

CPM Compartmentalization File Format Proposal

Version 1.4
April 6, 2025

Author: Nick Roessler (nroessler@draper.com)
Contributors: André Dehon, Linus Wong, Jialiang Zhang,
Jing Li, Greg Sullivan, Eli Boling, Silviu Chiricescu

Contents

1	Introduction	2
2	Overview and Definitions	2
3	Motivating Example	3
3.1	Example With No Context	4
3.2	Example With Execution Context	5
3.3	Example With Object Context	6
4	File Format	7
4.1	Top-Level Structure	8
4.2	Object Map	8
4.3	Subject Map	8
4.4	Privileges	9
5	Subject and Object Identification	10
5.1	Subject Identifiers	10
5.2	Object Identifiers	11
5.2.1	SubObject Identifiers	11
6	Context	13
6.1	Universal Context Specifiers	13
6.1.1	Call Context	13
6.2	Linux Kernel Context Specifiers	13
6.2.1	uid	13
6.2.2	gid	14
7	Grammar for Privileges	14
7.1	A Note on omitted fields, empty fields, allow all, and allow none	15
8	Specification Consistency Checking	16
9	Extension: Runtime Counts	16
9.1	Format Extension	16
9.2	Example	17
10	Extension: Object and Subject Sizes	17
10.1	Format Extension	18
10.2	Size Definitions	18
10.3	Example	18
11	Planned Extensions	18
12	Tool Coordination Options – options.yaml	19

1 Introduction

This document defines the file format for a compartmentalization specification that accompanies an ELF program. The format precisely describes (1) how elements of that program are decomposed into compartments, and (2) which operations are allowed or disallowed between those compartments. It is intended to serve both as a target format for policy generation, enabling the flexible expression of policies, as well as the input format for policy enforcement mechanisms. This format is intended to provide a standardized format for communication between teams and tools on the DARPA Compartmentalization and Privilege Management (CPM) program. For flexibility, it is expressed in terms of ELF symbols and source code lines instead of raw addresses. As such, it assumes that debug information of the protected program is available.

A system enforcing the compartmentalization should need only the ELF program with symbols and the CPM compartmentalization specification; it may realize the enforcement “in-place” on the input ELF program or may produce a new ELF binary that realizes the compartmentalization.

The format is a subset of YAML, a human-readable markup language. This document defines an early version of the format that is subject to change.

2 Overview and Definitions

The CPM compartmentalization file format encodes fine-grained privilege separation defenses that can be applied to a monolithic program. For example, a collection of functions that handle untrusted network packets can be grouped together; they can be permitted to access the network buffer where packets arrive from, but restricted from accessing other data or calling other code. The file format is concerned only with *policy* (how the system is decomposed into compartments) and makes no claims about *mechanism* (how the enforcement of said policy is implemented).

For pragmatism on real programs, the compartmentalization model describes permissions in terms of easily identifiable system elements (such as functions and objects that appear in the program’s symbol table) and low-level operations such as read, write, and call accessibility. At a high level, it can be thought of as defining an access control matrix over all the elements of the program to explicitly state what operations are allowed between the various code and data elements.

It is designed to require minimal (or no) code refactoring and to be able to be applied “in-place” to an existing system by restricting privileges in that system. We use the following definitions:

1. **Subject Domain:** A collection of functions that are grouped together and treated as one. Any calls or control-flow operations between the contained functions are implicitly permitted, but calls to other Subject Domains must be explicitly granted.
2. **Object Domain:** A collection of primitive data entities (*e.g.*, global variables) that are grouped together and treated as one. Access to an Object Domain is granted or revoked in its entirety.

3. **Operation:** One of four operation types that can be granted or revoked: `read`, `write`, `call`, and `return`. The `read` and `write` permissions govern memory accessibility and the `call` and `return` operations govern function call and return capability.
4. **Execution Context:** Any aspect of the program's runtime state that is to be used to differentiate that state from other states in determining allowed permissions. Examples of execution context include the current call stack, the entry point of the kernel, or whether the current user is root.
5. **Principal:** The combination of a Subject Domain and an Execution Context. A Principal is the granularity at which operation privileges are granted or revoked.
6. **Object Context:** The Execution Context under which a dynamically allocated object was allocated. Principals may condition their `read` and `write` permissions based on the Object Context of the accessed object. Static objects such as global variables have an empty Object Context.
7. **Object:** The combination of an Object Domain and an Object Context.
8. **Compartment:** A Principal and all of the Objects it can access.

