```
#def LEN 1 // parameterize pwd length
char* master_pwd;

void listen() {
  char pwd[LEN];
  while (1) {
    gets(pwd);
    // call external log library
    log("launch attempt");
    if (check_pwd(pwd)) {
      __sys_fire_missiles();
    }
  }
}
```

```
void init() {
  master_pwd = malloc(sizeof(int)*LEN);
  for (int i = 0; i < LEN; i++) {
    master_pwd[i] = ’0’;
  }
}

int check_pwd(int* pwd) {
  for (int i = 0; i < LEN; i++) {
    if (pwd[i] != master_pwd[i])
      return 0; // bad password
  }
  return 1; // report success
}
```

Figure 1: Example Password Checker

1 Introduction

In this paper we introduce a novel compartmentalization scheme that distinguishes between memory that is local to compartments, and memory shared between them by memory-safe pointers. Our Tagged C implementation places fewer requirements on the underlying tagged hardware that similar systems from the literature. We describe the specification of the policy in terms of an abstract machine that gives definition to many memory undefined-behaviors if they are internal to a compartment, allowing it to describe the behavior of the system in the presence of UB more precisely. We then prove that Tagged C, running our compartmentalization policy, preserves all properties of the abstract machine.

1.1 Motivating Example

Consider a program for firing missiles that listens on standard input for a password, checks it against the master password in its own memory, and reports whether it was correct. If correct, it launches missiles with a system call. It also logs that it recieved an attempted launch before checking, using an off-the-shelf logging library.

Should the logging library have a vulnerability, it might be used to undermine the security of the whole system. The harm that it can do is not obvious, assuming the basic restriction that it cannot call `__sys_fire_missiles` directly, but in a memory unsafe setting it could easily overwrite `master_pwd` to make it match the supplied password, or leak its value to be used in a future attempt.

One approach to making the system secure is to compartmentalize it. All of the above code is gathered in one unit, which we will call A, and kept separate from the logging library, which belongs to its own compartment, B. We might further keep the standard library in yet another compartment, C. The compartments' memories are kept disjoint, except that `gets` must take a pointer to an array that will be accessible to both A and C. (The string argument to `log` will also be a shared global, but this is less interesting.) The special `__sys_fire_missiles` function is modeled as an external call, outside of both compartments.

How do we know when this compartmentalization is successful? For this specific program, we could try to prove that it satisfies a particular *property* describing its dynamic behavior. In English, that property would be, “I only fire the missiles after recieving the valid password.” But we can’t prove that without first getting it into a more rigorous form. For that, we need a *trace semantics* that provides a simple description of how compartments talk to one another in a given execution. This semantics needs enable reasoning about program behavior in the presence of shared memory.

Our contributions are as follows:

- A novel compartmentalization policy for Tagged C that supports cross-compartment sharing with fewer constraints on available tags than similar systems from the literature.
- A formal model of C compartmentalization in the form of an abstract machine that supports sharing between compartments while keeping their memories isolated by construction.
- A proof that the compartmentalization policy is safe with respect to the abstract semantics.

1.2 Related Work

There are many compartmentalization mechanisms in the literature, and several formal characterizations of compartmentalized systems.

Abate et al. [] characterize compartmentalization in the presence of undefined behavior, treating UB as equivalent to compromise by an adversary. Their model does not support shared memory, and their policy places strong constraints on potential hardware implementations: they assume linear tags, which are unlikely to appear in realistic implementations.

Building on their work, El-Korashy et al. [] present a compartmentalization scheme that does support shared memory. They use a much more restrictive specification than Abate et al.: their source language is fully memory-safe and has no UB. Their focus is on showing that they can enforce their compartmentalized calling convention in a memory safe setting. They are not focused on the underlying enforcement hardware, but we can assume that any hardware implementation would need to be of similar complexity to others capable of enforcing full memory-safety, e.g. Acevedo de Amorim et al. [].

