A. Khan, D. Xu, and D. J. Tian. EC: Embedded systems compartmentalization via intra-kernel isolation. S&P'23.
[146] M. R. Khandaker, Y. Cheng, Z. Wang, and T. Wei. COIN Attacks: On insecurity of enclave untrusted interfaces in SGX. ASPLOS'20.
[147] D. Kilpatrick. Privman: A library for partitioning applications. ATC'03.
[148] Y. Kim, J. Lee, and H. Kim. Hardware-based always-on heap memory safety. MICRO'20.
[149] P. Kirth, M. Dickerson, S. Crane, P. Larsen, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, and M. Franz. PKRU-Safe: Automatically locking down the heap between safe and unsafe languages. EuroSys'22.
[150] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. SeL4: Formal verification of an OS kernel. SOSP'09.
[151] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. S&P'19.
[152] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. CRYPTO'96.
[153] A. Kogler, J. Juffinger, L. Giner, L. Gerlach, M. Schwarzl, M. Schwarz, D. Gruss, and S. Mangard. Collide+Power: Leaking inaccessible data with software-based power side channels. USENIX Security'23.
[154] M. Kolosick, S. Narayan, E. Johnson, C. Watt, M. LeMay, D. Garg, R. Jhala, and D. Stefan. Isolation without taxation: Near-zero-cost transitions for WebAssembly and SFI. *Proc. ACM Program. Lang.*, 6(POPL), 2022.
[155] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. EuroSys'17.
[156] P. Ladisa, H. Plate, M. Martinez, and O. Barais. SoK: Taxonomy of attacks on open-source software supply chains. S&P'23.
[157] B. W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1), 1974.
[158] H. Lee, C. Song, and B. B. Kang. Lord of the x86 rings: A portable user mode privilege separation architecture on x86. CCS'18.
[159] H. Lee, C. Song, and B. B. Kang. Harnessing the x86 intermediate rings for intra-process isolation. *IEEE Transactions on Dependable and Secure Computing*, 20(4), 2023.
[160] H. Lefeuvre, V.-A. Bădoiu, Y. Chien, F. Huici, N. Dautenhahn, and P. Olivier. Assessing the impact of interface vulnerabilities in compartmentalized software. NDSS'23.
[161] H. Lefeuvre, V.-A. Bădoiu, A. Jung, S. L. Teodorescu, S. Rauch, F. Huici, C. Raiciu, and P. Olivier. FlexOS: Towards flexible OS isolation. ASPLOS'22.
[162] H. Lefeuvre, D. Chisnall, M. Kogias, and P. Olivier. Towards (really) safe and fast confidential I/O. HotOS'23.
[163] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. LPAR'10.
[164] J. Liedtke. On micro-kernel construction. SOSP'95.
[165] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch. Glamdring: Automatic application partitioning for intel SGX. ATC'17.
[166] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. USENIX Security'18.
[167] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel. Light-Weight contexts: An OS abstraction for safety and performance. OSDI'16.
[168] B. Liu, B. Liu, H. Jin, and R. Govindan. Efficient privilege de-escalation for ad libraries in mobile apps. MobiSys'15.
[169] S. Liu, G. Tan, and T. Jaeger. PtrSplit: Supporting general pointers in automatic program partitioning. CCS'17.
[170] S. Liu, D. Zeng, Y. Huang, F. Capobianco, S. McCamant, T. Jaeger, and G. Tan. Program-Mandering: Quantitative privilege separation. CCS'19.
[171] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. CCS'15.
[172] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek. Software fault isolation with API integrity and multi-principal modules. SOSP'11.
[173] D. McKee, Y. Giannaris, C. O. Perez, H. Shrobe, M. Payer, H. Okhravi, and N. Burow. Preventing kernel hacks with HAKC. NDSS'22.
[174] S. Mergendahl, N. Burow, and H. Okhravi. Cross-language attacks. NDSS'22.
[175] D. Merkel. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239), 2014.
[176] A. Milburn, E. Van Der Kouwe, and C. Giuffrida. Mitigating information leakage vulnerabilities with type-based data isolation. S&P'22.
[177] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.
[178] L. Mogosanu, A. Rane, and N. Dautenhahn. Microstache: A lightweight execution context for in-process safe region isolation. RAID'18.
[179] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. RockSalt: Better, faster, stronger SFI for the x86. PLDI'12.
[180] M. J. Nam, P. Akritidis, and D. J. Greaves. Framer: A tagged-pointer capability system with memory safety applications. ACSAC'19.
[181] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan. Retrofitting fine grain isolation in the Firefox renderer. USENIX Security'20.
[182] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, and D. Stefan. Swivel: Hardening WebAssembly against Spectre. USENIX Security'21.
[183] S. Narayan, T. Garfinkel, M. Taram, J. Rudek, D. Moghimi, E. Johnson, C. Fallin, A. Vahldiek-Oberwagner, M. LeMay, R. Sahita, D. Tullsen, and D. Stefan. Going beyond the limits of SFI: Flexible and secure hardware-assisted in-process isolation with HFI. ASPLOS'23.
[184] V. Narayanan, T. Huang, D. Detweiler, D. Appel, Z. Li, G. Zellweger, and A. Burtsev. Redleaf: Isolation and communication in a safe operating system. OSDI'20.
[185] V. Narayanan, Y. Huang, G. Tan, T. Jaeger, and A. Burtsev. Lightweight kernel isolation with virtualization and VM functions. VEE'20.
[186] R. Nikolaev and G. Back. VirtuOS: An operating system with kernel virtualization. SOSP'13.
[187] R. Nikolaev, M. Sung, and B. Ravindran. LibrettOS: A dynamically adaptable multiserver-library OS. VEE'20.
[188] R. M. Norton. *Hardware Support for Compartmentalization*. PhD thesis, U. of Cambridge, 2016.
[189] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. CCS'15.
[190] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim. libmpk: Software abstraction for Intel memory protection keys (Intel MPK). ATC'19.