The `read` and `write` privileges are defined between a Principal and an Object. The `call` and `return` privileges are defined between a Principal and a Subject Domain. When no context is used, the model defines simple relationships between the program's code and objects.

3 Motivating Example

Consider the following code in which a user and admin password are defined and a user-supplied string is checked against both passwords:

```
1 #include <stdio.h>
2 #include <stdbool.h>
3 #include <string.h>
4
5 char user_password[] = "user123";
6 char admin_password[] = "admin100";
7
8 bool user_check_password(char * password)
9 {
10     return strcmp(password, user_password) == 0;
11 }
12
13 bool admin_check_password(char * password)
14 {
15     return strcmp(password, admin_password) == 0;
16 }
17
18 int main(int argc, char * argv[])
19 {
```

```

20 char * password = argv[1];
21 if (user_check_password(password))
22 {
23     // logged in as user
24 } else if (admin_check_password(password))
25 {
26     // logged in as admin
27 }
28 }

```

The following privileges are exercised by the program:

1. `main` can call both `user_check_password` and `admin_check_password`
2. Both `user_check_password` and `admin_check_password` can return to `main`
3. Both `user_check_password` and `admin_check_password` can call `strcmp`
4. `strcmp` can read the variables `user_password` and `admin_password`
5. `strcmp` can return to both `user_check_password` and `admin_check_password`

We describe the file format formally in Section 4, but include examples of the format for illustrative purposes here. We can express a compartmentalization for this program by first grouping together the functions and objects in the program into domains. At the simplest, we can place each function and object into its own domain like so:

```

# Create an Object Domain for each variable
object_map:
- name: UserPassword
  objects: [main.c|user_password]
- name: AdminPassword
  objects: [main.c|admin_password]

# Create Subject Domain for each function
subject_map:
- name: CheckUserPasword
  subjects: [main.c|user_check_password]
- name: CheckAdminPassword
  subjects: [main.c|admin_check_password]
- name: StringCompare
  subjects: [string.h|strcmp]
- name: Main
  subjects: [main.c|main]

```

After defining our domains, we can define the privileges we want to allow between those domains.

3.1 Example With No Context

In the simplest case where no execution context is used, we define 4 Principals and their allowed privileges, one for each function in the program. Without the inclusion of context,

our compartmentalization effectively defines simple relationships between the program's code and objects.

```
# Limit privileges to the intended set
privileges:
- principal:
    subject: CheckUserPassword
    can_call: [strcmp]
    can_return: [main]
    can_read: []
    can_write: []
- principal:
    subject: CheckAdminPassword
    can_call: [strcmp]
    can_return: [main]
    can_read: []
    can_write: []
- principal:
    subject: Main
    can_call: [CheckUserPassword, CheckAdminPassword]
    can_return: []
    can_read: []
    can_write: []
- principal:
    subject: StringCompare
    can_call: []
    can_return: [CheckUserPassword, CheckAdminPassword]
    can_read:
    - objects: [UserPassword, AdminPassword]
    can_write: []
```

This set of privileges makes explicit the set of call, return, read and write permissions listed at the beginning of this section and would configure the compartmentalization defense to enforce them.

Note that functions like `strcmp` can become overprivileged when they are used in different ways by the program. We can use context to limit these privileges further.

The above compartmentalization specification omitted the `execution_context` and `object_context` fields. We could have instead included `execution_context: all` for both principals and `object_context: all` for the non-empty `can_read` specification.