Thibault et al. [] go deeper into their prove effort: they prove safe compilation of a compartmentalized version of CompCert C down to a compartmentalized assembly language. In the process they give up sharing. Their treatment of UB is similar to Abate et al., but they add new UB in the form of violations of the compartment interfaces.

Compared to these works in total, our model supports cross-compartment sharing while placing fewer constraints on the hardware than El-Korashy et al. We also support a full C setting, though instead of a full compilation chain we attach our policy directly to the Tagged C source semantics. Our abstract machine is also more precise about its treatment of UB than the others: it gives definition to some UB, and so can be used to reason about the behavior of compartments that contain it but still display consistent internal behavior.

2 Abstract Sharing Semantics

In the example above, we discussed the security of a compartmentalized system in terms of which compartments have recieved pointers to which memory. We define a C semantics that builds this kind of reasoning into the memory model. The model aims to fulfill a few key criteria:

$$C \in \mathcal{C} \quad id \in ident$$

$$\begin{array}{rl}
m \in & mem \\
empty \in & mem \\
read \in & mem \to int \rightharpoonup val \\
write \in & mem \to int \to val \rightharpoonup mem \\
M ::= & \{ms \in \mathcal{C} + ident \rightharpoonup mem; \\
& \; stk \in list(\mathcal{C} + ident \times int \times int); \\
& \; heap \in list(\mathcal{C} + ident \times int \times int)\} \\
heap\_alloc \in & \mathcal{M} \to \mathcal{C} + ident \to int \rightharpoonup (int \times \mathcal{M}) \\
heap\_free \in & \mathcal{M} \to \mathcal{C} + ident \to int \rightharpoonup \mathcal{M} \\
stk\_alloc \in & \mathcal{M} \to \mathcal{C} + ident \to int \rightharpoonup (int \times \mathcal{M}) \\
stk\_free \in & \mathcal{M} \to \mathcal{C} + ident \rightharpoonup \mathcal{M} \\
perturb \in & \mathcal{M} \rightharpoonup \mathcal{M}
\end{array}$$

$$\begin{array}{rll}
\mathcal{R} ::= & \mathbf{L}(C) \mid \mathbf{S}(id, base) & base \in int \\
v \in val ::= & \ldots \mid Vptr\ r\ a & r \in \mathcal{R}, a \in int \\
e \in & ident \rightharpoonup (\mathcal{R} \times int) &
\end{array}$$

$$\begin{array}{rl}
state ::= & C, M, e, \ldots \mid \texttt{expr} \\
\mid & C, M, e, \ldots \mid \texttt{stmt} \\
\mid & CALL(f, M, \ldots) \\
\mid & RET(M, v, \ldots)
\end{array}$$

$$(\longrightarrow) \in state \times state$$

Figure 2: Definitions

- Compartments are obviously and intuitively isolated from one another by construction
- It is suitable for hardware enforcement without placing intensive constraints on the target
- Inter-compartment interactions via shared memory are possible
- Compartments can only access shared memory if they have first obtained a valid pointer to it, consistent with the C standard and "capability reasoning"

We don't necessarily care that compartments' internal behavior conforms to the C standard. In fact our model explicitly gives compartments a concrete view of memory, giving definition to code that would be undefined behavior in the standard, such as described in Memarian et al. [1].

Figure 2 defines the memory model that we use in our abstract machine. We separate the world into compartments, ranged over by C, each with its own distinct concrete memory. A concrete memory m partially maps machine integer (int) addresses to values with a basic axiomatization given in Figure 3. One memory of this kind is assigned to each compartment, and each shared object is allocated in its own separate concrete memory.

Memories are kept totally separate, fulfilling our first requirement: compartments' local memories are definitely never accessible to other compartments. Shared memories are only accessible via valid pointers.