[191] G. Parmer and R. West. Mutable protection domains: Towards a component-based system for dependable and predictable computing. RTSS'07.

[192] G. Parmer and R. West. Mutable protection domains: Adapting system fault isolation for reliability and efficiency. *IEEE Transactions on Software Engineering*, 38(4), 2012.

[193] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12), 1972.

[194] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. AdDroid: Privilege separation for applications and advertisers in Android. AsiaCCS'12.

[195] D. Peng, C. Liu, T. Palit, P. Fonseca, A. Vahldiek-Oberwagner, and M. Vij. μSWITCH: Fast kernel context isolation with implicit context switches. S&P'23.

[196] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis. xMP: Selective memory protection for kernel and user space. S&P'20.

[197] N. Provos. Improving host security with system call policies. USENIX Security'03.

[198] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. USENIX Security'03.

[199] J. Ravichandran, W. T. Na, J. Lang, and M. Yan. PACMAN: attacking ARM pointer authentication with speculative execution. ISCA'22.

[200] C. Reis, A. Moshchuk, and N. Oskov. Site isolation: Process separation for web sites within the browser. USENIX Security'19.

[201] E. Rivera, S. Mergendahl, H. Shrobe, H. Okhravi, and N. Burow. Keeping safe Rust safe with Galeed. ACSAC'21.

[202] N. Roessler, L. Atayde, I. Palmer, D. McKee, J. Pandey, V. P. Kemerlis, M. Payer, A. Bates, J. M. Smith, A. DeHon, and N. Dautenhahn. μSCOPE: A methodology for analyzing least-privilege compartmentalization in large software artifacts. RAID'21.

[203] N. Roessler and A. DeHon. SCALPEL: Exploring the limits of tag-enforced compartmentalization. *J. Emerg. Technol. Comput. Syst.*, 18(1), 2021.

[204] N. Roessler and A. DeHon. Protecting the stack with metadata policies and tagged hardware. S&P'18.