3.2 Example With Execution Context

If we want to strengthen the separation by limiting the privileges available to `strcmp` based on the current call stack, we can do that by redefining our principals to include additional context. We take the example from the last section, but split the `StringCompare` principal into two principals, one for each call chain under which `strcmp` can be called.

```
# Use context to limit StringCompare's privileges
```

```

privileges:
- principal:
  subject: StringCompare
  execution_context:
    call_context: [main, CheckUserPassword]
  can_call: []
  can_return: [CheckUserPassword]
  can_read:
- objects: [UserPassword]
  can_write: []
- principal:
  subject: StringCompare
  execution_context:
    call_context: [main, CheckAdminPassword]
  can_call: []
  can_return: [CheckAdminPassword]
  can_read:
- objects: [AdminPassword]
  can_write: []

```

This separation is more restrictive than the first: now `strcmp` can only access the password data and return to a single caller depending on the context in which it was called. However, it may also cost more overhead to enforce: the enforcement mechanism must now track the call stack and condition permissions based on its status. The compartmentalization format is able to express many possible privilege decompositions for the same program.

3.3 Example With Object Context

Another feature included in the format is conditioning privileges based on the Object Context of a dynamic object. This allows objects allocated from the same code point (allocator call) to be treated differently based on how they were allocated and what Principal is trying to access them.

We present a new example program to illustrate these concepts. It represents a system with multiple users, each of which has a unique user id, the `uid`. We assume that when any of the functions are called, the `uid` of the invoking user is known by the runtime environment and can be inspected by the enforcement system.

The function `create_key` creates a new key. The function `encrypt_message` takes a key and message as an input, and encrypts the message in-place.

For brevity, we omit other parts of the system such as how these functions would be called and the message objects, focusing only on the keys.

```

1 byte * create_key()
2 {
3     byte * key = malloc(KEY_LEN);
4     init_key(key, KEY_LEN);
5     return key;
6 }
7
8 void encrypt_message(byte * key, char * message)
9 {
10     // encrypt message with key
11 }

```

We show the Principal for EncryptMessage, the Subject Domain containing `encrypt_message`:

```

# Illustrate uses of object_context
privileges:
- principal:
    subject: EncryptMessage
    execution_context:
        uid: U
    can_call: []
    can_return: []
    can_read: []
    can_write:
- objects: [Key]
    object_context:
        uid: U

```

Here, the `execution_context` of the Principal expresses a symbolic value `U`, representing whatever `uid` is active in the context when the `EncryptMessage` Subject Domain is executing. The write access permission to the `Key` object is conditioned on the Object Context having the same `uid U` when it was allocated (*e.g.*, they are both 317). This technique can be used to limit each user's access to just their own data, preventing any bugs where keys could leak from one user to another.

In the following sections, we elaborate on these features and sections in more detail.

4 File Format

A CPM compartmentalization file has three major sections: Object Domains, Subject Domains, and Privileges. The Object Domains section defines how the program's objects are grouped into Object Domains. The Subject Domains section defines how portions of the program's code are grouped into Subject Domains. Lastly, the Privileges section defines Principals and which operations they are granted.

The Subject and Object Domain sections refer to subjects and objects in a program by their unique string identifiers (object IDs and subject IDs). This mapping and naming scheme is described in Section 5.

The Privileges section uses context identifier strings, which represent an aspect of the system's context. This mapping and naming scheme is described in Section 6.

Section 7 consolidates the description from this section into a more concise grammar.

4.1 Top-Level Structure

A CPM compartmentalization file must have a top-level element of the `dictionary` type that contains at least these three key-value pairs:

- `object_map` : a list of Object Domain definitions
- `subject_map` : a list of Subject Domain objects
- `privileges` : a list of Privilege definitions

4.2 Object Map

The top-level key `object_map` is a list of Object Domain Descriptors, each of which is of type `dictionary`.

Each such dictionary defines a new Object Domain and must have the following key-value pairs:

- `name` : the name for the object domain, which will be used to refer to this domain
- `objects` : a list of object ID strings

The following example shows an Object Map defining two Object Domains that together contain four objects.

```
object_map:  
- name: ObjectDomain1  
  objects: [objectID1]  
- name: ObjectDomain2  
  objects: [objectID2, objectID3, objectID4]
```

Where each object ID is a unique string that corresponds to an object as identified in Section 5.

