μSCOPE: A Methodology for Analyzing Least-Privilege Compartmentalization in Large Software Artifacts

Nick Roessler
University of Pennsylvania
nroess@seas.upenn.edu

Lucas Atayde
Rice University
lsa4@rice.edu

Imani Palmer
Null Hat Security
inp2@protonmail.com

Derrick McKee
Purdue University
derrick.mckee@gmail.com

Jai Pandey
Nvidia
jpandey@illinois.edu

Vasileios P. Kemerlis
Brown University
vpk@cs.brown.edu

Mathias Payer
EPFL
mathias.payer@nebelwelt.net

Adam Bates
University of Illinois
batesa@illinois.edu

André DeHon
University of Pennsylvania
andre@acm.org

Jonathan M. Smith
University of Pennsylvania
jms@cis.upenn.edu

Nathan Dautenhahn
Rice University
ndd@rice.edu

ABSTRACT

By prioritizing simplicity and portability, *least-privilege engineering* has been an afterthought in OS design, resulting in monolithic kernels where any exploit leads to total compromise. μSCOPE ("microscope") addresses this problem by automatically identifying opportunities for least-privilege separation. μSCOPE replaces expert-driven, semi-automated analysis with a general methodology for exploring a continuum of security vs. performance design points by adopting a quantitative and systematic approach to privilege analysis. We apply the μSCOPE methodology to the Linux kernel by (1) instrumenting the entire kernel to gain comprehensive, fine-grained memory access and call activity; (2) mapping these accesses to semantic information; and (3) conducting *separability analysis* on the kernel using both quantitative privilege and overhead metrics. We discover opportunities for orders of magnitude privilege reduction while predicting relatively low overheads—*at 15% mediation overhead, overprivilege in Linux can be reduced up to 99.8%*—suggesting fine-grained privilege separation is feasible and laying the groundwork for accelerating real privilege separation.

ACM Reference Format:
Nick Roessler, Lucas Atayde, Imani Palmer, Derrick McKee, Jai Pandey, Vasileios P. Kemerlis, Mathias Payer, Adam Bates, André DeHon, Jonathan M. Smith, and Nathan Dautenhahn. 2021. μSCOPE: A Methodology for Analyzing Least-Privilege Compartmentalization in Large Software Artifacts. In *24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '21), October 6–8, 2021, San Sebastian, Spain.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3471621.3471839

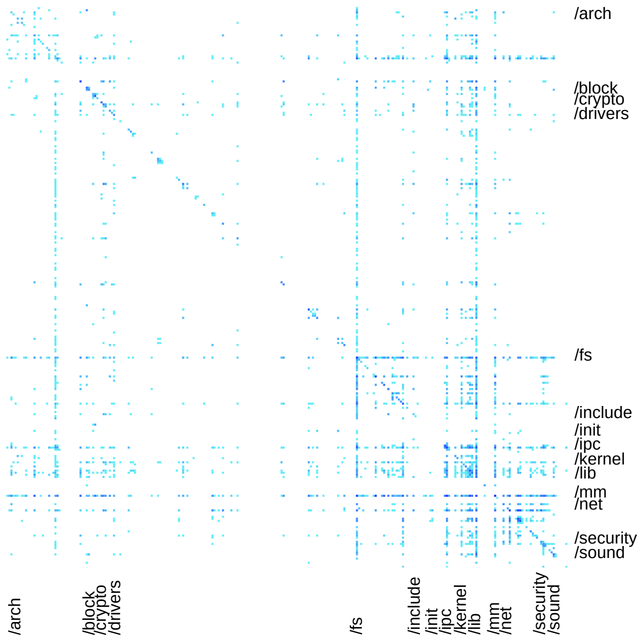


Figure 1: The interaction of code and objects in Linux kernel v4.10 at the directory level. Directories are in alphabetical order with labels shown on top-level directories; blank entries are nested in the preceding labeled directory. Color intensity indicates the logscale number of unique interaction edges *from* a directory (X axis) to code or data objects *owned by* another directory (Y axis). μSCOPE collects data at the instruction level; we aggregate to directories to produce a viewable figure.

1 INTRODUCTION

The Principle of Least Privilege is a key aspiration for secure system design [41, 62]. However, despite decades of work, we still use over-privileged software at every layer of the software stack. Fundamentally, composing systems while minimizing privilege is hard due to the complexity of defining privilege compartments and the performance challenges they impose [50], leading developers to simplify by building software with large, single trust domains. This is problematic because these "monolithic" software artifacts (*e.g.*, commodity operating systems) create an environment in which a single vulnerability could lead to full compromise of the system—for example, Project Zero's recent iOS exploit [10] was built from a single memory error in the kernel and led to a devastating zero-click, radio-transmitted and wormable complete device compromise. Facing a range of both external [65, 69] and insider threats [11, 36], the risks posed by monolithic software are not theoretical in nature, but a daily reality.

Addressing overprivilege in operating system design is a matter of both *mechanism* and *policy*. Privilege separation requires a self-protecting mechanism to enforce privilege boundaries and mediate any necessary boundary crossings (*i.e.,* a reference monitor [8]). Perhaps surprisingly, achieving this goal does not require abandoning today's operating systems (and all their source code) in favor of microkernel architectures (*e.g.,* L4 [25]); in fact, mechanisms for retrofitting privilege compartmentalization into monolithic kernels [18, 19, 52, 56, 72] and userspace applications [14, 22, 33, 34, 42, 46, 68, 71] have already been demonstrated. For example, Dautenhahn et al. [19] demonstrate that, by trapping on all updates to virtual memory, it is possible to embed an *intra-kernel* reference monitor (or "Nested Kernel") within an existing monolithic OS that can mediate accesses to physical memory or other system resources. They leverage the Nested Kernel to define a coarse-grained compartmentalization that assures the integrity of the core kernel in the presence of untrusted dynamically loaded modules.

These works demonstrate feasible *mechanisms* for retrofitting privilege separation, but their focus on coarse-grained compartmentalizations only scratches the surface of the Principle of Least Privilege. Why should a bug in one kernel subsystem have any bearing at all on the integrity of another completely independent subsystem? For that matter, why should a bug in one kernel function undermine the integrity of other unrelated lines of code?

These questions are a matter of *policy*. Privilege separation requires us to (retroactively) identify privilege compartments that provide a reasonable tradeoff between security and cost. With upwards of tens of millions of lines of code to consider, manually defining policies and privilege boundaries is infeasible. Unfortunately, while recent attempts at privilege reduction [6, 12, 23, 29, 35, 45, 47] have improved upon influential, but labor-intensive, early work [13, 37, 58, 72], they still fall short in terms of both least-privilege identification and automation. In these approaches, an expert either labels sensitive data (*e.g.,* private keys) or low-integrity components (*e.g.,* input parsing), and then performs a semi-automated compartmentalization routine that minimizes access to the sensitive data and/or the reach of the low-integrity code. However, even for state-of-the-art metric-based techniques [47], these approaches fall short of whole-system privilege reduction, instead protecting a few coarse-grained critical compartments. This is because they depend on the availability and omniscience of experts to label security-relevant data, code, or components—where, for massive systems like an operating system, there may be no such single expert. *At present, we have no systematic approach to identifying and evaluating privilege separation opportunities in monolithic software artifacts whose scale exceeds the knowledge of a single developer.*

With this in mind, we present μSCOPE ("Systematizing Compartmentalization Opportunities for Privilege Encapsulation"), a methodology that enables the identification of *whole-system* privilege reduction opportunities without requiring manual analysis by experts. μSCOPE instruments and profiles software activity at the granularity of instructions and objects, encoding each reference (*i.e.,* privilege requirement) in a novel low-level access control matrix, the CAPMAP (Context-Aware Privilege Memory Access Pattern). μSCOPE then uses the CAPMAP as the ground truth with which it compares competing software *compartmentalization hypotheses* that are either drawn from syntactic code structure (*e.g.,* functions, files, directories) or procedurally-identified through data-driven clustering algorithms that combine frequently interacting code and data. μSCOPE introduces a metric that allows it to evaluate the level of privilege separation that is possible for a given compartmentalization strategy compared to both monolithic (fully overprivileged) and the minimum-required-to-run (least privilege) baselines, then uses a performance model that estimates the cost of enforcement for a range of potential isolation mechanisms.

To demonstrate the power of μSCOPE and evaluate whether privilege separation is generically feasible, we apply μSCOPE to analyze the notoriously overprivileged Linux kernel. We identify the privilege separability of kernel objects, show the range of compartmentalizations that can be achieved in terms of aggregate levels of privilege separation and overhead, and automatically identify the data structures and design patterns that are important candidates for refactoring. These results demonstrate the utility of μSCOPE's automated privilege analysis. Figure 1 previews our results under a *directory-based* compartmentalization process. Here, individual instructions (references) are clustered by the directory in which the code resides. Even under this relatively coarse compartmentalization, the large amount of whitespace indicates massive privilege separation opportunities for Linux. Even more surprising, our performance analysis suggests that enforcing such privilege separation opportunities *might be* practical and eliminates costly manual separation efforts from exploring impractical compartmentalizations.

In summary, our primary contributions include:

- μSCOPE, a framework for comprehensive, automated *privilege analysis* (Sec. 5). It consists of four main components: (1) A novel low-level *privilege representation*, the CAPMAP; (2) A *compartmentalization model* that relaxes the standard object ownership model; (3) *Quantitative Metrics* for characterizing both privilege (the novel *privilege set*), and performance; (4) *Separability analysis*, a novel systematic exploration of entire compartmentalization spaces.
- An implementation of μSCOPE for the Linux kernel, binding the C language abstractions to the CAPMAP model (Sec. 6). μSCOPE's analysis code and data sets are available from https://fierce.cs.rice.edu/uscope/.
- A characterization of the degree to which Linux is privilege separable, including automated identification of potential refactorings (Sec. 8). We uncover opportunities for orders of magnitude in privilege separation, up to a 500x reduction (99.8%) in overprivilege, at predicted overhead of approx. 15%, suggesting that fine-grained privilege separation may be possible with low overhead in monolithic kernels. Further, we have released a browsable explorer[1] to allow researchers to better understand the interactions between Linux objects observed by μSCOPE.

2 MOTIVATION

As a concrete example to illustrate our concerns and motivate our approach, let us consider the credential structure (`struct cred`) from the Linux kernel (Fig. 2). This data structure controls the privileges that user space subjects (*e.g.,* processes, users) have to system resources (*e.g.,* tasks, files, sockets) [20]. As such, malicious

[1]https://fierce.cs.rice.edu/uscope/object_explorer

```
/* The security context of a task ... */
struct cred {
 kuid_t      uid;             // real UID of task
 kgid_t      gid;             // real GID of task
 kcap_t      cap_permitted;   // caps we're permitted
 struct key* proc_keyring;    // keyring for process
 struct user_struct *user;    // real user ID
 struct user_namespace *uns;  // user namespace
};
```

Figure 2: The (simplified) `struct cred` data structure used by the Linux kernel for task access control and privilege management.

manipulation of this structure is a common vector for privilege escalation. For example, a recent vulnerability can be exploited to change the UID field of a user space process' credential structure to that of root, thus gaining *root privilege* and access to all system resources [60].