[205] J. M. Rushby. Design and verification of secure systems. SOSP'81.

[206] J. M. Rushby. Proof of separability a verification technique for a class of security kernels. ISP'82.

[207] M. Russinovich, M. Costa, C. Fournet, D. Chisnall, A. Delignat-Lavaud, S. Clebsch, K. Vaswani, and V. Bhatia. Toward confidential cloud computing. *Commun. ACM*, 64(6), 2021.

[208] J. H. Saltzer. Protection and the control of information sharing in multics. *Commun. ACM*, 17(7), 1974.

[209] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proc. of the IEEE*, 63(9), 1975.

[210] M. Sammler, D. Garg, D. Dreyer, and T. Litak. The high-level benefits of low-level sandboxing. *Proc. ACM Program. Lang.*, 4(POPL), 2019.

[211] V. A. Sartakov, L. Vilanova, and P. Pietzuch. CubicleOS: A library OS with software componentisation for practical isolation. ASPLOS'21.

[212] H. Sasaki, M. A. Arroyo, M. T. I. Ziad, K. Bhat, K. Sinha, and S. Sethumadhavan. Practical byte-granular memory blacklisting using Califorms. MICRO'19.

[213] S. Savage and B. N. Bershad. Some issues in the design of an extensible operating system (panel statement). OSDI'94.

[214] R. R. Schell, P. J. Downey, and G. J. Popek. Preliminary notes on the design of secure military computer systems, 1973.

[215] B. Schlüter, D. Borkmann, and P. Krysiuk. BPF and Spectre: Mitigating transient execution attacks. https://popl22.sigplan.org/details/prisc-2022-papers/11/BPF-and-Spectre-Mitigating-transient-execution-attacks. Viewed 25/09/2024.

[216] D. Schrammel, S. Weiser, R. Sadek, and S. Mangard. Jenny: Securing syscalls for PKU-based memory isolation systems. USENIX Security'22.

[217] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss. Donky: Domain keys – efficient in-process isolation for RISC-V and x86. USENIX Security'20.

[218] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz. Nyx-net: Network fuzzing with incremental snapshots. EuroSys'22.

[219] J. Seo, D. Kim, D. Cho, T. Kim, and I. Shin. FlexDroid: Enforcing in-app privilege separation in Android. NDSS'16.

[220] S. Shekhar, M. Dietz, and D. S. Wallach. AdSplit: Separating smartphone advertising from applications. USENIX Security'12.

[221] L. Shi, Y. Wu, Y. Xia, N. Dautenhahn, H. Chen, B. Zang, and J. Li. Deconstructing Xen. NDSS'17.

[222] R. Shu, P. Wang, S. A. Gorski III, B. Andow, A. Nadkarni, L. Deshotels, J. Gionta, W. Enck, and X. Gu. A study of security isolation techniques. *ACM Comput. Surv.*, 49(3), 2016.

[223] L. Soares and M. Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. OSDI'10.

[224] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. HDFI: Hardware-assisted data-flow isolation. S&P'16.

[225] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. SoK: Sanitizing for security. S&P'19.

[226] R. Strackx, P. Agten, N. Avonds, and F. Piessens. Salus: Kernel support for secure process compartments. *EAI Endorsed Transactions on Security and Safety*, 2(3), 2015.

[227] M. Sun and G. Tan. Nativeguard: Protecting Android applications from third-party native libraries. WiSec '14.

[228] M. Sung, P. Olivier, S. Lankes, and B. Ravindran. Intra-unikernel isolation with Intel Memory Protection Keys. VEE'20.

[229] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. *ACM Trans. Comput. Syst.*, 24(4), 2006.

[230] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. S&P'13.

[231] C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. ATC'17.

[232] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. ERIM: Secure, efficient in-process isolation with protection keys. USENIX Security'19.

[233] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. CCS'19.

[234] M. Van Steen and A. S. Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 3 edition, 2020.

[235] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith. BreakApp: Automated, flexible application compartmentalization. NDSS'18.