Each object in a program must have its object ID included in at most one Object Domain and the defined Object Domain names must all be unique. If a program object is not included in an Object Domain, then any access to that object is considered a violation. Object Domain names may contain alphanumeric characters plus the “_” and “.” characters and should not include any white space.

4.3 Subject Map

The top-level key `subject_map` is a list of Subject Domain Descriptors, each of which is of type `dictionary`.

Each such dictionary defines a new Subject Domain and must have the following key-value pairs:

- `name` : the name for the subject domain, which will be used to refer to this domain
- `subjects` : a list of subject ID strings

The following example shows an Subject Map defining two Subject Domains that together contain four subjects.

```
subject_map:  
- name: SubjectDomain1  
  subjects: [subjectID1, subjectID2]  
- name: SubjectDomain2  
  subjects: [subjectID3, subjectID4]
```

Where each subject ID is a unique string that corresponds to a subject as identified in Section 5.

Each subject in a program must have its subject ID included in at most one Subject Domain. The defined Subject Domain names must all be unique and must not collide with any Object Domain names. If a program subject (function) is not included in any Subject Domain, then any call to that subject is considered a violation. Subject Domain names may contain alphanumeric characters plus the “_” and “.” characters and should not include any white space.

4.4 Privileges

The value stored at the top-level key **privileges** is a list of Privilege Descriptors, each of which is of type **dictionary**.

Each Privilege Descriptor defines (1) a Principle, and (2) the privileges that principal in the system has. Each such Privilege Descriptor object must have the following key-value pairs:

- **principal** : a Principal Descriptor object
- **can_call** : a list of Subject Domain names that can be called by this principal
- **can_return** : a list of Subject Domain names that this principal can return to
- **can_read** : a list of Access Descriptor objects
- **can_write** : a list of Access Descriptor objects

Where a Principle Descriptor object is of type **dictionary** and has the following key-value pairs:

- **subject** : the name of a Subject Domain
- **execution_context** : a Context Descriptor object

Where a Context Descriptor object is of type **dictionary** and may contain zero or more context key-value pairs. The context identifier strings, mapping and interpretation are described in Section 6.

An Access Descriptor object is of type **dictionary** and has the following key-value pairs:

- **objects** : a list of Object Domain names

- `object_context` : a Context Descriptor object

The following example shows a Privilege section that contains an entry for just one principal:

```
privileges:
- principal:
  subject: SubjectDomain1
  execution_context:
    call_context: [subjID1, any]
    uid: U
  can_call: [SubjectDomain2]
  can_return: [SubjectDomain3]
  can_read:
- objects: [ObjectDomain1, ObjectDomain2]
  object_context:
    uid: U
- objects: [ObjectDomain4]
  can_write:
- objects: []
```

The privilege lists in a Privilege Descriptor (*e.g.*, `can_call`) may be empty, which indicates no allowed privileges of that operation type.

To deny all access, specify an empty list. For example the definition `- objects: []` in the above example explicitly states that that Principal cannot write any objects.

Only a single Privilege Descriptor can be present for each principal (combination of Subject Domain and Execution Context).

5 Subject and Object Identification

The subject and object identifier strings presented in Section 4 are used to uniquely map to a corresponding element of the program. Each such string should uniquely identify an element of the program, and each piece of code and data in the program should have exactly one unique identifier string that identifies it.

We present our naming and mapping scheme here.

5.1 Subject Identifiers

The following identification scheme is used for subjects (code):

- **Functions with symbols and sizes:** For functions that have a both a symbol name and a known size, the identifier for the function is taken to be the concatenation of: (1) the name of the containing compilation unit, (2) a pipe character, and (3) the name of the symbol, *e.g.*, “main.c|main”.
- **Functions without a symbol and size:** Any function that does not have both a symbol name and known size will be identified by the concatenation of: (1) the

name of the containing compilation unit, (2) a pipe character, and (3) the name of the originating file. This can occur when *e.g.*, assembly source code has functions that are not annotated with a size. All functions that originate from the same source file and do not have a size share the same identifier and are considered the same subject. Note that some functions in a file may have sizes and some may not; in this case the functions with sizes will be identified by the above clause and all remainders will be identified by this one and will be considered the same subject.