In a monolithic kernel, the attack surface for `struct cred` alone is enormous: this field could be modified by any of the 104,240 potentially unsafe write instructions in the kernel; *i.e.,* any bug that allows an attacker to control one of the more than *a hundred thousand* possibly unsafe write instructions could become a vector to manipulate its contents. However, as our analysis identifies, only 113 write instructions from 31 functions should legitimately have access to `struct cred` objects. The other 104,127 write instructions (99.89%) hold excess privilege that reduce the security of the entire system. In other words, this privilege escalation attack is made possible because a compromised kernel component has privileges beyond those required to do its job.

2.1 Our Approach: Quantifying Privilege

This example illustrates a concrete way to quantify privilege and overprivilege: we can count the number of instructions that strictly require a given privilege, then compare that count to the number that the system actually allows. Further, if we were to divide the kernel into compartments such that only writes within a compartment with legitimate need could access `struct cred` objects, we could quantify the reduction in writes (privileges) that came from that particular compartmentalization. Reducing the number of instructions with privilege to write to `struct cred` makes it harder to find a vulnerability that can be exploited to corrupt `struct cred`, reducing its attack surface. If there is, on average, one exploitable security vulnerability in the code every 1,000 writes [9, 31, 51], a system with a hundred thousand privileged write instructions has a $1-(1-10^{-3})^{10^5}$=$1-3.5\times10^{-46}\approx100\%$ chance of having an exploitable vulnerability, while a compartment with only 100 such write instructions has only a $1-(1-10^{-3})^{10^2}=9.5\%$ chance.

Of course, even if we limit the code that can directly write into `struct cred` objects, an attacker could launch a confused deputy attack and invoke one of the 31 authorized functions in an attempt to manipulate *that* function into making the desired change. Since Linux is a monolithic kernel, any of the 51,258 other functions could conceivably be manipulated to call one of those 31 authorized functions. However, there are actually only 26 "secondary" functions with legitimate need to call those `struct cred`-authorized functions. Allowing the other 51,222 functions access is, again, overprivilege. A true least-privilege policy would remove that unnecessary privilege, further reducing `struct cred`'s attack surface.

However, `struct cred` isn't the only object in a program that carries security implications. For example, page table entries and secret keys are also clearly security-critical. We can perform similar analyses on each of these to quantify the minimum privileges necessary to run, the overprivilege of the monolithic design, and the privilege implications of a particular compartmentalization. We include a web-based object explorer generated from our tool to show the usage patterns of other kernel objects: μSCOPE object explorer.[1]

3 SECURITY MODEL

Threat Model. This work considers a realistic and powerful attacker that has discovered an exploitable software vulnerability (*e.g.,* memory corruption, disclosure, or code execution) in a monolithic software artifact, *e.g.,* an OS kernel.[2] It is possible for the attacker to trigger this exploit through the target system's interactions with low-integrity components such as user-space processes, network communications, or peripheral devices. Leveraging this exploit, the attacker seeks to take control of the system or gain access to confidential data – under normal circumstances, the above exploit alone would be sufficient to take full control of the system. We conservatively make no assumption about the specific system objects that the adversary seeks to access or corrupt; it is possible that *any* object is relevant to the attacker's objectives.

System Model. The target system is not without its own defenses. We assume it is equipped with a state-of-the-art reference monitor [8], the likes of which have been concretely instantiated in recent work [14, 18, 19, 22, 27, 33, 34, 42, 46, 52, 56, 68, 71, 72]. This mechanism can provide *complete mediation* over attempted accesses at arbitrarily fine granularities, down to the memory references contained in individual lines of code. It is *tamper proof,* meaning that the reference monitoring cannot be disabled during operation. Due to its small size, the reference monitor has been *verified* to operate correctly. Critically, this reference monitor is able to operate securely and correctly without the use of hardware-based protection – that is, it executes in the same protection ring [63] as the tasks that it mediates – allowing it to restrict the privileges of other Ring 0 code. However, while the mechanism for privilege separation is assumed to be present, the optimal *security policy* for minimizing the privilege of the attacker is unknown.

4 DESIGN GOALS

The aim of μSCOPE is to systematically analyze fine-grained, whole-system privileges within monolithic trust environments. Specifically, it aims to enable (1) comprehensive privilege analysis and policy derivation, (2) automated instead of manual analysis, and (3) exploration of the continuum of privilege-performance points rather than a handful of single points in the space.

4.1 Comprehensive Privilege Coverage

Prior work has focused on manual or semi-automated compartmentalization by experts [12, 24, 29, 37, 47, 58, 67, 72]. In general, these approaches selectively (1) sandbox buggy components (e.g., parsers) or (2) protect a limited subset of sensitive data (code-pointers or

[2] The recent Project Zero iOS zero click radio exploit is an example of such a vulnerability that allows circumvention of all mitigations in a monolithic kernel [10].

secret keys). However, considering the capabilities and objectives of our attacker, such an approach is not sufficient because it only restricts the privileges of one or two critical components. Our solution must be able to define a privilege policy that assures that the attacker's privileges will be always be restricted, even at an *arbitrary* and *unknown* entry point into the system.

4.2 Automated Analysis

Today's state-of-the-art in privilege reduction is based on manual, expert analysis to identify what excess privileges the system should remove. As code bases grow in age and complexity, the demand for experts outstrips their availability and capability. For the largest of code bases, many of which are decades old, no single person is an expert on the whole system and all of its interactions. For example, today's Linux kernel contains 28 million Lines-of-Code, contributed by over 19,000 developers [1], leaving it susceptible to a wide range of vulnerabilities [11]. Accepting that experts may not be available and may be fallible, our solution must take an automated approach to privilege analysis.

4.3 Privilege Continuum

Between a fully-separated, least-privilege design and a monolithic design, there is a vast set of possible decompositions at various points in the security vs. performance tradeoff space. With current manual and semi-automated compartmentalization techniques, it is prohibitively expensive to explore even a fraction of this space because each point requires (1) expert analysis and (2) significant engineering to evaluate the viability of the choice. Furthermore, a common concern is that privilege separation is not viable at fine granularities due to performance costs, which deters practitioners and researchers alike from even considering such options. Instead, our solution must systematically explore a wide range of points in the compartmentalization continuum. The tools we develop must be flexible and easily integrate expert domain-specific knowledge, to the extent available, through parameter adjustment or by placing constraints on the search space.

5 THE μSCOPE METHODOLOGY

In this section, we present the generic μSCOPE methodology. We show its concrete application to Linux in Sec. 6.

5.1 Privilege Model and CAPMAP

The μSCOPE privilege model is based on mapping software components into *subject* and *object* domains in order to track their access *privileges* at runtime. In object-oriented languages, innate definitions for subject and object emerge based on the language's structure. However, such definitions are not apparent in procedural languages such as C. Moreover; our objective is to evaluate a continuum of privilege separation tradeoffs, some of which may conflict with the object-oriented abstraction. Instead, we define a *privilege* as an ISA-level operation (memory read, memory write, function call, return, and memory deallocation) that may be performed by a *subject* (instruction) on an *object* (virtual address region of memory). We choose this low-level representation due to its generality; all access privileges can be reduced to instruction- and byte-level, regardless of the program language.

Def. 1 (Privilege). *A* privilege *allows an instruction, $i \in I$, to perform a low-level operation, $op \in Ops$, on object, $o \in O$. I is the set of all instructions, O the set of all objects, and Ops, the set of low-level operations.*

This instruction-level privilege separation represents the finest-grained separation that we identify in μSCOPE (Sec. 5.3.1). For this finest-grained definition, the machine instructions I form our subject domain. For allocations and frees, we use the instruction that performs the call to the allocator/free routine as the identifier for that subject. Objects are likewise labeled by the instruction that calls the allocator routine. However, each instruction is also an object since it can be called (and potentially written, in case of mutable code), allowing us to capture privileges needed to make individual calls and returns. Aside from dynamically loaded or generated code (considered in Sec. 11), identifying dynamically allocated objects with allocating instructions means the set of object classes are limited to the set of statically allocated objects and statically known allocation instructions. Therefore, the set of instructions and objects can be determined at compile time and do not change during execution.

For context sensitive privilege analysis, it is possible to extend the subject tuple to include *separation contexts*, such as the call chain or kernel entry point. For practical reasons (*e.g.,* state explosion in the dynamic tracing system) we leave such exploration to future work. Note, however, that the metrics presented here can easily accommodate context sensitivity. Our algorithmic approaches (Sec. 6.2) can also handle context-sensitive subjects as is, but further specialization may be needed to exploit context to its fullest extent.

Next, we define a privilege predicate $priv(i, o, op)$ that indicates if instruction i is allowed to perform op op on object o. Different definitions of the function $priv(i, o, op)$ represent candidate policies on the continuum of the privilege separation design space. $priv(i, o, op)$ is an embodiment of Lampson's access matrix [41]. This simple operation matches the minimal conditions that Lampson identifies for isolated execution, selected because of its generality expressing privileges and its ability to easily map to compiler IR or assembly level operations.

Def. 2 (Privilege Set). *The Privilege Set (PS) is the set of all privileges for which $priv(i, o, op)$ is true for a program.*

A given PS can be modeled as a graph that encodes the whole-system privileges of the associated program. The instructions $i \in I$, and objects $o \in O$, are vertices in the graph, while $priv(i, o, op)$ defines whether or not there is an edge of type $op \in Ops$ between the nodes i and o. Alternately, PS can be modeled as an access matrix where rows are instructions and objects are rows and columns while op will appear in $cell(i, o)$ if $priv(i, o, op)$ is true.