Pointers into local and shared memories are distinguished from one another by construction. A pointer value consists of a pair: a region ranged over by r drawn from the set $\mathcal{R}$ that determines which memory it accesses, and a machine integer address representing its concrete position. A region is either $\mathbf{L}(C)$, for local pointers into the compartment C, or $\mathbf{S}(id, base)$, where id is an abstract identifier and $base$ is a machine integer. Regions are identified in general by their compartment identifiers or abstract identifiers, collectively the set $\mathcal{C} + ident$. A "super-memory" M is a record containing a map from region identities to memories, ms, and lists of allocated regions for both stack and heap, stk and $heap$.

$$\textbf{WR1}: \textit{write}\ m\ a\ v = m' \to \textit{read}\ m\ a = v$$

$$\textbf{WR2}: \textit{read}\ m\ a = v \to \textit{write}\ m\ a'\ v' = m' \to a \neq a' \to \textit{read}\ m'\ a = v$$

$$\frac{M.\textit{ms}\ C = m \qquad \textit{read}\ m\ a = v}{C, M, e \mid *(\textit{Vptr}\ \mathbf{L}(C)\ a) \longrightarrow C, M, e \mid v}\textsc{ValOfL}$$

$$\frac{M.\textit{ms}\ \textit{id} = m \qquad \textit{read}\ m\ a = v}{C, M, e \mid *(\textit{Vptr}\ \mathbf{S}(\textit{id}, \textit{base})\ a) \longrightarrow C, M, e \mid v}\textsc{ValOfS}$$

$$\frac{M.\textit{ms}\ C = m \quad \textit{write}\ m\ a\ v = m' \quad M' = M[\textit{ms}\ m \mapsto m']}{C, M, e \mid *(\textit{Vptr}\ \mathbf{L}(C)\ a) := v \longrightarrow C, M', e \mid v}\textsc{AssignL}$$

$$\frac{M.\textit{ms}\ \textit{id} = m \quad \textit{write}\ m\ a\ v = m' \quad M' = M[\textit{ms}\ \textit{id} \mapsto m']}{C, M, e \mid *(\textit{Vptr}\ \mathbf{S}(\textit{id}, \textit{base})\ a) := v \longrightarrow C, M', e \mid v}\textsc{AssignS}$$

Figure 3: Reads and Writes

$M.\textit{heap}$ and $M.\textit{stk}$ are lists of triples (r, a_1, a_2) representing the allocated regions of the heap and stack, respectively. Once an object is allocated within a region, reads and writes are guaranteed to succeed within its bounds in that region's memory. Reads and writes to unallocated regions may failstop, but if they do not the behave consistently with the axioms in Figure 3. The allocation and free operations for both stack and heap act on the super-memory as axiomatized in Figure 4.

This axiomatization serves to abstract away concrete details about memory layout that may be specific to a given compiler-allocator-hardware combination. We can understand any particular instance of $\mathcal{M}$ as an oracle that divines where the target system will place each allocation and, with knowledge of the full layout of memory, determines what happens in the event of an out-of-bounds read or write.

Using a flat memory model inside of compartments gives two benefits. First, it is generally less expensive to enforce, and therefore can be implemented on tagged hardware that is more restrictive. Second, it allows the abstract machine to reason about how programs with some forms of memory UB will behave after compilation, given a particular compiler and allocator.

On a call to `malloc` in compartment C, the allocation oracle α provides a base address that doesn't overlap with an allocated object in that memory, and the appropriately-sized region starting at that base address is allocated in $M[C]$. A call to `malloc_shared`, meanwhile, generates a fresh block identifier b and returns the pointer $(b, 0)$; however, such a call may fail due to the system as a whole running out of memory.

Values now include two kinds of pointer: block-offset pairs that access the shared memory, and pairs of compartment identifiers and integers that access that compartment's memory.

$$v ::= \cdots \mid \textit{sptr}\ b\ \textit{off} \mid \textit{lptr}\ \textit{addr}$$

The latter local pointers are subject to several restrictions, namely that they cannot be passed or returned across a compartment boundary, nor written to shared memory. Doing so is undefined behavior. This means that they will only ever be accessible to the compartment that owns them. [SNA: Note: they could alternatively be forbidden from being recieved/read.]