- **Dynamically generated code:** TBD

5.2 Object Identifiers

Object identifier strings consist of 4 values delimited by a pipe character (“|”). The first field is the *entity type*, which indicates which type of data entity the object is. The *entity type* field is required for all object identifier strings; the subsequent fields may be populated or left empty depending on the entity type as shown in Table 1. The second field is a *path* which for some entity types is used to identify where in the source code the object originates. The third field is the *line number* which is used for some entity types alongside the *path* to indicate a specific line of source code. The fourth field, the *name*, is used for some entity types to indicate a name that is used to identify the entity, such as a symbol name or device name.

An example object identifier string is “GLOBAL|/src/dir/file.c|42|foo”

	Entity type	Path	Line #	Name	Sub-objects?
Global Variable	GLOBAL	Path to compilation unit	Declaration line	Symbol name	✓
Heap Allocation	HEAP	Path to file containing allocator call	Line of allocator call	–	✓
Stack Frame	STACK_FRAME	Path to function	–	Function name	✓
Stack Region	STACK_REGION	Path to stack allocation	Line of allocator call	–	–
IO	IO	Path to device tree	Line where device declared	Device name	–
Other	OTHER	Optional	Optional	Optional	–

Table 1: Object identifier string fields

5.2.1 SubObject Identifiers

For entity types in Table 1 that have an “✓” in the rightmost column, there is additional syntax to refer to “subobjects”. For example, if a global variable is an `employee` struct,

we might want to refer to the `.last_name` or `.ssn` fields of that global as separate fine-grained objects.

Subobject references can be nested. For example, if an `employee` struct has a field `.name` that is itself a struct with fields `.last` and `.first`, then, if a local variable `emp` of function `update` in file `employee.c` is an `employee` struct, then we may refer to the last name using

```
STACK_FRAME|src/dir/employee.c|update.emp.last
```

From Table 1, then, the format for specifying subobjects is

- GLOBAL: `... | <symbol name> . <field name> [. <field name>]*`
- HEAP: `... | <allocation line> . <field name> [. <field name>]*`
- STACK_FRAME: `... | <function name> . <local variable name> [. <field name>]*`

Intra-struct padding It is often the case that fields in a struct are separated by “padding” bytes so that field accesses are aligned according to native word size. Adding padding bytes is the compiler’s prerogative, and so of course different compilers, and different architectures, will result in different amounts of padding bytes for the same source code structure. When the compiler generates instructions to access non-first fields in a struct, it will take into account any padding bytes used to lay out the structure in memory.

A compartmentalization specification, possibly informed by observed behavior, may want to specifically allow or disallow access to padding bytes.

For example, there may be some code that treats a struct as one big byte array – e.g. for copying a struct. And there may be other code that accesses individual fields and stays within the declared size bounds of those fields. There may be some code that is observed to not stay strictly within declared field bounds when accessing fields of a struct.

The above three situations can be addressed symbolically when defining objects (to be used to grant access) as follows:

1. All bytes of a structure: Refer to the struct name. E.g. `emp`
2. Individual fields, within declared bounds: Refer to `struct_name.field_name`. E.g. `emp.last_name`
3. Padding bytes – to allow access outside of declared struct field bounds but within the bounds of the structure as a whole, again just refer to the name bound to the struct. E.g. `emp`.

Casting In C, *casting* is used to inform the compiler how to treat an otherwise untyped array of bytes. In the context of subobject references in compartmentalization specifications, it is a challenge for automatic compartmentalization tools to relate low-level pointer offsets to relevant source code casts. Regardless of how the symbolic field names used in compartmentalization specifications were derived, however, enforcement mechanisms should face no ambiguity, modulo reasoning about padding, as to what are the offsets to be used when checking object accesses.

6 Context

In this section we define the set of context key-value pairs used by Context Descriptor objects (Section 4). For each context key, we define the allowed values when specified under both the Execution Context and Object Context. We define two broad classes of context specifiers: those that are universal and may apply to any program (Section 6.1) and those that are specific to the Linux kernel (Section 6.2). The CPM compartmentalization format may be extended to other application domains or systems by defining new context specifiers suitable for that system.