Given the notion of privilege sets, it would clearly be valuable to identify PS_{min}, the minimum privilege set needed in order for the program to run. Our system will derive PS_{min} dynamically through the notion of CAPMAPs:

Def. 3 (CAPMAP). *The Context-Aware Privilege Memory Access Pattern (CAPMAP) is the minimum PS necessary for a program to run during the course of an observed execution. That is, $capmap(i, o, op)$ is the least privilege definition of $priv(i, o, op)$; if any privilege (i, o, op) is removed from the CAPMAP, the program cannot perform its task.*

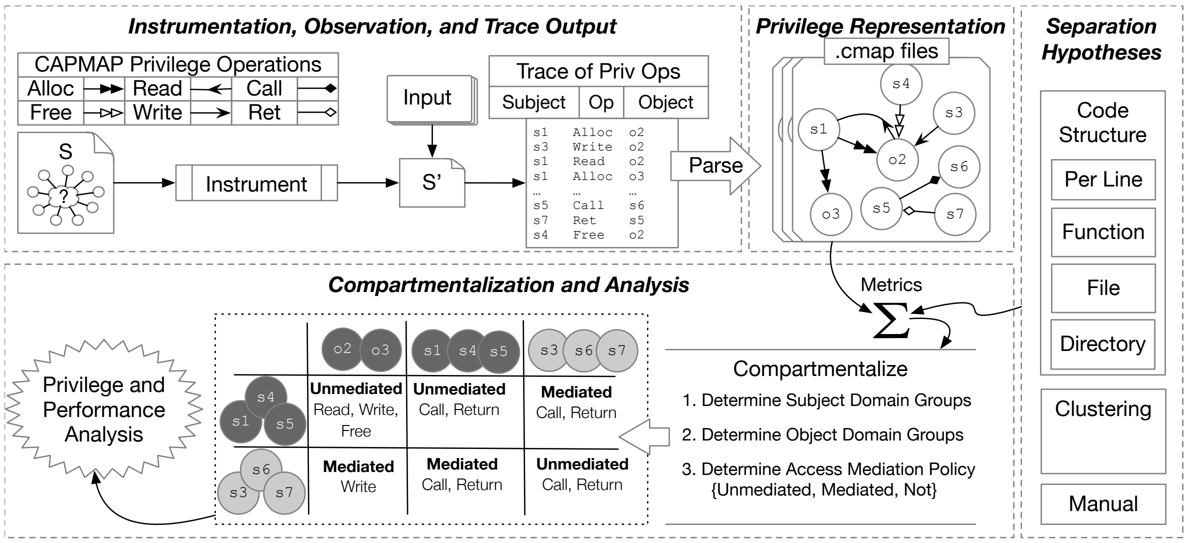


Figure 3: μSCOPE Overview. A software system S with unknown privilege separability is instrumented to trace its operations (read, write, call, return, and free) at the level of instructions and data objects. The trace is then transformed into a CAPMAP, a low-level representation of the privilege required by the software system. An analysis engine operates on the CAPMAP, allowing it to explore a range of compartmentalization hypotheses. We define new metrics to measure the privilege permitted by a given compartmentalization and use a simple analytical model to estimate the performance cost of enforcing the compartmentalization with a range of possible hardware mechanisms.

As a lower bound for $capmap(i, o, op)$, we include all privileges observed during one to many dynamic executions of the program (Sec. 7); we discuss the potential threats to validity posed by our dynamic analysis based approach in Sec. 11.

5.2 Compartmentalization Model

While PS_{min} privilege is ideal from a security perspective, instruction-level least privilege is a single (and, perhaps, impractical) point in the privilege-performance continuum. Instead, our compartmentalization model gathers individual instructions and primitive objects together into larger groupings. We call a grouping of instructions a *Subject Domain* ($sd \in SD$) and a grouping of objects an *Object Domain* ($od \in OD$), each of which is a collection of primitive instructions and objects, respectively.

We divide the entire code into a set of groups, $sd \in SD$. Each instruction, i, goes in exactly one sd. Similarly, we divide the data into groups with each object, o, in exactly one od. Recall that, since each instruction is also an object, each sd is also an od (or $SD \subset OD$).

Our basic compartmentalization model must specify for each operation op whether access from an sd to an od is: *Not* allowed, allowed but *Mediated*, or allowed *Unmediated*. The table in Fig. 3 shows one particular decision of an algorithm. Specifically, we define the mediation types as the following:

- *Not* access is appropriate when the subject group does not use an operation on an object group; we grant no privileges between sd and od for op.
- *Mediated* operations are dynamically validated against the CAPMAP at the fine-grained instruction and object level. This supports CAPMAP allowed, least-privilege access without allowing unnecessary access from other instructions in that subject group, thereby achieving high security but imposing per-access costs.
- *Unmediated* access between subject and object groupings mean that any instruction for the particular op from the sd to any object in the od will be permitted without fine-grained runtime monitoring. *Unmediated* edges represent a coarse-grained relaxation of privilege, but allow frequently interacting components to reduce costs. This matches a virtual-memory protection model where a subject domain maps in the object domain.

We can think of each sd and the set of ods to which it has unmediated access as a compartment. This allows each od to exist within multiple compartments. The mediation type may differ with the op type to allow different operational privileges; for example an od group that is only read by an sd may be mapped *Unmediated* for read but *Not* for write, call, return, and free. The SD and ODs form nodes in the coarser compartmentalization graph.

Def. 4 (Compartmentalization). *A* compartmentalization *is a division of instructions and objects into Subject Domain and Object Domain sets and an assignment of edge types,* $Type(sd, od, op)$*, to one of {Not, Mediated, Unmediated} for all* (sd, od, op) *triplets.*

We can reflect the privilege reduction of a given compartmentalization back to instruction-level privileges by consulting this coarse compartmentalization graph:

$$\begin{aligned} priv_{compart}(i, o, op) = capmap(i, o, op) \vee \quad (1) \\ \exists sd, od \Big((o \in od) \wedge (i \in sd) \wedge \\ (Type(sd, od, op) = \text{Unmediated}) \Big) \end{aligned}$$

In other words, the compartmentalized graph starts with all the minimum privileges observed in the CAPMAP. Then, additional unmediated edges are added between all instructions in sd and all objects in od. As a result, if any instruction $i \in sd$ and object $o \in od$ have an operation privilege defined in the CAPMAP, every instruction and object in the (sd, od) compartment is granted that operation privilege. Note that our compartmentalization model is more general than conventional models that typically (1) require objects to exist within at most one compartment (have unmediated edges from a single subject) and (2) assign object ownership based on the allocating subject.

5.3 Metrics

μSCOPE treats compartmentalization as an optimization problem over the privilege-performance space. To do so, it uses metrics that can be computed on a CAPMAP augmented with dynamic privilege counts to capture tradeoffs in privilege and separation costs.

5.3.1 Privilege. To quantify the privilege that exists in the system under various compartmentalizations, we use the size of the privilege set, $|PS|$ (see Sec. 2). To make the numbers generally meaningful for comparison, the Privilege Set Ratio (PSR) is defined as a ratio of the $|PS|$ under a particular compartmentalization and the $|PS|$ of the monolithic case, *i.e.,* when the whole task is a single compartment. We break down five different operations (read, write, call, return, and free) and provide a separate PSR for each.[3]

Simply put, we add one unit of privilege to the $|PS|$ for each particular instruction that is allowed to perform the specified operation on a particular object. For memory reads and writes case, the unit object is a byte of memory, and we group together all the bytes allocated by a particular static instruction as a single object class. For calls and returns, the unit is a single function entry or return point. The total privilege then is the weighted sum of all instructions and the objects they are allowed to operate upon. Specifically, for each operation type op, we can compute $|PS(op)|$ for any $priv(\cdot)$ definition as a weighted sum over the privileges that exist:

$$|PS(op)| = \sum_{i \in I} \sum_{o \in O} cpriv(i, o, op) \times w(o, op) \tag{2}$$

Here $cpriv$ simply has a 1 when $priv(i, o, op)$ is true, and 0 when it is false. $w(o, op)$ is a weighting function that potentially depends on the operation, the size of the object, and the security importance of the object. In the simplest case, it could be the size of the object in bytes.

The reference count for the monolithic case, $|PS_{mono}(op)|$, is simply the case where all feasible privileges exist. So, we evaluate Eq. 2 with $priv = priv_{mono}$:

$$priv_{mono}(i, o, op) = \begin{cases} true, & \text{if } i \text{ performs } op \\ false, & \text{otherwise} \end{cases} \tag{3}$$

Conversely, for the least-privilege compartmentalization $PS_{min}(op)$, every instruction is its own sd and every object is its own od. We can compute $|PS_{min}(op)|$ as Eq. 2 with $priv(i, o, op) = capmap(i, o, op)$. With this in mind, the lower bound of PSR is given as:

$$PSR_{min}(op) = |PS_{min}(op)| / |PS_{mono}(op)| \tag{4}$$

For the compartmentalization case where edges are typed as *Not*, *Mediated*, or *Unmediated*, we compute Eq. 2 using $priv(i, o, op) = priv_{compart}(i, o, op)$ from Eq. 1. A concrete example to illustrate these metrics is shown in App. A.

[3]Other types of operations, such as jumps or memory allocation, can be represented in the same way.

5.3.2 Performance Model. To reason about the overhead of a candidate compartmentalization, we build a model to estimate the impact of these external operations, assigning a fixed cost to each mediated, unmediated, and internal operation:

$$\begin{aligned} T_{sep} \quad = \quad & T_{unsep} + \sum_{op \in OPS} N_{med}(op) \times T_{med}(op) \\ & + \sum_{op \in OPS} N_{unmed}(op) \times T_{unmed}(op) \\ & + \sum_{op \in OPS} N_{int}(op) \times T_{int}(op) \end{aligned} \tag{5}$$

Here T_{sep} is the estimated execution time for the separated design while T_{unsep} is the original, unseparated execution time. $T_{med}(op)$ is the additional time for a mediated external operation op, and $N_{med}(op)$, $N_{unmed}(op)$, and $N_{int}(op)$ are the total number of mediated, unmediated, and internal operations of type op. T_{unmed} is the additional time for an unmediated external operation. $T_{int}(op)$ is the the additional time for an internal operation, a call or return inside the SD, when separated for modeling cases, like SFI [26] (Sec. C), where each of these operations adds some overhead. We can calculate the number of mediated external accesses for a particular compartmentalization as:

$$N_{med}(op) = \sum_{i \in I} \sum_{o \in KO} d(i, o, op) \times tops(i, o, op) \tag{6}$$

$$d(i, o, op) = \begin{cases} 1, & \text{if } \neg\, (\exists sd, od\, ((o \in or) \land (i \in sc) \\ & \qquad \land\, Type(sd, od, op) = \text{Unmediated})) \\ 0, & \text{otherwise} \end{cases}$$

$tops(i, o, op)$ is the number of times i performs op on o. $d(i, o, op)$ is a similar calculation to Eq. 1 that identifies all edges in the fine-grained privilege map that are associated with an unmediated edge in the coarse-grained compartmentalization graph. We calculate unmediated and internal operations similarly with different conditions on $d(i, o, op)$. This model does not explicitly account for temporal or blocking effects; as such, the numbers are best interpreted as averages. We treat `memcpy` as a single mediated operation.

5.4 Separability Analysis

Once we have a CAPMAP to represent necessary privileges (Sec. 5.1), a dynamic performance trace to represent relative frequency of use, a compartmentalization model that defines the space of legal compartments (Sec. 5.2), and metrics for privilege and performance (Sec. 5.3), it becomes possible to systematically analyze the space of compartmentalizations. We could generate all such compartmentalizations, evaluate their privilege and performance metrics, and report the full continuum of privilege-performance points obtainable for the system. Unfortunately, the full set of compartments is too large to practically enumerate for all but the most trivial systems.