[236] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. CODOMs: Protecting software with code-centric memory domains. ISCA'14.

[237] A. Voulimeneas, J. Vinck, R. Mechelinck, and S. Volckaert. You shall not (by)pass! Practical, secure, and fast PKU-based sandboxing. EuroSys'22.

[238] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. SOSP'93.

[239] K. G. Walter, S. I. Schaen, W. F. Ogden, W. C. Rounds, D. G. Shumway, D. D. Schaeffer, K. J. Biba, F. T. Bradshaw, S. R. Ames, and J. M. Gilligan. Structured specification of a security kernel. ICRS'75.

[240] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. USENIX Security'09.

[241] J. Wang, X. Xiong, and P. Liu. Between mutual trust and mutual distrust: Practical fine-grained privilege separation in multithreaded applications. ATC'15.

[242] N. Wanninger, J. Bowden, K. Shetty, and K. Hale. Isolating functions at the hardware limit with Virtines. EuroSys'22.

[243] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical capabilities for UNIX. USENIX Security'10.

[244] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. S&P'15.

[245] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan. CT-wasm: type-driven secure cryptography for the web ecosystem. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[246] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A.-R. Sadeghi. TIMBER-V: Tag-isolated memory bringing fine-grained enclaves to RISC-V. NDSS'19.

[247] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI), 2003.

[248] L. Wilke, J. Wichelmann, M. Morbitzer, and T. Eisenbarth. Sevurity: No security without integrity : Breaking integrity-free memory encryption with minimal assumptions. S&P'20.

[249] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. ASPLOS'02.

[250] C. Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. EASE '14.

[251] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. ISCA'14.

[252] C. Wright. djbdns: more than just a mouthful of consonants. *Linux Journal*, 2008(173), 2008.

[253] Y. Wu, J. Sun, Y. Liu, and J. S. Dong. Automatically partition software into least privilege components using dynamic data dependency analysis. ASE'13.

[254] J. Xu, K. Lu, Z. Du, Z. Ding, L. Li, Q. Wu, M. Payer, and B. Mao. Silent bugs matter: A study of compiler-introduced security bugs. USENIX Security'23.

[255] C. Yagemann, M. A. Noureddine, W. U. Hassan, S. Chung, A. Bates, and W. Lee. Validating the integrity of audit logs against execution repartitioning attacks. CCS'21.

[256] Y. Yarom and K. Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache Side-Channel attack. USENIX Security'14.

[257] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. S&P'09.

[258] M. Young, A. Tevanian, R. Rashid, D. Golub, and J. Eppinger. The duality of memory and communication in the implementation of a multiprocessor operating system. SOSP'87.

[259] Z. Yuan, S. Hong, R. Chang, Y. Zhou, W. Shen, and K. Ren. VDom: Fast and unlimited virtual domains on multiple architectures. ASPLOS'23.

[260] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik. APISan: Sanitizing API usages through semantic Cross-Checking. USENIX Security'16.

[261] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *ACM Trans. Comput. Syst.*, 20(3), 2002.

[262] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. CCS'11.

[263] X. Zhang, A. Ahlawat, and W. Du. AFrame: Isolating advertisements from mobile applications in Android. ACSAC '13.

Appendix A. Methodology

A.1. Taxonomy Appendix (complements §3)

We include a paper if it fits compartmentalization as defined in §2. We consider venues ranked A* by CORE 2023 in security, systems, and programming languages: S&P, USENIX Security, CCS, NDSS, ASPLOS, OSDI, SOSP, PLDI. We manually triage titles and abstracts in an *ablative, conservative manner*: when processing titles, papers clearly unrelated to compartmentalization are discarded. Abstracts are inspected if the title does not enable an unambiguous decision, and are sufficient to make an unambiguous decision in the most cases. We analyze the content of papers to determine if the work address one of P1-P3.