Allocation The abstract operations $\textit{heap\_alloc}$ and $\textit{stk\_alloc}$ yield addresses at which they locate a newly allocated object, either within a compartment's local region or in its own isolated region. In the latter case, the address provided becomes the base tracked by that region's pointers. Since

$$\textbf{HA}: \mathit{heap\_alloc}\ M\ r\ sz = (a, M') \to M'.\mathit{heap} = (r, a, a+sz) :: M.\mathit{heap} \wedge M'.\mathit{stk} = M.\mathit{stk}$$

$$\textbf{HF}: (r, a_1, a_2) \in M.\mathit{heap} \to \exists M'.\mathit{heap\_free}\ M\ B\ a_1 = M' \wedge M'.\mathit{heap} = M.\mathit{heap} - (r, a_1, a_2) \wedge M'.\mathit{stk} = M.\mathit{stk}$$

$$\textbf{AM}: (r, a_1, a_2) \in M.\mathit{heap} \cup M.\mathit{stk} \to \exists! m. M.\mathit{ms}\ r = m$$

$$\textbf{AR}: (r, a_1, a_2) \in M.\mathit{heap} \cup M.\mathit{stk} \to m = M.\mathit{ms}\ r \to a_1 \le a < a_2 \to \exists v.\mathit{read}\ m\ a = v$$

$$\textbf{AW}: (r, a_1, a_2) \in M.\mathit{heap} \cup M.\mathit{stk} \to m = M.\mathit{ms}\ r \to a_1 \le a < a_2 \to \exists m'.\mathit{write}\ m\ a\ v = m'$$

$$\textbf{DISJ}: (r, a_1, a_2) \in M.\mathit{heap} \cup M.\mathit{stk} \to (r, a_1', a_2') \in M.\mathit{heap} \cup M.\mathit{stk} \to a' + sz < a_1 \vee a_2 \le a'$$

$$\textbf{PERT1}: \mathit{perturb}\ M = M' \to M'.\mathit{heap} = M.\mathit{heap} \wedge M'.\mathit{stk} = H.\mathit{stk}$$

$$\frac{\mathit{expr} = \texttt{malloc}(\mathit{Vint}\ sz) \qquad \mathit{heap\_alloc}\ M\ C\ sz = (a, M')}{C, M, e \mid \mathit{expr} \longrightarrow C, M', e \mid \mathit{Vptr}\ \mathbf{L}(C)\ a}\ \textsc{MallocL}$$

$$\frac{\mathit{fresh}\ id \qquad \mathit{heap\_alloc}\ M\ id\ sz = (a, M') \qquad \mathit{expr} = \texttt{malloc\_share}(\mathit{Vint}\ sz)}{C, M, e \mid \mathit{expr} \longrightarrow C, M', e \mid \mathit{Vptr}\ \mathbf{S}(id, a)\ a}\ \textsc{MallocS}$$

$$\frac{v = \mathit{Vptr}\ \mathbf{L}(C)\ a \qquad \mathit{heap\_free}\ M\ C\ a = M'}{C, M, e \mid \texttt{free}(v) \longrightarrow C, M', e \mid \mathit{Vundef}}\ \textsc{FreeL}$$

$$\frac{v = \mathit{Vptr}\ (\mathbf{S}(id, \mathit{base}))\ \mathit{base} \qquad \mathit{heap\_free}\ M\ id\ a = M'}{C, M, e \mid \texttt{free}(v) \longrightarrow C, M', e \mid \mathit{Vundef}}\ \textsc{FreeS}$$

$$\textbf{SA}: \mathit{stk\_alloc}\ M\ r\ sz = (a, M') \to M'.\mathit{stk} = (r, a, a+sz) :: M.\mathit{stk} \wedge M'.\mathit{heap} = M.\mathit{heap}$$

$$\textbf{SF}: M.\mathit{stk} = (r, a_1, a_2) :: S' \to \exists M'.\mathit{stk\_free}\ M\ r\ a_1 = M' \wedge M'.\mathit{stk} = S' \wedge M'.\mathit{heap} = M.\mathit{heap}$$

$$\textbf{PERT2}: (r, a_1, a_2) \in M.\mathit{heap} \cup M.\mathit{stk} \to \mathit{perturb}\ M = M' \to a_1 \le a < a_2 \to \mathit{read}\ (M.\mathit{ms}\ r)\ a = v \to \mathit{read}\ (M'.\mathit{ms}\ r)\ a = v$$