Any context keys not explicitly set in a Context Descriptor are assumed to be wildcard *e.g.*, apply to all execution contexts.

6.1 Universal Context Specifiers

6.1.1 Call Context

The `call_context` context key refers to the current call stack. It has a value of list type, where each element is either the identifier string of a function in the program or the special wildcard `all` value. The first item is interpreted as the base of the stack (first stack frame), with each subsequent list item indicating a nested call higher in the stack. The special wildcard value matches any number of (or zero) stack frames. This key-value pair is allowed in both the Execution Context and the Object Context.

For example:

- `[main.c|main, main.c|check_user_password]` matches one call stack: `main` as the first called function at the bottom of the stack with one callee `check_user_password`.
- `[main.c|main, all]` matches all call stacks where `main` is the first called function (including just `main`).
- `[all, string.h|strcmp]` matches all call stacks where `strcmp` is at the top of the stack (currently executing function) no matter what calls took place prior.

6.2 Linux Kernel Context Specifiers

6.2.1 uid

The `uid` context key refers to the effective uid (user id) indicated in the `task_struct` of the currently executing task. It is a scalar string type and may be set to one of these values:

- “root” : the task belongs to the root user (uid of 0). Allowed in Execution Context and Object Contexts.
- “user” : the task belongs to any user except the root user (uid \neq 0). Allowed in Execution Context and Object Contexts.
- “all” : matches any uid (same as not specifying). Allowed in Execution Context and Object Contexts.

- “<var name>”: In an Execution Context, matches any `uid` value and binds the variable name to that value. In an Object Context, refers to the `uid` bound in the Execution Context.

6.2.2 gid

The `gid` context specifier refers to the effective `gid` (group id) indicated in the `task_struct` of the currently executing task. It is a scalar string type and may be set to one of these values:

- “all” : matches any `gid` (same as not specifying). Allowed in Execution Context and Object Contexts.
- “<var name>”: In an Execution Context, matches any `gid` value and binds the variable name to that value. In an Object Context, refers to the `gid` bound in the Execution Context.

Note that the `gid` is defined differently from the `uid` because there is no safe assumption that the group with `gid 0` is known to be a special root value. As such, the only allowed context operation for the `gid` is checking that the object was allocated by the same group that is performing an access.

7 Grammar for Privileges

Table 2 presents an informal grammar using a mix of *BNF* and YAML syntax. Italicized *Names* are types (*nonterminals* of the grammar). We use YAML “flow style” syntax. For example, “[*Thing*]” is a comma-separated list (“sequence” in YAML terms) of objects of type *Thing*, and “{ field1: *Type1*; field2: *Type2*; }” is a record (“map” in YAML terms) with field names (aka “keys”) “field1” and “field2”, of types *Type1* and *Type2*, respectively. Lists may be empty, specified as “[]”. Note that YAML can be expressed using either “block” or “flow” styles - most YAML libraries can deal with both styles. For example, a YAML list in block style is a sequence of equally-indented lines each beginning with a “-”, whereas a YAML list in flow style is a comma-delimited sequence surrounded by “[]”s.

A production such as “*NonTerm* ::= [*Type*] | all” means that a *NonTerm* may be defined either by a list of *Types* or by the token “all”.

ObjectDomainName means an identifier that corresponds to the “name:” field given in an earlier definition of an *ObjectDomain*. Similarly for *SubjectDomainName*.

The rightmost two columns clarify how to specify either “allow none” or “allow all” for a given field.

If a field is not required, the production defining that field is prefixed with a “?”, and the “all” column is the default value for that field. That is, the omission of a field in the YAML is equivalent to specifying the field with the given “all” value.