The CAPMAP with dynamic frequency counts on edges gives us a graph to which we can apply standard single- and multi-objective graph clustering and partitioning algorithms to gain access to the interesting points in the continuum. This allows us, for example, to formulate compartmentalization as constrained graph clustering optimization problems by placing constraints on properties of the compartments (*e.g.*, subject size, object size, maximum number of edges on subject or object) and the privilege metric (Eq. 2) or performance (Eq. 5) and identifying objective functions to minimize, such as excess privilege ($|PS(op)| - |PS_{min}(op)|$), performance overhead ($(T_{sep} - T_{unsep})/T_{sep}$) or the ratio of privilege and performance ($|PS(op)|/T_{sep}$). Using a sequence of optimization queries, we can establish bounds on feasible performance and privilege points in the space. Furthermore, since the models themselves are parametric (*e.g.*, relative weighting of operations and objects), analyses can be tuned for different needs (*e.g.*, privacy vs. integrity) and mechanisms (Sec. 6.6), and adjusted for perceived importance (*e.g.*, object weighting, Sec. 8.8). We provide concrete examples of parameterization and heuristic clustering algorithms in Secs. 6.2 through 6.6.

6 MAPPING LINUX AND C TO μSCOPE

In this section we apply the generic μSCOPE methodology to the Linux kernel. We present a concrete instance of the approach that makes selections for: (1) language bindings to generate meaningful identifiers for subjects and objects, (2) specific algorithms for choosing subject groups, object groups, and access mediation, (3) specific privilege metric weights for our analysis, and (4) a specific set of mechanism costs to estimate the performance overhead of separation, given a range of possible enforcement mechanisms. These decisions represent initial design choices and offer many parameterizations.

6.1 Mapping C for Fine-Grained Identification

Each machine instruction in the *vmlinux* must be mapped to a SD, and each static and dynamically allocated C object must be mapped to an OD. Objects includes global and per-CPU variables, as well as objects from Linux's dynamic allocators (Sec. 7.1). For simplicity of analysis, we statically compile all required kernel modules.

6.2 Subject Domains

The data in the weighted CAPMAP provides us with rich, low-level information about the control-flow flow and data-accessing patterns of code, from which we can intelligently produce subject domains. Because clustering is known to be NP-hard [7], we use a lightweight, greedy clustering algorithm that assigns instructions into clusters. More heavyweight clustering would only increase the high separability we are able to identify. We begin the algorithm by placing each function into its own cluster; we then proceed to perform repeated cluster-merge operations until an assignment of code into Subject Domains is produced. To determine which clusters to merge at each step, we consider all possible pairs and compute the ratio of a utility function to that of a cost function for that pair; we then take the pair with the highest ratio, perform the merge, and repeat. The utility function we use is the expected performance savings of combining the two clusters: by combining frequently interacting pieces of code, we save on the costs of cross-compartment calls between those clusters. The cost function we use is the net increase in $|PS|$ incurred by the merge—that is, after merging two clusters, the code and data of each can be exposed to the code of the other (in the case of Unmediation), and $|PS|$ captures this quantification. The algorithm stops when there are no merges left with a ratio above a specified minimum threshold α (that is, no merges are favorable in terms of performance savings to $|PS|$). Intuitively, α specifies the acceptable tradeoff level of performance cost per unit of $|PS|$.

By varying values of α, we can produce a range of Subject Domains at various points in the privilege-performance continuum. We refer to subject domains constructed from this clustering algorithm from their values of α. We include a web-based compartment explorer for compartments generated with this algorithm: μSCOPE compartment explorer.[4]

6.3 Object Domains

After assigning instructions into Subject Domains, we then assign the objects from the CAPMAP into ODs. At the most fine-grained level, each object would be mapped into its own Object Domain (*e.g.,* the data allocated from each allocation site, or each global variable, would be its own OD). For some enforcement mechanisms, such as Virtual Memory using an MMU, there may be significant performance implications for subjects that are allowed access to many ODs (*e.g.,* TLB pressure). For these enforcement mechanisms, we run an object clustering algorithm that combines object classes together into coarser ODs, so that no SD has access edges to more than a specific object limit number of ODs. For some of the enforcement mechanisms we model (capability hardware, direct hardware support) no object clustering is applied.

To cluster objects, we use a greedy clustering algorithm similar to the one we use for creating subject domains. We begin by assigning each object class into its own OD. We then iteratively consider each SD that has access edges to more than the object limit number of ODs. For each such SD, we consider all pairs of ODs accessed by the SD as candidates for a merge. We select the pair that has the lowest value of a cost function, merge those ODs into a single OD, then move on to the next SD that is over the limit until all SDs satisfy the object limit constraint. The cost function we use to evaluate merges is the net total increase of $|PS|$ that would result from the merge—since merging object classes will open up more PS (due to each OD being possibly mapped unmediated in multiple SDs). We set the object limit to 64 to match the number of entries in the DTLB on modern CPUs [17].

6.4 Access Mediation

For each Subject Domain, Object Domain and operation type triple (sd, od, op) we must choose a mediation type (Sec. 5.2). If the operation is not included in the CAPMAP, then the mediation is typed as *Not* and the operation is not allowed. For operations that are allowed, the mediation is typed as either *Mediated* or *Unmediated*.

We begin our algorithm with all edges typed as *Mediated*. We then pick the edge that yields the largest performance savings per unit increase of $|PS|$ to unmediate. We set its type as *Unmediated*, record the properties of the compartmentalization, then repeat the same process until all edges are typed as *Unmediated*. Note that this tradeoff curve connects the two extremes (all-*Mediated* and all-*Unmediated*) but that moderate points are likely more attractive concrete compartmentalizations that balance minimizing privilege with performance cost. Privilege-performance tradeoff curves generated from mediation selection are presented in Sec. 8.

[4]https://fierce.cs.rice.edu/uscope/compartment_explorer

Table 1: Performance profile modeling parameters.

Architecture	$T_{unmed}(op)$			$T_{med}(op)$	
	r, w	call, ret		r, w,	call,
	free	int	ext	free	ret
Kernel Context	0	0	6000	6000	6000
Page Table + EPT	0	0	450	1500	650
SFI (baseline)	50	25	25	150	50
SFI (optimized)	5	5	5	150	50
Capability Hardware	0	0	600	50	600
Direct Hardware	0	10	10	10	10

6.5 Weighting Parameters

For the privilege optimization objective used during clustering and mediation, we take a simple linear sum across the individual PS metrics for ops, $|PS(op)|$. Another decision to make is how best to weight objects. At a uniform object weight of 1, PSR could be interpreted as the ratio of permitted interactions in an access control matrix compared to the monolithic case. However, larger objects (such as composite structures containing multiple fields) likely represent additional privilege. We weight objects by their size; a size component in the weight also means that a refactoring to split apart objects reduces privilege. This means that our PSRs can be interpreted as an exposure reduction per byte compared to the monolithic case. Weight tuning is discussed further in Sec. 8.8.

For global objects and code we take the size to simply be the static size in bytes. For heap objects we take the size to be the average live data size in bytes associated with the allocation site in our dynamic runs. We model stack memory as a single monolithic object with a size equal to the average number of live stack bytes. Important future work will be decomposing stack memory for more fine-grained separation. For calls and returns we use $w(o, \{call, ret\}) = 1$. An advantage of the above weighting scheme is that it can be applied automatically with no human intervention (Sec. 4.2). There is an opportunity to further tune the compartmentalization algorithms by scaling the various privilege operation components or by weighting them according to a policy; e.g., confidentiality or integrity.

6.6 Performance Profiles

For demonstration, we use a set of *performance profiles* that illustrate a range of potential costs for different protection mechanisms (Tab. 1). All entries are given in cycles; references and calibration are detailed in App. C. The numbers are best interpreted as average times for operations including typical caching effects; as such, the simple model does not account for the specific time of each operation instance in context. Consequently, we pick conservative values to use for these averages, and, most importantly, the profiles model costs that span orders of magnitude to illustrate how curves shift with a range of costs.

7 EXPERIMENTAL METHODS

7.1 CAPMAP Tracer

To collect CAPMAPs from the Linux kernel, we use Memorizer [61]. Memorizer is a tracing kernel that uses a combination of source code annotations and compile-time tooks to capture every call, return, allocation, free, and memory access. Captured traces are stored in memory and written out after a tracing run for post-processing and analysis. We disable KASLR so that addresses are consistent across runs and use a single core configuration, but otherwise use the default kernel 4.10.0 configuration from Ubuntu LTS 16.04. Read, write and call logging are turned off during boot, but memory allocations are still traced. Logging is enabled before running a workload or LTP [3] test on the kernel. This means the CAPMAPs produced do not include permissions needed only during boot.

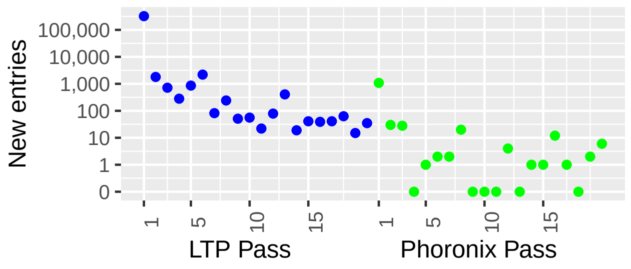


Figure 4: Linux kernel dynamic tracing privilege coverage. Twenty passes of the LTP test suite are added to a single CAPMAP (blue), followed by twenty passes of the Phoronix benchmarks (green). Each point shows the total number of new CAPMAP graph entries that are observed for the first time in that pass of testing. Note the log-scale Y axis.

7.2 Coverage Test Sets

To exercise the kernel and build an initial CAPMAP, we use the Linux Test Project (LTP) test suite [3] (release 20180926). The LTP contains suites of tests for stressing various kernel components (*e.g.*, scheduling, syscalls). We run all the tests applicable to our configuration (App. B). To improve coverage, we run the test suite twenty times. In Fig. 4, we show the number of CAPMAP entries (vertices and edges) that are found (instruction, object, or privilege used for the first time) by the LTP tests as they are added to a single CAPMAP (blue). On the last pass of the test suite, 35 new entries were added, for a cumulative total of 331,013 graph elements after training. To collect coverage CAPMAPs, we run the LTP tests on the tracing kernel using QEMU for a total of ~8 CPU-months.

7.3 Performance Benchmarks

While the LTP benchmarks are good for coverage testing, their emphasis on coverage means they do not represent a typical Linux workload that one would see in practice. To represent more typical performance, we run the Phoronix Test Suite [5] (v8.2.0) for performance overhead assessment. We combine the `kernel` and `linux-system` test suites and run all of the benchmarks that run on our configuration (22, see App. B). When we add twenty passes of the Phoronix benchmark CAPMAPs to the full coverage CAPMAP produced from the LTP runs, 1,196 (0.36%) new CAPMAP entries are discovered (green in Fig. 4).[5] Ten of the full benchmark passes encountered one or zero new instruction-level privileges; note that the privileges exercised in Phoronix but were not present in the LTP suite indicate ways to improve the quality of LTP.