Figure 4: Allocation Steps and Axioms

$$\frac{}{C, M, e \mid \odot(\mathit{Vptr}\ I\ a) \longrightarrow C, M, e \mid \mathit{Vptr}\ I\ (\langle\odot\rangle a)}\textsc{Unop}$$

$$\frac{}{C, M, e \mid (\mathit{Vptr}\ I\ a) \oplus (\mathit{Vint}\ i) \longrightarrow C, M, e \mid \mathit{Vptr}\ I\ (a\langle\oplus\rangle i)}\textsc{BinopPointerInteger}$$

$$\frac{}{C, M, e \mid (\mathit{Vint}\ i) \oplus (\mathit{Vptr}\ I\ a) \longrightarrow C, M, e \mid \mathit{Vptr}\ I\ (i\langle\oplus\rangle a)}\textsc{BinopIntegerPointer}$$

$$\frac{}{C, M, e \mid (\mathit{Vptr}\ I\ a_1) \oplus (\mathit{Vptr}\ I\ a_2) \longrightarrow C, M, e \mid \mathit{Vint}\ (a_1\langle\oplus\rangle a_2)}\textsc{BinopPointers}$$

Figure 5: Arithmetic Operations Involving Pointers

the $*\_alloc$ operations are parameterized by the identity of the region they are allocating within, they are allowed to make decisions based on that information, such as clumping compartment-local allocations together to protect them using a page-table-based enforcement mechanism. This is a nondeterministic semantics, but given any particular instantiation of the oracle, the semantics becomes deterministic.

Allocations are guaranteed to be disjoint from any prior allocations in the same region. (In fact, when targeting a system with a single address space, we further restrict them to be disjoint across all bases.) Addresses in allocated regions are guaranteed successful loads and stores, and once an unallocated address has been successfully accessed it will behave consistently until new memory is allocated anywhere in the system, at which point all unallocated memory again becomes unpredictable.

The *perturb* operation similarly represents the possibility of compiler-generated code using unallocated memory and therefore changing its value or rendering it inaccessible. The only facts that are maintained over a call to perturb are those involving addresses in allocated regions. Perturb happens during every function call and return, because the compiler needs to be free to reallocate memory during those operations, but it may happen at other points in the semantics as well.

Arithmetic and Integer-Pointer Casts Most arithmetic operations are typical of C. The interesting operations are those involving integers that have been cast from pointers. We give concrete definitions to all such operations based on their address. As shown in Figure 5, if they involve only a single former pointer, the result will also be a pointer into the same memory; otherwise the result is a plain integer. If the former pointer is cast back to a pointer type, it retains its value and is once again a valid pointer. Otherwise, if an integer value is cast to a pointer, the result is always a local pointer to the active compartment.

Calls and Returns There are two interesting details of the call and return semantics: they allocate and deallocate memory, and they can cross compartment boundaries. In the first case, we need to pay attention to which stack-allocated objects are to be shared. This can again be done using escape analysis: objects whose references never escape, can be allocated locally. Objects whose references escape to another compartment must be allocated as shared. Those that escape to another function in the same compartment can be treated in either way; if they are allocated locally but are later passed outside the compartment, the system will failstop at that later point

$$\mathit{alloc\_locals}\ M\ C\ [\,] = (M, \lambda id.\bot)$$

$$\mathit{alloc\_locals}\ M\ C\ (id, \mathbf{L}, sz) :: ls = (M'', e[id \mapsto (\mathbf{L}(C), i)])$$
$$\text{where } \mathit{stk\_alloc}\ M\ C\ sz = (i, M') \text{ and } \mathit{alloc\_locals}\ M'\ C\ ls = (M'', e)$$