Type	Value(s) for type	none	all
<i>Spec</i> ::=	{ object_map: [<i>ObjectDomain</i>], subject_map: [<i>SubjectDomain</i>], privileges: [<i>Privilege</i>] }		
<i>ObjectDomain</i> ::=	{ name: <i>String</i> , objects: [<i>ObjectId</i>] }		
<i>SubjectDomain</i> ::=	{ name: <i>String</i> , subjects: [<i>SubjectId</i>] }		
<i>Privilege</i> ::=	{ principal: <i>Principal</i> , ? can_call: [<i>SubjectDomainName</i>] all, ? can_return: [<i>SubjectDomainName</i>] all, ? can_read: [<i>Object</i>] all, ? can_write: [<i>Object</i>] all }	[] [] [] []	all all all all
<i>Principal</i> ::=	{ subject: <i>SubjectDomain</i> , ? execution_context: <i>Context</i> all }		all
<i>Object</i> ::=	{ objects: [<i>ObjectDomainName</i>] all, ? object_context: <i>Context</i> all }	[]	all all
<i>Context</i> ::=	{ ? call_context: [<i>SubjectDomainName</i> all], ? uid: root user <i>Variable</i> all, ? guid: <i>Variable</i> all }	[] [] []	[all] all all

Table 2: Grammar for YAML specification of compartmentalization policies

7.1 A Note on omitted fields, empty fields, allow all, and allow none

As mentioned above, fields with a “?” before the production can be omitted entirely, and that is equivalent to having included the field with the “allow all” value given in the rightmost column. This is to be consistent with the notion of “subsetting” the compartmentalization grammar, in which case omitted elements are not tracked/matched.

For privileges, we want to specify whether to (1) “allow all” (e.g. allow any execution context), (2) “allow none” (e.g. to disallow reading any objects), or (3) allow access to a specific set. The “none” and “all” columns in Table 2 make clear how to match no or all elements. For some privilege fields, “allow none” does not make sense. For example, specifying “none” for an execution context makes no sense; it would be tantamount to disallowing the given Principle from doing anything. In that case, one can simply not mention the given Subject Domain at all. Similarly for call_context - specifying “none” would mean that the Subject Domain or Objects involved have no privileges – in which case, why mention them in the first place? For fields with no value in the “none” column, we claim that “allow none” does not make sense in any case for that field.

Finally, it is possible to give a field name, followed by a colon and then end of line. That is interpreted as equivalent to the value in the “none” column. For example, “can_read:” (followed by end of line) is equivalent to “can_read: []”. This is meant to be consistent with YAML, where an empty field is interpreted as the “null” value.

To be clear, for a Principal (combination of a Subject Domain and Execution Context) defined in a compartmentalization specification, omission of a privilege field specification is equivalent to not tracking the corresponding operation/access, and so is equivalent

to “allow all”. However, for all Principals not given *some* privileges, then all tracked privileges are disallowed, that is, “allow none”.

8 Specification Consistency Checking

There are a number of “sanity checks” that can be performed on the YAML provided as a compartmentalization policy and subsetting spec. These checks include:

1. field names must match the grammar in Table 2
2. references to Subject Domains and Object Domains in privileges must correspond to Subject and Object Domains defined in the `object_map` and `subject_map` fields of the spec
3. empty fields must have a defined “none” value in Table 2

When counts are added, there are additional, obvious consistency checks.

Draper has developed a tool for ingesting YAML, checking for consistency, and, optionally, emitting the YAML with all defaulted fields explicitly specified. In the future this will be added to the public GitHub.

9 Extension: Runtime Counts

With only minor adjustments to the format, the number of dynamic uses of each privilege can be encoded using the same formulation of Objects, Subjects, Principals and Contexts. This enables the format to additionally encode dynamic privilege traces of a running system, aligning with the goal of having a privilege and compartmentalization formulation that can be learned or extracted from a running system.

9.1 Format Extension

For this extension, additional count fields are augmented to the Privileges section of the format to annotate privileges with their runtime usages. This makes the trace format a superset of the base format while maintaining the property that the trace format also parses as a valid interchange format.

Two additional fields are added to the Privilege Descriptor object:

1. `call_counts` : a list of equal length to the `can_call` list. Each element of the list represents the number of times the corresponding Subject Domain at the same index was called.
2. `return_counts` : a list of equal length to the `can_return` list. Each element of the list represents the number of times the corresponding Subject Domain at the same index was returned to.