For performance modeling, we boot the tracing kernel on a bare metal system with a 2.1 GHz Intel Xeon CPU E5-2620 and 128GB of memory.[6] We collect baseline kernel runtime T_{unsep} (Eq. 5) from

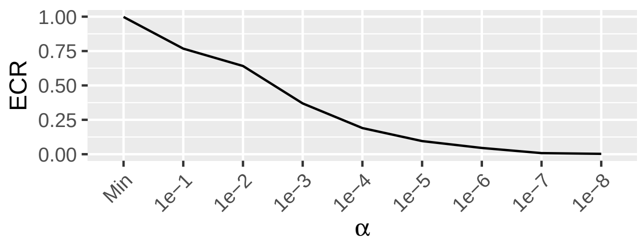
[5]These runs for coverage assessment were also collected using QEMU.
[6]We use the same `vmlinux` image in the coverage and performance experiments.

Sep. Hypothesis	**TopDir.**	**α=5e-8**	**α=1e-7**	**Dir.**	**α=5e-7**	**α=1e-6**	**α=1e-5**	**File**	**α=1e-4**	**α=1e-3**	**Func.**
PSR all-*Unmediated*	0.215	0.00520	0.00427	0.0302	0.00257	0.00133	0.000771	0.00289	0.000618	0.000578	0.000567
PSR half-*Unmediated*	0.0687	0.00282	0.00255	0.00745	0.00147	0.000697	0.0003102	0.000833	0.000204	0.000167	0.000140
PSR all-*Mediated*	0.0476	0.00047	0.00040	0.00272	0.00018	0.000085	0.0000552	0.000200	0.000047	0.000045	0.000040
struct key	6.10%	0.815%	0.785%	5.50%	0.770%	0.760%	0.738%	1.40%	0.730%	0.720%	0.720%
struct cred	38.4%	25.1%	23.0%	20.2%	16.6%	1.63%	1.10%	1.52%	0.695%	0.670%	0.664%
struct buffer_head	24.5%	24.2%	22.3%	20.1%	16.0%	3.95%	0.846%	2.05%	0.614%	0.604%	0.604%
struct file	71.6%	24.8%	22.9%	29.2%	17.0%	10.9%	4.66%	3.25%	2.65%	1.32%	1.16%

Table 2: Aggregate PSR and object write accessibility. For each separation hypothesis (row 1) we show the range of the aggregate PSR metric based on edge mediation (rows 2-4). Rows 5-8 show the percent of write instructions that have write privilege to the shown object in the half-*Unmediated* case. Some objects are very separable (`struct key`, `struct cred`) whereas other objects are poorly encapsulated and are difficult for the algorithms to separate (`struct file`).

	TopDir.	Dir.	File	Func.
ECR	0.21	0.35	0.66	1.00

(a) Syntactic Subject Domains ECR



(b) Clustered Subject Domains ECR

Figure 5: The External Call Ratio for the syntactic domains (top) and the algorithmic clustered domains (bottom).

the same system with the exact same kernel configuration, except with tracing disabled (vanilla Linux) using `perf` [4]. We also collect baseline function counts in the same manner on an independent run. Some functions are invoked proportional to runtime. As a result, our tracing kernel runs see more function invocations (by 27% on average) than the baseline function counts. For overhead estimates, we use the function counts from the baseline system and scale operation counts proportionally.

8 LINUX SEPARABILITY RESULTS

8.1 Linux Performance Separability

One important characteristic for performance is the External Call Ratio (ECR); that is, the fraction of dynamic calls that are external to the subject for a given choice of SDs and hence pay separation overhead costs. Fig. 5 (top) shows the ECR for domains generated from source code structure, and Fig. 5 (bottom) shows how the ECR trends with α for our algorithmically generated domains. At an α parameter of 10^{-4} the clusterer achieves a smaller External Call Ratio than the TopDir syntactic domain, which has compartments that are 400× larger on average. This shows the advantage of the clustering algorithms over the syntactic cuts: they have the freedom to place functions with high call connectivity in the same compartment to minimize the cost of domain crossings.

8.2 Linux Privilege Separability

Tab. 2 shows how much separation we can get under various separation hypotheses. For each separation hypothesis (row 1) we show the range of the aggregate privilege metric *PSR* from three edge assignments: all-*Mediated* (row 2), half-*Unmediated* (row 3) and all-*Unmediated* (row 4).

To show how the accessibility of several concrete objects trends with *PSR* and our various separation hypotheses, we pick a set of common Linux kernel objects (rows 5-8) and show the percent of write instructions from live code that have write privilege in the half-*Unmediated* case. Note that some objects are very separable (`struct cred`) while others are less so (`struct file`).

8.3 Privilege-Performance Continuum

Fig. 6 shows how we trade off total Privilege Set Ratio and performance overhead for the PageTable+EPT Performance Profile (Sec. 6.6, Tab. 1). Given a tolerance for a certain level of overhead, the privilege-performance graph allows us to see what level of privilege reduction we can potentially obtain. *This is a key advantage of systematic analysis and making the continuum available to developers.* The data shows there is a large potential for privilege reduction without manual refactoring or paying a substantial performance penalty. At a 15% overhead, we can achieve a privilege reduction of 500×. Note that we calculate overheads for kernel time, which is typically a small fraction of total time for most applications.

Each curve in Fig. 6 represents the range of privilege-performance points generated by edge mediation choices (Sec. 6.4), with the low-privilege/high-overhead end being fully mediated and the high-privilege/low-overhead end being all unmediated accesses. The fact the curves typically have a knee where the overhead drops quickly at the expense of a small change in privileges shows the value of allowing a small amount of unmediated access. Note that the domains produced from clustering (colored lines) provide substantially better privilege-performance tradeoffs than the code-structured domains (grayscale lines). Larger domains (produced from a smaller α value) have more privilege since no mediation is applied to calls and returns within a domain. Larger domains have lower costs since more calls and returns are internal to the domain and incur no overhead.

8.4 Highly-Connected Objects and Refactoring

There are some object outliers in the kernel that are accessed by many subjects; these objects pose the greatest challenges in object separability. The most highly accessed objects, measured in number of accessing functions, are `task_struct` (1,136), `ext4_inode` (610), `file` (529), and `dentry` (406).[1] These objects would induce high overhead if they could only be placed in a compartment with a single subject. The ability to mark edges as unmediated in our compartmentalization model, and, particularly, to allow unmediated

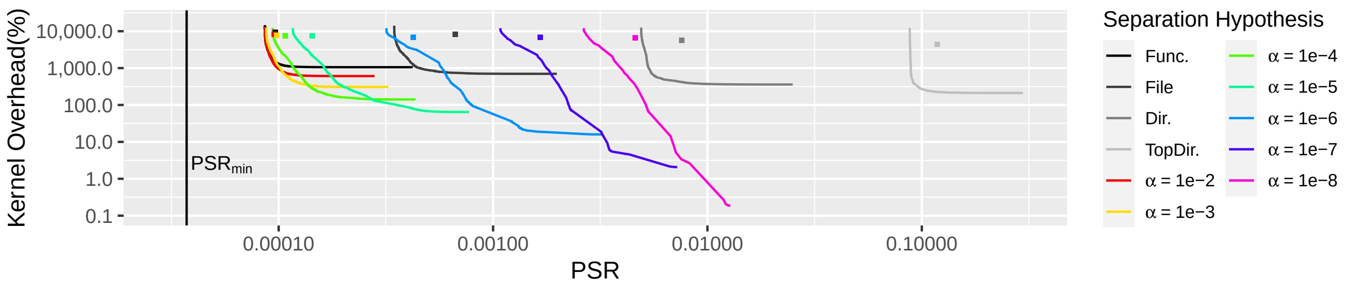


Figure 6: The privilege-performance continuum for each separation hypothesis using the EPT enforcement mechanism. The privilege lower bound (PSR$_{min}$) is shown as a black vertical line. The squares show the privilege-performance point when each object is owned (Unmediated) by the single subject with the highest access frequency.

access to an object from multiple subjects, can keep the overhead down for these subjects (Sec. 5.2). In Fig. 6, the squares show what would happen if we forced every object exclusively into the single compartment that accessed it most frequently. As can be seen, this inhibits all high-performance design points.

Importantly, this kind of analysis sets us up to consider refactorings that would improve separability. For example, we can run the compartmentalization algorithms on a moderate domain size ($\alpha = 10^{-6}$) and apply the mediation restriction that each object is owned (unmediated) by the single subject with the most accesses. The objects responsible for the largest fraction of mediated accesses from other subjects tells us directly which objects are poorly encapsulated and are preventing the algorithms from finding a tight separation. The worst offending objects of this type, measured by their fraction of the total dynamic accesses, are `task_struct` (responisble for 12.2% of all mediated accesses), `ext4_inode_info` (8.7%), `seq_file` (8.2%) and `seq_buf` (6.7%); this suggests that large improvements in seperability are possible through refactoring a small subset of the overall system, and that μSCOPE analysis can be used to guide these efforts.

8.5 Highly-Connected Subjects and Localizing

Similarly, there are some subject outliers that access many objects. The worst offenders were common C library operations (*e.g.*,`memcpy`, `strcmp`). To improve their separability, we add a new config option to the kernel to inline these functions into their calling compartments—this approach of localizing or replicating code is a simple way to remove the object overprivilege for stateless functions.

Of the remaining high object-degree functions, the worst offenders were related to strings—there are tens of thousands of read-only string constants in the kernel recording various messages and names. The function with the highest object degree was `filldir` which accepts a `char * name` argument and performs reads to 2,093 string constants. Excluding string constants, the highest object degree functions were `sysfs_add_file_mode_ns` (169) and `internal_create_group` (147), which access many global variables related to permissions. The functions with the most edges to heap objects were `__rcu_process_callbacks` (81), `__call_rcu` (80), and `__mutex_init` (40). With the help of a human designer to indicate where it is safe, these functions with high object privilege could be localized into compartments to produce a more separable design and μSCOPE can guide these priorities. We note that a majority of

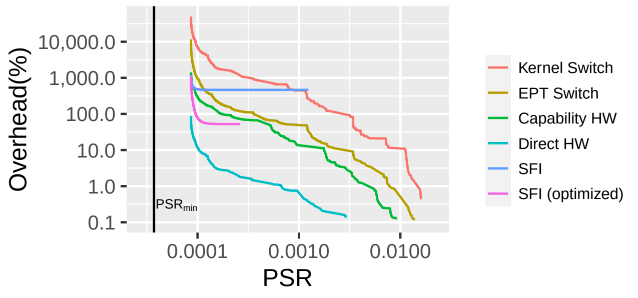


Figure 7: The Pareto-optimal privilege-performance tradeoff curve for each enforcement mechanism. The Pareto-optimal curve shows the lowest-overhead point for each PSR value found from any domain.

object clustering merges (Sec. 6.3) were combining together read-only string constants due to their large representation in high object degree functions. The algorithms intentionally avoid combining objects used by disparate pieces of code or unnecessarily opening up read and write permissions due to the large increase in PS that results from exposing objects to new code or operation types.

8.6 Allocator-Use Patterns

We further see that the allocating subject is often not the subject that uses the object the most. Object-style constructor/accessor patterns are common in the kernel. For example, `get_empty_filp()` is the sole allocator of `struct file` objects, but only performs around ~3% of dynamic accesses to such objects. We find that for heap objects, on average, the allocating function only performs around ~6% of accesses while the function with the most accesses performs around ~20%. This indicates that the allocator of an object is a poor predictor of actual dynamic use, and is therefore not a good method for defining compartments.