$$\mathit{alloc\_locals}\ M\ C\ (id, \mathbf{S}, sz) :: ls = (M'', e[id \mapsto (\mathbf{S}(id, i), i)])$$
$$\text{where } \mathit{alloc\_locals}\ M\ C\ ls = (M', e) \text{ and } \mathit{stk\_alloc}\ M'\ C\ sz = (i, M'')$$

$$\frac{f = (C, \mathit{locals}, s) \quad \mathit{alloc\_locals}\ M\ C\ \mathit{locals} = (M', e)}{\mathit{CALL}(f, M) \longrightarrow C, \mathit{perturb}\ M', e \mid s}\textsc{FromCallstate}$$

Figure 6: Call Semantics and Local Variables

(see Section 3).

We assume that each local variable comes pre-annotated with how it should be allocated, with a simple flag **L** or **S**, so that a function signature is a list of tuples $(id, \mathbf{L} \mid \mathbf{S}, sz)$. (Aside: there is a case to be made we should just allocate all stack objects locally barring some critical use case for share them. Doing so would simplify the model here.)

The allocation and deallocation of stack memory is shown in the step rules in Section 6. In the full semantics, calls and returns step through intermediate states, written CALL and RET. During the step from the intermediate callstate into the function code proper, the semantics looks up the function being called and allocates its local variables before beginning to execute its statement. And during the step from the `return` statement into the intermediate returnstate, the semantics likewise deallocates every variable it had previously allocated.

3 Cross-compartment interfaces

In this system, each function is assigned to a compartment. A compartment interface is a subset of the functions in the compartment that are publicly accessible. At any given time, the compartment that contains the currently active function is considered the active compartment. It is illegal to call a private function in an inactive compartment.

Public functions may not receive $L(\dots)$-region pointer arguments. Private functions may take either kind of pointer as argument. $L(\dots)$ pointers may also not be stored to shared memory regions. This guarantees that there can be no confusion between shared pointers and a compartment's own local pointer that escaped its control. Violations of a compartment interface exhibit failstop behavior.

As a consequence of these rules, a compartment that recieves a pointer as an argument or loads it from shared memory may rely on it not aliasing with any local pointer. It could recieve such a pointer that has been cast to an integer type, but this is not a problem: if it casts it back to a pointer, the risks are the same as if it cast any other integer to a pointer.

$$\begin{array}{ll} \tau_V ::= \mathtt{L}(C)|\mathtt{S}(a)|\emptyset & \in \mathit{nat} \\ \tau_C ::= C & C \in \mathcal{C} \\ \tau_L ::= \mathtt{L}(C)|\mathtt{S}(a)|\emptyset & \\ \sigma := n & n \in \mathit{nat} \end{array}$$

4 Implementing Compartmentalization in Tagged C

A Tagged C policy defines three distinct types of tags: value tags drawn from set τ_V, control tags drawn from τ_C, and location tags from τ_L. It also requires a *policy state* type σ. It instantiates a set of *tag rules*, each of which parameterizes the behavior of key *control points* in the semantics. Tag rules are written in a procedural style, assigning tags to their outputs by name. The state $s : \sigma$ can always be accessed and assigned to.

In this policy, control tags represent compartments, with the special *PC tag* tracking the active compartment. Value tags distinguish between pointers that are local to each compartment, pointers to each shared object, and all other values. Location tags mark addresses as being allocated to a particular compartment or shared object, or else unallocated. The policy state is a counter that tracks the next allocation color.

```
**CallT**$(\mathcal{P}, pt)$
$\mathcal{P}' := pt$
```

```
**RetT**$(\mathcal{P}_{CLE}, \mathcal{P}_{CLR}, vt)$
$\mathcal{P}' := \mathcal{P}_{CLR}$
```