An additional field is added to the Access Descriptor object, which is found in both the `can_read` and `can_write` lists.

1. **counts** : a list of equal length to the **objects** list in the containing Access Descriptor. Each element of the list represents the number of times the corresponding Object Domain was accessed.

Note that a typical trace file would define a Subject Domain for each function and an Object Domain for each primitive object (“reflexive domains”) to record the privilege uses at the finest granularity, which may then be post-processed by analysis tools.

When a CPM compartmentalization file has no count extensions (no additional count fields) and is interpreted as a trace, an implicit count of 1 for each privilege listed is assumed. This assumption enables privileges derived from static analysis with no runtime counts to be combined with dynamic traces.

9.2 Example

Below is a complete example of a Privilege Descriptor augmented with the runtime counts. It assumes that there are some Subject Domains and Object Domains already defined.

```
- principal:
  subject: SubjectDomain1
  execution_context:
    uid: user
  can_call:
  - SubjectDomain2
  - SubjectDomain3
  call_counts:
  - 594
  - 433
  can_return:
  - SubjectDomain2
  return_counts:
  - 990
  can_read:
  - objects:
    - ObjectDomain1
    - ObjectDomain3
    counts:
    - 348
    - 141
  can_write:
  - objects: []
    counts: []
```

10 Extension: Object and Subject Sizes

Similar to the runtime counts extension, the format can be extended to include object and subject sizes (both measured in bytes). These can be used for privilege calculations

that include a size component.

10.1 Format Extension

For this extension, an additional **size** field is augmented to each Object Domain Descriptor and each Subject Domain Descriptor.

In an Object Domain Descriptor, the **size** key should contain a list of integers of equal length to the **objects** list. Each such integer corresponds to the size of the corresponding object at the same index.

In an Subject Domain Descriptor, the **size** key should contain a list of integers of equal length to the **subjects** list. Each such integer corresponds to the size of the corresponding subject at the same index.

10.2 Size Definitions

Object and subject sizes are both measured in bytes. The sizes should correspond to the sizes of those entities in the final system and thus may depend on the compiler and other operating conditions. Subject sizes are taken to be the number of bytes of code in the corresponding function or assembly routine. Object sizes are taken to be the size in bytes of the data entity.

Dynamically allocated objects need special treatment for determining size. The size extension is intended for privilege calculations, and thus we choose a definition that captures the intent of our privilege metrics. For the HEAP entity type, we take the size to be the maximum number of bytes that are live at once from that allocation site over all observed traces. For the STACK_FRAME entity type, we take the size to be the size of the stack frame times the maximum number of observed live instances of that frame over all observed traces.

10.3 Example

Below is an example that includes object and subject sizes:

```
object_map:
- name: UserPassword
  objects: [main.c|user_password, main.c|admin_password]
  sizes: [64, 64]
...
subject_map:
- name: CheckUserPasword
  subjects: [main.c|user_check_password]
  sizes: [140]
```

11 Planned Extensions

This is an early version of the CPM compartmentalization file format. We plan to add additional context specifiers, such as:

1. namespace
2. cgroup
3. process lineage

We plan to add additional subject and object identifiers to address all data used by the Linux kernel, including:

1. memblock memory
2. per-cpu variables
3. memory mapped IO
4. dynamically loaded modules

12 Tool Coordination Options – options.yaml

For any large system, there is a huge number of potential compartmentalization specifications, even restricting consideration to those that admit all observed traces.

Thus compartmentalization generation tools must be aware of what compartmentalization policies can be efficiently implemented by a given enforcement mechanism, in order to minimize privilege while staying within some performance overhead bounds.

We adopt the convention that any file ending in `options.yaml` will contain a YAML key-value dictionary of keys meant to coordinate the generation of policies with a particular enforcement mechanism.

Following is an initial list of keys and values for `options.yaml`. We expect that the list will grow over time.

not-supported Values: names in grammar from Table 2 that have a “?” prefix. For example, if an enforcement mechanism cannot track call context for calls/returns, or allocation context for objects, an `options.yaml` file would include

```
not-supported: [execution_context, object_context]
```