8.7 Performance of Various Mechanisms

Fig. 7 shows the privilege-performance Pareto tradeoff curves for the performance profiles introduced in Sec. 6.6 over our range of compartmentalizations. Capturing a range of performance overheads in our profiles allows us to illustrate how the tradeoffs shift, and possibly reshape, with different mechanism costs. The profiles also illustrate how lightweight mechanisms can enable higher privilege separation for lower costs. For example, at an overhead estimate of only ~1%, direct hardware support allows us to achieve the same level of separation that would impose a ~50% overhead for the EPT model. This highlights another reason automated compartmentalization that has access to the full compartmentalization

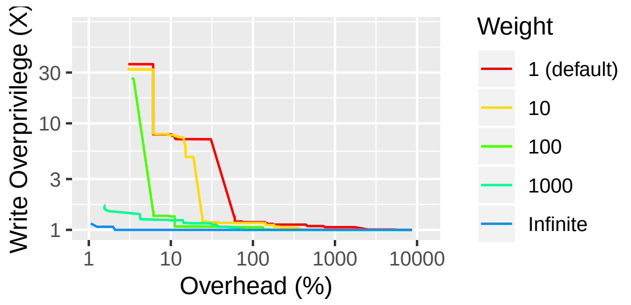


Figure 8: The impact of increasing `struct cred`'s write weight on its final write overprivilege. By increasing its weight, the write overprivilege can be driven lower for the same overhead level, giving a designer an easy tool for tuning the protection of a chosen object.

continuum is important—it allows a system to easily adapt to exploit new hardware support with lower costs for separation.

8.8 Security Tuning

Tab. 2 shows the write exposure of `struct cred` for various separation hypotheses and mediation levels; this data is from a default, fully-automatic compartmentalization flow. A developer can easily control the overprivilege on objects they deem sensitive (like `struct cred`) by increasing their weighting relative to other objects. This will drive the algorithms (Secs. 6.2-6.4) to reduce the overprivilege exposure for these items. In Fig. 8 we show the impact of increasing `struct cred`'s write weight on its final write exposure. This illustrates the advantages of automation in responding to evolving threat models and security preferences.

9 EXPLOIT CASE STUDY

The compartmentalization model introduced by μSCOPE can be qualitatively evaluated by studying concrete kernel exploits. This section analyzes three CVEs relative to various compartmentalizations to assess the concrete security implication of the privilege metric and separation methodology. We leave a more complete and systematic analysis across all kernel CVEs to future work.

CVE-2017-7308 is a vulnerability in the Linux 4.8 network stack that allows an unprivileged user to cause a kernel heap out-of-bounds write that can grant root access to an unprivileged user. The user facing packet-socket interface provides clients with the ability to request kernel networking data structures, like ring buffers, but lacks a critical security check. An adversary can submit a malformed request to the interface to build a ring buffer and overwrite a kernel timer function pointer. A common target is to use this to invoke code in `arch/x86/kernel/cpu/common` that disables two critical security protections (SMEP and SMAP [16]) by overwriting `CR4`. With these protections disabled, the user process can force the kernel into reading and executing memory in the user address space, which can then be used to grant a user full root access to the host.

Directory level compartmentalization (as well as the more fine-grained separations) would have prevented the exploit detailed above by removing the attack edge where the overwritten function pointer (in the kernel timer mechanism) is used to call the sensitive functions that disable SMEP/SMAP (in `arch/x86/kernel/cpu/common`).

CVE-2017-18344 [40] tracks a vulnerability in one of the POSIX timer system-call interfaces that enables unprivileged code to read arbitrary regions of kernel virtual and physical memory. The problem is that the `timer_create` system call fails to validate an input, specifically the `sigev_notify` field in a `k_itimer` structure, which is used to define a POSIX interval timer. The `sigev_notify` field is used to index into a global array of strings. The PoC uses the out-of-bounds read to access user space pages from within a kernel thread and eventually map arbitrary kernel pages into the user address space. The existing exploit fails when SMAP is enabled, *i.e.,* two large compartments, but event without that, this example hints at the broader need for compartmentalization and mediated access within the kernel. The function that executes the overflow only requires access to six objects, and can thus be restricted to avoid the corruption. Furthermore, this function is called so rarely that the clustering algorithms never grouped it with other code, and so in all of our compartmentalizations the out-of-bounds read is never permitted access to any other data.

CVE-2017-15649 is a use-after-free vulnerability that is caused by a race condition in the `net` kernel subsystem. After the race condition is triggered, a dangling reference to a freed heap object of type `struct packet_fanout` is held by a live structure. An attacker can manipulate the contents of the freed-but-accessible object by causing a fresh allocation of a similar size to claim and access the same memory. The `struct packet_fanout` contains a function pointer `id_match`, which, when overwritten, offers a control-flow hijack opportunity when the function pointer is later used. In a system that enforces CAPMAP compartmentalizations, only a small subset of the functions in the system have write permission to these objects, meaning that even the initial corruption will be more complex to execute and must be done through the `net` subsystem.

Assuming the function pointer can be overwritten successfully, there is a single instruction that performs the hijacked call. In Tab. 9 we show (1) the $|PS_{call}|$ of the specific indirect call instruction, (2) the total number of gadgets accessible to the hijacked domain, (3) the number of distinct registers that can serve as stack pivot targets, (4) whether or not Ropper can construct a write-what-where gadget, (5) whether or not Ropper succeeds in constructing a payload, and (6) the estimated overhead of that separation for the All-Unmediated case (see Fig. 6 for full tradeoff-curves). To determine whether Ropper succeeds in constructing a payload, we add an additional pass to Ropper in which it filters out gadgets that are made inaccessible to the hijacked domain by μSCOPE. This shows that the general compartmentalization algorithms based on PS not only eliminate needed privileges but also that exploiting this vulnerability without a typical ROP chain significantly increases the attacker's work factor as they must perform repeated confused-deputy attacks [30] to reach their target.

10 RELATED WORK

Early privilege separation approaches *reduced privilege* by manually decomposing a system [37, 58, 72]; such efforts require significant human capital, in the form of time and domain expertise, and are thus limited in terms of both scalability and the level of privilege reduction provided. Later approaches introduced various degrees of automation that reduce, but do not eliminate, the human capital requirement. This can be achieved through requesting

	Mono.	TopDir.	α=**1e-8**	α=**1e-7**	Dir	α=**1e-6**	α=**1e-5**	File	α=**1e-4**	Func.
$\lvert PS_{call}\rvert$	12,759,707	1,143,488	866,112	16,368	247,977	5,216	1,440	44,132	1,264	1,264
Total Gadgets	796,304	65,215	47,291	872	17,554	252	39	3,313	12	12
Stack Pivot Target Regs.	6	2	2	0	1	0	0	1	0	0
Write-What-Where	Y	Y	Y	N	N	N	N	N	N	N
Ropper Payload Succeeds	Y	Y	Y	N	N	N	N	N	N	N
Estimated Overhead	0%	200%	~1%	~1%	340%	15%	62%	670%	130%	1000%

Figure 9: CVE-2017-15649 metrics and statistics.

developer annotation of source code to derive privilege compartments [12, 13, 29, 47, 54], or the combination of partial code annotations with analysis infrastructures to further reduce developer burden (SMV [35], SOAAP [29], Wedge [12], ACES [15], ERIM [67], lwC [44], PM [47],[23]). These systems either retain the coarse-grained, default-allow model of privilege, or, in cases where they can support many compartments, still depend on experts: they are "semi-automatic" at best, providing an incomplete and *ad hoc* exploration of the privilege-performance space. Microkernels [11, 38, 59] and other manual separation efforts [2, 32, 53, 70] have been applied to OSs, but lack the automation and exploration advantages of μSCOPE's approach.

11 DISCUSSION AND FUTURE WORK

μSCOPE analysis shows that the Linux kernel runs with excessive privilege (over 25,000×) and has the potential for considerable privilege reduction (500×) while indicating minimal restructuring and excessive overhead (15%). This should be viewed as bringing an enticing opportunity to light, but it stops far short of showing how to engineer solutions that fully exploit it; it will still require significant contributions to fully extract this promise.

Coverage and Dynamic Analysis: μSCOPE uses dynamic analysis to collect privileges and their runtime usage counts for privilege and performance analysis. Our coverage results stabilized over our test suites and kernel workloads (Fig. 4), indicating that our analysis is quite comprehensive for the configuration under study. However, coverage is a limitation of dynamic analysis. Like other works [15], our framework could be combined with static analysis for a hybrid design. The PS_{min} difference between static and dynamic analysis would be interesting to explore. Note that, most mechanisms will incur some generalization when applying a CAPMAP, e.g., accesses could be generalized on a per-function or per-module level. It is unlikely that data will only be touched in uncovered passes and the implicit generalization will naturally include some potentially missed accesses. Omissions discovered from μSCOPE can be used to improve the quality of kernel test suites [3], and μSCOPE could be used in conjunction with related fields such as Whitebox Fuzz Testing [28] in discovering test cases for additional privilege coverage. Our needs for privilege coverage are well aligned with the needs for test coverage by the community at large.

Runtime modes, usability and alert messages: The reference monitor would support two modes: `audit mode` (in which violations are written to a log file) and `strict mode` (in which violations produce failstop behavior). The logs produced from `audit mode` include rich context, including the call stack and instruction-level access that produced the violation, which allows an engineer to discern whether or not to include the missing privilege and how to extend the testing suite to capture it. A system would typically be run in `audit mode` until the rate of violations drops below an acceptable threshold. Even in `strict mode`, note that not all violations would cause the OS to terminate: when acting on behalf of a program, only the offending system call or process need fail.

Interface Integrity: Our privilege metric identifies a "first order" separation in that it quantifies memory accessibility and the reachability of function calls. It does not assess indirect privilege that might come from, for example, an exported getter or setter. Refining privilege metrics to account for such effects, such as making the weight $w(o, op)$ of a call be a function of the privilege available to the callee, would be interesting future work.

Correlation of Security and Privilege Metrics: We hypothesize that privilege reduction is strongly correlated with security improvement (Sec. 4) and provide some evidence that it does (Sec. 2). Nonetheless, there is a need for a more complete and systematic characterization of the relationship between privilege separation and security to refine and validate efforts such as this one and PM [47].

Dynamically Loaded or Generated Code: In some cases a static instruction-level CAPMAP will not be adequate to define privileges: kernels load dynamic kernel modules, application software loads dynamically linked libraries, and code can be compiled dynamically (and compilation may be data-dependent). In these cases we can identify subjects and objects at a higher-level.

12 CONCLUSION

In this work we conduct a limit study on the privilege separability of complex software. Our analysis is made possible by μSCOPE, a framework that includes new models, metrics and algorithms for exploring the continuum of compartmentalization spaces. We apply μSCOPE to the Linux kernel and show that orders of magnitude of privilege separation are possible, how the separability of kernel objects can be explored and tuned, and that we can automatically identify important refactorings for further improving separability. We also highlight the potential for lightweight separation mechanisms to enable greater privilege separation for lower costs. These results demonstrate the utility of systematic privilege analysis.