For simplicity, global variables are always local to a compartment. (Modifying the policy to allow multiple compartments to share global variables would be straightforward.) Dynamic memory

If it is lax, the allocated memory is tagged with the compartment identity only. But if it is strict, both the pointer and the memory location are tagged with the owning compartment and a fresh allocation color. Once we have a color attached to a pointer, it is propagated along with the pointer, including through any arithmetic operations provided that the other operand is not tagged as a pointer.

```
**LocalT**$(\mathcal{P}, \mathtt{ty}_{typ})$
let $\mathbf{comp}(C) := \mathcal{P}$ in
$vt' := \emptyset$;
if $\mathit{safe}\ C$
then  $pt' := \mathbf{clr}(C, s)$;
      $lt' := \mathbf{clr}(C, s)$;
      $s := s + 1$
else  $pt' := \mathbf{comp}(C)$;
      $lt' := \mathbf{comp}(C)$;
```

```
**MallocT**$(\mathcal{P}, pt, vt)$
let $\mathbf{comp}(C) := \mathcal{P}$ in
$vt' := \emptyset$;
if $\mathit{safe}\ C$
then  $pt' := \mathbf{clr}(C, s)$;
      $lt' := \mathbf{clr}(C, s)$;
      $s := s + 1$
else  $pt' := \mathbf{comp}(C)$;
      $lt' := \mathbf{comp}(C)$;
```

```
**GlobalT**$(\mathtt{x}_{glb}, \mathtt{ty}_{typ})$
$vt' := \emptyset$;
$lt' := \mathit{owner}(\mathtt{x})$
```

```
**BinopT**$(\oplus, \mathcal{P}, vt_1, vt_2)$
  case  $vt_1, vt_2$  of
  $\mathbf{clr}(C, a), \emptyset$
  $\emptyset, \mathbf{clr}(C, a)$     $\Rightarrow$   $\mathbf{clr}(C, a)$
  $\emptyset, \emptyset$              $\Rightarrow$   $\emptyset$
  $\_, \_$                            $\Rightarrow$   **fail**
```

Like allocations, loads and stores have different behavior depending on whether or not the active compartment is strict or lax. In a lax compartment, it merely compares the PC tag to the location tag of the target address. In a strict compartment, it also checks that the pointer color matches that of the target address.

```
$\textbf{LoadT}(\mathcal{P}, pt, vt, lt)$
let $\textbf{comp}(C) := \mathcal{P}$ in
if $strict\ C$
then $\textbf{assert}\ lt = \textbf{comp}(C)$
$vt' := vt$
else $\textbf{assert}\ \exists a.pt = \textbf{clr}(C, a) \wedge lt = \textbf{clr}(C, a)$
$vt' := vt$
```

```
$\textbf{StoreT}(\mathcal{P}, pt, vt, lt)$
let $\textbf{comp}(C) := \mathcal{P}$ in
if $strict\ C$
then $\textbf{assert}\ lt = \textbf{comp}(C)$
$\mathcal{P}' := \mathcal{P}; vt' := vt; lt' := lt$
else $\textbf{assert}\ \exists a.pt = \textbf{clr}(C, a) \wedge lt = \textbf{clr}(C, a)$
$\mathcal{P}' := \mathcal{P}; vt' := vt; lt' := lt$
```

5 Proving Correctness

Our correctness proofs relate the abstract machine defined above to the Tagged C semantics, instantiated with the policy above.

Bisimulation Proof We define the bisimulation relation $\mathfrak{R}$ between states of the abstract machine and states of Tagged C. We'll define the specific relation below. We first prove that, for any machine states S and $\hat{S}$, if $S\mathfrak{R}_\alpha\hat{S}$, then one of three cases holds:

- $S \longrightarrow S'$ and $\hat{S} \longrightarrow \hat{S}'$, S' and $\hat{S}'$ both have the same allocator state α', and $S'\ \mathfrak{R}_{\alpha'}\ \hat{S}'$. Memory trace events are either related, or internal.
- S and $\hat{S}$ are both stuck.
- $\hat{S}$ does not step due to failstop.