Researchers also published influential works pre-2003, in other (types of) venues or communities, or in the industry. We aim to cover these as well. We do this with a recursive search in the references called *snowballing* [250], and by factoring in works from our own expertise. For the recursive reference search, we extract the references of all 96 works identified previously and repeat the filtering procedure.

A.2. Source Search Appendix (complements §4)

The keywords considered for the automated search of package source-code in the Debian archive are: *sandbox, privilege, isolat*, separat*, compartment, partition, domain, capabilit** (and derived words, e.g., partition*(ed|ing)*, separat*(ed|ion)*). This yields 361 packages, which we manually triage to determine if they implement compartmentalization, resulting in 19 packages. The many false positives are composed of software that do whole application sandboxing [46, 103], drop privileges (whole application least privilege), implement internal user access-control policies that do not qualify as compartmentalization (e.g., databases), or use isolation or separation keywords to refer to other development practices (e.g., "isolate a component in a class", "protected" methods in C++). We look up the vendor website of each of the 19 packages for software from the same vendor that also has a compartmentalized design (missed in our initial search because less popular or not packaged in Debian), resulting in 37 more compartmentalized programs, most from the OpenBSD userland, totaling 56 packages. We complete this list with previous works (§3, 3 programs), and our knowledge of the field (2 programs), to reach a corpus of 61 mainstream compartmentalized programs.

Choice of the Debian Archive. We choose the Debian archive because it 1) covers open-source software from all origins, 2) is very large (59K+ packages [1]), and 3) comes with Popcon [2], a popularity metric which maps to our "mainstream" criteria to make the search practicable.

We recognize that a systematic search in the Debian archive does not cover compartmentalized software in the mobile ecosystem (e.g., Android or iOS). Beyond the difficulty to integrate it in the paper's space, a systematic search of mobile applications for compartmentalization patterns is difficult due to the unavailability of sources in popular application stores. We therefore leave it for future works.

Program List. *Labels.* From our systematic search: [S]; From related works: [R]; From our field knowledge: [K].

- *OpenBSD Privilege-Separated Userland*: openssh[S], bgpd[S] (openbgpd), dhclient[S], dhcpd[S], dvmrpd[S], eigrpd[S], file[S], httpd[S], iked[S], ldapd[S], ldpd[S], mountd[S], npppd[S], ntpd[S] (openntpd), ospfd[S], ospf6d[S], pflogd[S], radiusd[S], relayd[S], ripd[S], script[S], smtpd[S] (opensmtpd), syslogd[S], tcpdump[S], tmux[S], xconsole[S], xdm[S], Xserver[S] (Xenocara), ypldap[S], pkg_add[S], xlock[S], snmpd[S], dhcrelay[S], rbootd[S], pppoe[S], mopd[S], afsd[S], rdate[S] (openrdate), sndiod[S], isakmpd[S], named[S], acme-client[S].
- *Web Servers*: Apache HTTPd[S], Nginx[S], Lighttpd[S]
- *Browser Site Isolation*: Chrome[S], Firefox[S], Epiphany[S] (all similar with implementation-related nuances)
- *Separated mail transfer agent architectures (and inspired approaches)*: qmail[K], Postfix[S], djbdns[K]
- *Microkernels*: MINIX[R], L3/L4 family[R], and many others.
- *Single-application classes*: IRSSI[S], Debian man[S], DHCPCD[S], VSFTPD[S], Firefox library sandboxing[R], Google V8[S] (libnode), Dovecot[S], Wireshark[S].

Appendix B. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

This paper investigates why the compartmentalization of software is still not a mainstream practice and how this status quo can be improved. The paper proposes a unified model for the systematic analysis, comparison, and directing of compartmentalization approaches. Additionally, the paper analyzes how compartmentalization is adopted in real-world applications and presents some recommendations for the research and adoption of compartmentalization.

B.2. Scientific Contributions

- Independent Confirmation of Important Results with Limited Prior Research
- Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1) A comprehensive taxonomy of existing efforts with a structured approach to categorizing compartmentalization methods across multiple dimensions
2) A large-scale review of both research efforts and mainstream software systems