ACKNOWLEDGMENTS

This research was funded in part by DARPA contracts HR0011-18-C-0011 and HR001119S0089-AMP-FP-034; NSF grants CNS-1513687, TWC-1513854, CNS-1801601, CNS-16-57534, CNS-17-50024 and CNS-2008867; ERC StG 850868; and ONR grant BAA N00014-17-S-B010. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the official policy or position of the U.S. Government.

REFERENCES

[1] [n.d.]. The Linux Kernel Open Source Project on Open Hub. https://www.openhub.net/p/linux.

[2] [n.d.]. SELinux Project. https://selinuxproject.org/.

[3] 2018. Linux Test Project. https://linux-test-project.github.io.

[4] 2018. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

[5] 2018. Phoronix Test Suite. https://www.phoronix-test-suite.com.

[6] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. 1986. Mach: A New Kernel Foundation for UNIX Development. In *Proc. USENIX*. 93–112.

[7] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. 2009. NP-hardness of Euclidean sum-of-squares clustering. *Machine Learning* 75, 2 (01 May 2009), 245–248. https://doi.org/10.1007/s10994-009-5103-0

[8] James P. Anderson. 1972. *Computer Security Technology Planning Study*. Technical Report ESD-TR-73-51. Air Force Electronic Systems Division.

[9] V.R. Basili and B.T. Perricone. 1984. Software Errors and Complexity: An Empirical Investigation. (1984), 42–52.

[10] Ian Beer. 2020. An iOS zero-click radio proximity exploit odyssey. https://googleprojectzero.blogspot.com/2020/12/an-ios-zero-click-radio-proximity.html?m=1.

[11] Simon Biggs, Damon Lee, and Gernot Heiser. 2018. The Jury Is In: Monolithic OS Design Is Flawed—Microkernel-based Designs Improve Security. In *Proceedings of the ACM Asia-Pacific Workshop on Systems (APSys)*.

[12] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*. USENIX Association, Berkeley, CA, USA, 309–322.

[13] David Brumley and Dawn Song. 2004. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13 (SSYM'04)*. USENIX Association, Berkeley, CA, USA, 5–5.

[14] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. 2016. Shreds: Fine-Grained Execution Units with Private Memory. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 56–71. https://doi.org/10.1109/SP.2016.12

[15] Abraham A. Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. 2018. ACES: Automatic Compartments for Embedded Systems. In *27th USENIX Security Symposium (USENIX Security 2018)*. USENIX Association, 65–82. https://www.usenix.org/conference/usenixsecurity18/presentation/clements

[16] Intel Corporation. [n.d.]. 4.10.1 Paging Modes and Control Bits. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf

[17] Intel Corporation. [n.d.]. *Intel 64 and IA-32 Architectures Optimization Reference Manual.*

[18] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. 2007. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. ACM, New York, NY, USA, 351–366. https://doi.org/10.1145/1294261.1294295

[19] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. 2015. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. *SIGARCH Comput. Archit. News* 43, 1 (March 2015), 191–206. https://doi.org/10.1145/2786763.2694386

[20] David Howells. [n.d.]. Credentials in Linux. https://www.kernel.org/doc/Documentation/security/credentials.txt.

[21] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. HardBound: Architectural Support for Spatial Safety of the C Programming Language. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 103–114. http://acg.cis.upenn.edu/papers/asplos08_hardbound.pdf

[22] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Jr., Benjamin C. Pierce, and André DeHon. 2015. Architectural support for software-defined metadata processing. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 487–502.

[23] Xinshu Dong, Hong Hu, Prateek Saxena, and Zhenkai Liang. 2013. A quantitative evaluation of privilege separation in web browser designs. In *European Symposium on Research in Computer Security*. Springer, 75–93.

[24] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazieres, Frans Kaashoek, and Robert Morris. 2005. Labels and event processes in the Asbestos operating system. In *ACM SIGOPS Operating Systems Review*, Vol. 39. ACM, 17–30.

[25] Kevin Elphinstone and Gernot Heiser. 2013. From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels?. In *Proceedings of the ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, New York, NY, USA, 133–150. https://doi.org/10.1145/2517349.2522720

[26] Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. 2006. XFI: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 75–88.

[27] Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. 2020. Enclosure: language-based restriction of untrusted libraries. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[28] Patrice Godefroid, Michael Y Levin, and David Molnar. 2008. Automated Whitebox Fuzz Testing. In *The Network and and Distributed System Security Symposium NDSS*.

[29] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. 2015. Clean Application Compartmentalization with SOAAP. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 1016–1031. https://doi.org/10.1145/2810103.2813611

[30] Norm Hardy. 1988. The Confused Deputy (or why capabilities might have been invented). *SIGOPS Operating Systems Review* 22, 4 (October 1988), 36–38.

[31] L. Hatton. 1997. Reexamining the fault density component size connection. *IEEE Software* 14, 2 (Mar 1997), 89–97. https://doi.org/10.1109/52.582978

[32] M.S. Hecht, M.E. Carson, C.S. Chandersekaran, R.S. Chapman, L.J. Dotterer, V.D. Gligor, W.D. Jiang, A. Johri, G.L. Luckenbaugh, and N. Vasudevan. 1987. UNIX without the Superuser. In *Proceedings of the Summer 1987 USENIX Conference*. USENIX Association.

[33] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 489–504.

[34] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. 2016. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Association for Computing Machinery, Vienna, Austria, 393–405. https://doi.org/10.1145/2976749.2978327

[35] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. 2016. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *ACM Conf on Computer and Communication Security*. https://doi.org/10.1145/2976749.2978327

[36] Paul A. Karger. 1987. Limiting the Damage Potential of Discretionary Trojan Horses. In *1987 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Los Alamitos, CA, USA, 32. https://doi.org/10.1109/SP.1987.10011

[37] Douglas Kilpatrick. 2003. Privman: A Library for Partitioning Applications. In *Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference, June 9-14, 2003, San Antonio, Texas, USA*. 273–284.

[38] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 207–220.

[39] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 437–452. https://doi.org/10.1145/3064176.3064217

[40] Andre Konovalov. [n.d.]. Linux kernel: CVE-2017-18344: arbitrary-read vulnerability in the timer subsystem. https://www.openwall.com/lists/oss-security/2018/08/09/6

[41] Butler W. Lampson. 1974. Protection. *SIGOPS Oper. Syst. Rev.* 8, 1 (Jan. 1974), 18–24. https://doi.org/10.1145/775265.775268

[42] Wenhao Li, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. 2015. Reducing World Switches in Virtualized Environment with Flexible Cross-World Calls. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 375–387. https://doi.org/10.1145/2749469.2750406

[43] W. Li, Y. Xia, H. Chen, B. Zang, and H. Guan. 2015. Reducing world switches in virtualized environment with flexible cross-world calls. In *International Symposium on Computer Architecture (ISCA)*. 375–387. https://doi.org/10.1145/2749469.2750406

[44] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation.*

[45] Shen Liu, Gang Tan, and Trent Jaeger. 2017. PtrSplit: Supporting General Pointers in Automatic Program Partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. ACM, New York, NY, USA, 2359–2371. https://doi.org/10.1145/3133956.3134066

[46] Shen Liu, Dongrui Zeng, Yongzhe Huang, Frank Capobianco, Stephen McCamant, Trent Jaeger, and Gang Tan. 2019. Program-Mandering: Quantitative Privilege Separation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. Association for Computing Machinery,

London, United Kingdom, 1023–1040. https://doi.org/10.1145/3319535.3354218

[47] Shen Liu, Dongrui Zeng, Yongzhe Huang, Frank Capobianco, Stephen McCamant, Trent Jaeger, and Gang Tan. 2019. Program-mandering: Quantitative Privilege Separation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, UK) *(CCS '19)*. ACM, New York, NY, USA.

[48] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 1607–1619. https://doi.org/10.1145/2810103.2813690

[49] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. 2011. Software fault isolation with API integrity and multi-principal modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 115–128.

[50] Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* Ph.D. Dissertation. Johns Hopkins University, Baltimore, MD, USA. AAI3245526.

[51] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz. 2004. An empirical study of software reuse vs. defect-density and stability. In *Proceedings. 26th International Conference on Software Engineering*. 282–291. https://doi.org/10.1109/ICSE.2004.1317450

[52] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. 2020. Lightweight Kernel Isolation with Virtualization and VM Functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Lausanne, Switzerland) *(VEE '20)*. Association for Computing Machinery, New York, NY, USA, 157âĂŞ171. https://doi.org/10.1145/3381052.3381328

[53] Elliott I. Organick. 1972. *The Multics System: An Examination of Its Structure.* MIT Press, Cambridge, MA, USA.

[54] Gabriel Parmer and Richard West. 2011. Mutable protection domains: Adapting system fault isolation for reliability and efficiency. *IEEE Transactions on Software Engineering* 38, 4 (2011), 875–888.

[55] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. 2017. kRˆX: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In *Proc. of EuroSys*. 420–436.

[56] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. 2020. xMP: Selective Memory Protection for Kernel and User Space. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 563–577. https://doi.org/10.1109/SP40000.2020.00041

[57] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis. 2020. xMP: Selective Memory Protection for Kernel and User Space. In *2020 IEEE Symposium on Security and Privacy (SP)*. 563–577. https://doi.org/10.1109/SP40000.2020.00041

[58] Niels Provos, Markus Friedl, and Peter Honeyman. 2003. Preventing Privilege Escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12 (SSYM'03)*. USENIX Association, Berkeley, CA, USA, 16–16.

[59] Richard F. Rashid and George G. Robertson. 1981. Accent: A Communication Oriented Network Operating System Kernel. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles* (Pacific Grove, California, USA) *(SOSP '81)*. ACM, New York, NY, USA, 64–75. https://doi.org/10.1145/800216.806593

[60] Rick. 2018. Never-Ending Security: eBPF and Analysis of the Get-Rekt-Linux-Hardened.c Exploit for CVE-2017-16995.

[61] Nick Roessler, Yi Chien, Lucas Atayde, Peiru Yang, Imani Palmer, Lily Gray, and Nathan Dautenhahn. 2021. Lossless instruction-to-object memory tracing in the Linux kernel. In *Proceedings of the 14th ACM International Conference on Systems and Storage*. 1–12.

[62] Jerome H. Saltzer and Michael D. Schroeder. 1975. The Protection of Information in Computer Systems. *Proc. IEEE* 63, 9 (1975), 1278–1308.