Then, by proving that for any program, its initial states in both machines are related by $\mathfrak{R}$, we can show by coinduction that every trace produced by one machine has a related trace produced by the other.

Consquences of Bisimulation

6 Machine Constraints

Now we consider the constraints that this system places on potential implementations. In particular, in a tag-based enforcemen mechanism with a limited quantity of tags, is this system realistic? In general it requires a unique tag per compartment, as well as one for each shared allocation. In the extreme, consider a system along the lines of ARM's MTE, which has four-bit tags. That could only enforce this semantics for a very small program, or one with very little shared memory (fewer than sixteen tags, so perhaps two-four compartments and around a dozen shared objects.)

On the other hand, this semantics is a reasonable goal under an enforcement mechanism with even eight-bit tags (512 compartments and shared objects.) If we go up to sixteen bits, we can support programs with thousands of shared objects.

That said, it only takes a minor adjustment for this model to be enforceable in even the smallest of tag-spaces. Instead of separate dynamic blocks for each shared object, we let the set of memory

$$\begin{aligned}
\mathit{val} &::= \ldots \mid \mathit{Vptr}\ r\ a \qquad r \in 2^{\mathcal{C}} \\
e &\in \mathit{ident} \rightharpoonup (2^{\mathcal{C}} \times \mathit{int}) \\
M &\in \mathcal{M} \subseteq 2^{\mathcal{C}} \to \mathit{mem} \\
\mathit{heap\_alloc} &\in \mathcal{M} \to 2^{\mathcal{C}} \to \mathit{int} \rightharpoonup (\mathit{int} \times \mathcal{M}) \\
\mathit{heap\_free} &\in \mathcal{M} \to 2^{\mathcal{C}} \to \mathit{int} \rightharpoonup \mathcal{M} \\
\mathit{stk\_alloc} &\in \mathcal{M} \to 2^{\mathcal{C}} \to \mathit{int} \rightharpoonup (\mathit{int} \times \mathcal{M}) \\
\mathit{stk\_free} &\in \mathcal{M} \to 2^{\mathcal{C}} \rightharpoonup \mathcal{M}
\end{aligned}$$

$$\frac{M\ r = m \qquad \mathit{read}\ m\ a = v}{C, M, e \mid *(\mathit{Vptr}\ r\ a) \longrightarrow C, M, e \mid v}$$

$$\frac{M\ r = m \quad \mathit{write}\ m\ a\ v = m' \quad M' = M[r \mapsto m']}{C, M, e \mid *(\mathit{Vptr}\ r\ a) := v \longrightarrow C, M', e \mid v}$$

$$\frac{\mathit{heap\_alloc}\ M\ r\ sz = (p, M')}{C, M, e \mid \texttt{malloc}_r(\mathit{Vint}\ sz) \longrightarrow C, M', e \mid \mathit{Vptr}\ r\ p}$$

Figure 7: Selected Rules for Explicit Sharing

regions consist of the powerset of compartments, each region corresponding to the set of compartments that have permission to access it. We parameterize each instance of `malloc` with such a set, which will be the base of each pointer that it allocates. We write the identifiers for these `malloc` invocations $\texttt{malloc}_r$. Then we replace the relevant definitions and step rules with those in Figure 7, with call and return steps changed similarly.

This version can be enforced with a number of tags equal to the number of different sharing combinations present in the system—in the worst case this would be exponential in the number of compartments, but in practice it can be tuned to be arbitrarily small. (In extremis, all shared objects can be grouped together to run on a machine with only two tags.) Sadly it strays from the C standard in its temporal memory safety: under some circumstances a shared object can be accessed by a compartment that it has not (yet) been shared with.

References

[1] Memarian, K., Gomes, V. B. F., Davis, B., Kell, S., Richardson, A., Watson, R. N. M., and Sewell, P. Exploring C semantics and pointer provenance. *Proc. ACM Program. Lang. 3*, POPL (Jan. 2019).