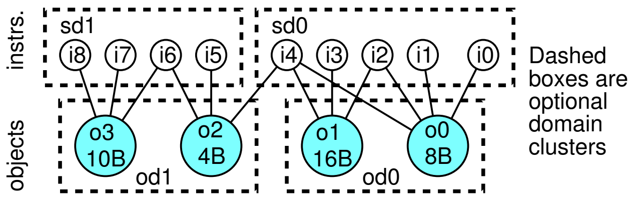
[63] Michael D. Schroeder and Jerome H. Saltzer. 1972. A Hardware Architecture for Implementing Protection Rings. *Commun. ACM* 15, 3 (March 1972), 157–170. https://doi.org/10.1145/361268.361275

[64] Bin Shi, Lei Cui, Bo Li, Xudong Liu, Zhiyu Hao, and Haiying Shen. 2018. ShadowMonitor: An Effective In-VM Monitoring Framework with Hardware-Enforced Isolation. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID) (LNCS, 11050)*. Springer Nature, 670–690. https://doi.org/10.1007/978-3-030-00470-5_31

[65] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 48–62.

[66] Tsuna. 2010. How long does it take to make a context switch? https://blog.tsunanet.org/2010/11/how-long-does-it-take-to-make-context.html. https://blog.tsunanet.org/2010/11/how-long-does-it-take-to-make-context.html

[67] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1221–1238. https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner



Compartmentalization	$\|PS\|$	PSR
Monolithic (writeable code)	9×(38+9×4)	1.00
Monolithic (non-writeable code)	9 × 38	0.51
sd0, sd1, od0, od1 (all edges unmediated)	246	0.37
sd0, sd1, od0, od1 (sd0 to od1 mediated)	180	0.27
sd0, sd1, each ok own domain (o1, o2 mediated)	140	0.21
Full Separation ($\|PS_{min}\|$) (each ik and ok is own domain)	122	0.18

$w(o, op)$=$o.size$; assume 4 Byte instructions.

Figure 10: Privilege metric illustration.

[68] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. {ERIM}: Secure, Efficient In-Process Isolation with Protection Keys ({MPK}). In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1221–1238.

[69] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018.* http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_08-3_Vasilakis_paper.pdf

[70] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. 2010. Capsicum: Practical Capabilities for UNIX.. In *USENIX Security Symposium*, Vol. 46. 2.

[71] R. N. M. Watson, R. M. Norton, J. Woodruff, S. W. Moore, P. G. Neumann, J. Anderson, D. Chisnall, B. Davis, B. Laurie, M. Roe, N. H. Dave, K. Gudka, A. Joannou, A. T. Markettos, E. Maste, S. J. Murdoch, C. Rothwell, S. D. Son, and M. Vadera. 2016. Fast Protection-Domain Crossing in the CHERI Capability-System Architecture. *IEEE Micro* 36, 5 (Sept 2016), 38–49. https://doi.org/10.1109/MM.2016.84

[72] Emmett Witchel, Junghwan Rhee, and Krste Asanović. 2005. Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*. ACM, New York, NY, USA, 31–44. https://doi.org/10.1145/1095810.1095814

[73] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control Flow Integrity & Randomization for Binary Executables. In *IEEE Symposium on Security and Privacy*. http://bitblaze.cs.berkeley.edu/papers/CCFIR-oakland-CR.pdf

A PSR METRIC EXAMPLE

Fig. 10 shows a simple graph and illustrates how PSR is calculated for these cases. The baseline monolithic case assumes writeable code. The rest of the cases assume non-writeable code, removing many privileges. Even the clustered case with all unmediated edges reduces privileges due to the *Not* edge from $sd1$ to $od0$. With the $sd0$ to $od1$ edge mediated (fourth row), the link only adds the base CAPMAP 4 units of privilege for $i4$ to access object $o2$. Changing the $sd0$ to $od1$ edge from mediated to unmediated (third row) adds 5×14=70 units since all instructions in $sd0$ now have privilege over all objects in $od1$, for a net increase of 70-4=66 privileges.

B TESTS AND BENCHMARKS

The Phoronix benchmarks we use are shown in Fig. 11, along with the amount of memory we associate with each kind of memory type based on our object weighting model. Code corresponds to the size of the `.text` section in the compiled `vmlinux` binary. Globals include the combined final size of the global objects stored in the `.data`, `.rodata` and `.bss` sections. The SLUB, Page and VMalloc allocators show the average data size of live objects as discussed in our object weighting model. μSCOPE treats stack memory as a single object, with a size equal to the average number of live stack bytes across all kernel stacks. Memblock memory is physically allocated from the Memblock subsystem; it is treated as a single object with a size equal to the number of used pages. Lastly, VMEMMAP corresponds to the size of the sparse virtual memory map structure used by Linux for fast translations. `gnupg` uses an unusually large amount of SLUB heap memory, which we identified as the Linux buffer cache.

We use the following LTP tests: `admin tools`, `can`, `cap bounds`, `commands`, `connectors`, `containers`, `crypto`, `dio`, `fcntl locktests`, `filecaps`, `fs`, `fsbind`, `fs ext4`, `fs perms simple`, `fsx`, `hugetlb`, `input`, `io`, `kernel_misc`, `ltp aio stress`, `ltplite`, `math`, `mm`, `network commands`, `nptl`, `pipes`, `power management tests`, `sched`, `power management tests exclusive`, `quickhit`, `securebits`, `stress`, `syscalls`, `syscalls ipc`, `timers`, and `tpm tools`.

C PERFORMANCE PROFILES

Page Table Process Protection: One way to provide separation is via the use of virtual memory. Subject Domains are mapped to processes. Object Domains are mapped to contiguous, page-aligned regions of virtual memory, which may contain either static objects or serve as pools for dynamic allocations for that OD. Unmediated pages are mapped in to the process page table according to their allowed permissions. Mediated accesses generate a trap to a supervisor, which performs the CAPMAP check, and, if allowed, performs the operation or performs a context switch in the case of a cross-domain call or return. This is modeled by the 6000 cycles for mediated operation based on a rough estimates from [43] and [66]. A highly optimized microkernel context switch might be less expensive [25] and closer to some of the leaner options that follow.

EPT Protection: The `vmfunc` [48] operation in modern Extended Page Tables (EPT) makes it less expensive to change page tables for an operation. Unmediated reads, writes, and frees, as well as internal calls, are directly mapped in the page table so they can complete with no overhead. External calls and returns make an explicit `vmfunc` call to perform the context switch; external mediated calls and returns perform a CAPMAP check in the `vmfunc` call [43, 48]. Mediated reads and writes will trap, check the CAPMAP, and perform the `vmfunc` operation from the trap, when allowed [64]. We model two traps (at 200 cycles each), two `vmfunc` calls (at 450 cycles each) and one CAPMAP lookup (at 200 cycles) for the total of 1500 cycles for a read, write or free. For the call, we model one `vmfunc` call and one CAPMAP lookup. We calibrated the `vmfunc` timing by measuring the kernel overhead time for Page Table protection implementation in the public release of xMP [57] and counting the number of added `vmfunc`s; so the 450 cycle represents the average cycles added per `vmfunc` including caching effects. These numbers are consistent with [64]. The raw number of cycles in the `vmfunc` is closer to 150, consistent with [39], but that doesn't include the caching impact. We measure the 200 cycles per CAPMAP lookup from our hash implementation.

Software Fault Isolation (SFI): In an SFI scheme [26, 49, 55], we can check any potentially unmediated read, write, or free access with an inline code monitor. We model two cases, one standard (baseline), and one optimized based on information we have in the CAPMAP (optimized). When we know from the CAPMAP that a small number of unmediated objects is accessed from a particular instruction, these can be checked quickly with specialized, inline base and bounds checks [55]. Our tracing shows that the dynamic distribution of accessed objects is highly skewed (call entropy is 0.041, read/write entropy is 0.265); this means the checks can be constructed in a tree organized like a Huffman encoding. Our optimized model includes the cost for a single check in the average case. Mediated accesses can be checked with a hash table lookup. External calls and returns can be wrapped with a springboard to permit only CAPMAP allowed operations and change context information when a call or return changes domains [73].

Capability Hardware: Capability hardware can restrict operations without requiring virtual memory and hence page table changes [71]. Mediated read and write operations can use capability pointers. Mediated calls and returns still require some time to check the CAPMAP and change capabilities, but this can be less expensive than a traditional OS context switch. We take the 600 cycle estimate from Tab. 2 in [71]. In the best case, mediated reads and writes can use the capability pointers, but may require some addition cycles to select and load capability pointers. Unmediated operations on a single OD can use the capability bounds check to eliminate per reference costs.

Direct Hardware Support: If we were to design hardware directly to support the CAPMAP, it could look like the HardBound hardware hash mechanism [21] or a cached tag rule checking mechanism [22]. That is, on every operation, the hardware uses the program counter and the address of the object to consult a hardware cached hash of the CAPMAP. Hits to the cache will add no overhead, while misses will incur a few cycles to fetch replacement entries for the CAPMAP cache. We use 10 cycles as a crude estimate for average time of a reference considering most references will take 0 time, but 5% of references may take 200 cycles. Since the miss rate will decrease with cluster size, assuming this high, fixed miss rate will make overhead results increasingly pessimistic with increasing cluster size.

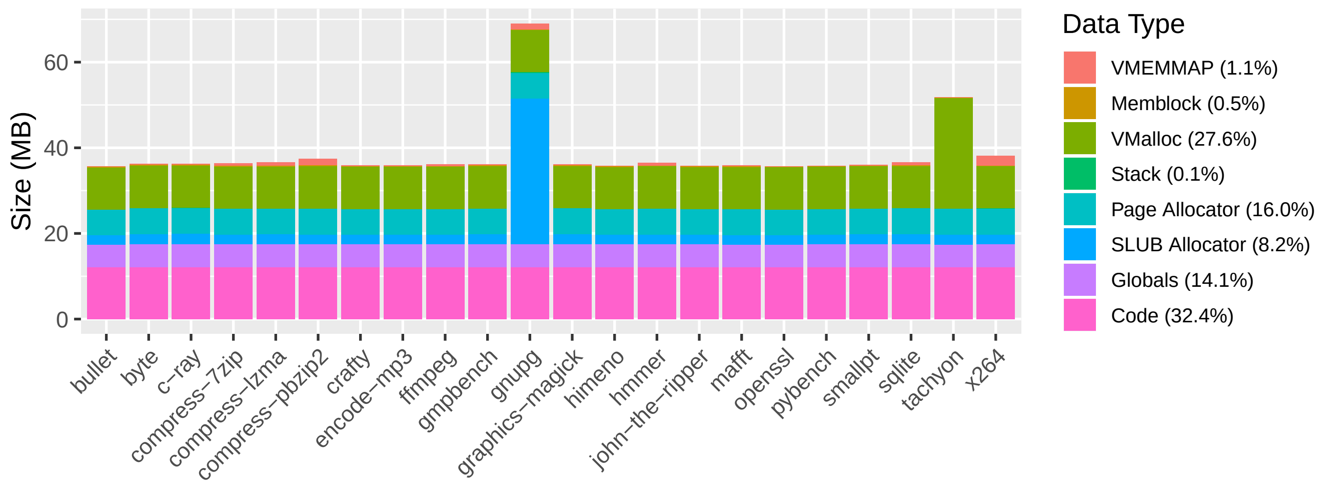


Figure 11: The amount of data we associate with each kind of memory region based on our object weighting model.