Efficient Compartments with Sharing in Tagged C

Sean Anderson

April 2, 2024

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 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:

```
#def LEN 1 // parameterize pwd length      void init() {
char* master_pwd;                            master_pwd = malloc(sizeof(int)*LEN);
                                             for (int i = 0; i < LEN; i++) {
void listen() {                                master_pwd[i] = '0';
  char pwd[LEN];                             }
  while (1) {                              }
    gets(pwd);
    // call external log library           int check_pwd(int* pwd) {
    log("launch attempt");                   for (int i = 0; i < LEN; i++) {
    if (check_pwd(pwd)) {                       if (pwd[i] != master_pwd[i])
      __sys_fire_missiles();                       return 0; // bad password
    }                                        }
  }                                          return 1; // report success
}                                          }
```

Figure 1: Example Password Checker

- 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. [1] 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. [3] 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. [2].

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.

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

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

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

$$
\begin{aligned}
state ::={}& C, M, e, \dots \mid \texttt{expr} \\
& \left| \begin{array}{l} C, M, e, \dots \mid \texttt{stmt} \\ CALL(f, M, \dots) \\ RET(M, v, \dots) \end{array} \right.
\end{aligned}
$$

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

Figure 2: Definitions

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:

- 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. [4].

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.

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

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

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

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

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

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

Figure 3: Reads and Writes [SNA: Is the $ms\ id \mapsto$ syntax acceptable?]

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 $\mathcal{C} + ident$. An environment e maps variable names to their regions and position. 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 $heap$.

$M.heap$ and $M.stk$ 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 in each region 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.

Allocation The abstract operations $heap\_alloc$ and 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 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

4

HA : $heap\_alloc\ M\ r\ sz = (a, M') \rightarrow$
$M'.heap = (r, a, a + sz) :: M.heap$
$\wedge M'.stk = M.stk$

HF : $(r, a_1, a_2) \in M.heap \rightarrow$
$\exists M'.heap\_free\ M\ B\ a_1 = M' \wedge$
$M'.heap = M.heap - (r, a_1, a_2) \wedge$
$M'.stk = M.stk$

AM : $(r, a_1, a_2) \in M.heap \cup M.stk \rightarrow$
$\exists! m.M.ms\ r = m$

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

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

DISJ : $(r, a_1, a_2) \in M.heap \cup M.stk \rightarrow$
$(r, a_1', a_2') \in M.heap \cup M.stk \rightarrow$
$a' + sz < a_1 \vee a_2 \le a'$

PERT1 : $perturb\ M = M' \rightarrow$
$M'.heap = M.heap \wedge M'.stk = H.stk$

$$\frac{\begin{array}{c} expr = \texttt{malloc}(Vint\ sz) \\ heap\_alloc\ M\ C\ sz = (a, M') \end{array}}{C, M, e \mid expr \longrightarrow C, M', e \mid Vptr\ \mathbf{L}(C)\ a} \text{\small MALLOCL}$$

$$\frac{\begin{array}{c} fresh\ id \quad heap\_alloc\ M\ id\ sz = (a, M') \\ expr = \texttt{malloc\_share}(Vint\ sz) \end{array}}{C, M, e \mid expr \longrightarrow C, M', e \mid Vptr\ \mathbf{S}(id, a)\ a} \text{\small MALLOCS}$$

$$\frac{v = Vptr\ \mathbf{L}(C)\ a \quad heap\_free\ M\ C\ a = M'}{C, M, e \mid \texttt{free}(v) \longrightarrow C, M', e \mid Vundef} \text{\small FREEL}$$

$$\frac{\begin{array}{c} v = Vptr\ (\mathbf{S}(id, base))\ base \\ heap\_free\ M\ id\ a = M' \end{array}}{C, M, e \mid \texttt{free}(v) \longrightarrow C, M', e \mid Vundef} \text{\small FREES}$$

SA : $stk\_alloc\ M\ r\ sz = (a, M') \rightarrow$
$M'.stk = (r, a, a + sz) :: M.stk \wedge$
$M'.heap = M.heap$

SF : $M.stk = (r, a_1, a_2) :: S' \rightarrow$
$\exists M'.stk\_free\ M\ r\ a_1 = M' \wedge$
$M'.stk = S' \wedge M'.heap = M.heap$

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

Figure 4: Allocation Steps and Axioms

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

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

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

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

Figure 5: Arithmetic Operations Involving Pointers

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 (see Section 2).

We assume that each local variable comes pre-annotated with how it should be allocated, with a simple flag $\mathbf{L}$ or $\mathbf{S}$, so that a function signature is a list of tuples $(id, \mathbf{L} \mid \mathbf{S}, sz)$. (Aside: there is a

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

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

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

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

Figure 6: Call Semantics and Local Variables

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 $\texttt{return}$ statement into the intermediate returnstate, the semantics likewise deallocates every variable it had previously allocated.

Cross-compartment interfaces In this system, each function is assigned to a compartment. We write the static mapping from function identifiers $\texttt{f}$ to compartments $comp(\texttt{f})$. Each compartment has an interface that is the subset of its functions that are publicly accessible. We write that a function $\texttt{f}$ is public in C as $pub(\texttt{f}, C)$.

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.

Local (stack allocated) variables are statically designated as shared or compartment-local. We write that $\texttt{x}$ is designated shared as $share(\texttt{x})$. For heap allocated objects, it is the call-sites to malloc that are statically designated.

7

3 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, function pointers (and which compartment their targets belong to), 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. These definitions are shown in Figure 7, with *comp* being a static mapping from function identifiers to compartment identifiers.

We implement the division between regions using the LocalT and MallocT tag rules (for stack and heap allocations, respectively). These rules tag the memory locations of freshly allocated objects with $\texttt{L}(C)$ or $\texttt{S}(a)$ depending on the sharing status of the variable being allocated or of the call to malloc. In the latter case, a is a fresh shared region identifier. They apply the same tag to the base pointer of the allocated object. 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.

On loads and stores, we compare the tag on the pointer to that of the memory location being accessed. If they are local, we also compare them to the active compartment. If the tags match, the load or store may proceed, else it failstops.

Finally the StoreT, ArgT, and RetT rules enforce the restriction on the escape of local pointers.

4 Proving Correctness

Our correctness proofs relate the abstract machine defined above to the Tagged C semantics, instantiated with the policy above. Specifically we prove a bi-directional simulation between them: for any step that the abstract machine might take in the execution of a program, there is an equivalent step in the Tagged C instantiation, and vice versa. The backwards direction is particularly important because it means that the Tagged C semantics do not permit any behaviors that the abstract semantics forbid.

5 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

$$\tau_V ::= \text{L}(C) \mid \text{S}(a) \mid \mathbb{F}(\text{f}_{fun}) \mid \emptyset \quad a \in nat$$
$$\tau_C ::= C \qquad\qquad\qquad\qquad\quad C \in \mathcal{C}$$
$$\tau_L ::= \text{L}(C) \mid \text{S}(a) \mid \emptyset$$
$$\sigma := n \qquad\qquad\qquad\qquad\quad n \in nat$$

FunT(f_{fun})
$pt' := \mathbb{F}(\text{f}_{fun})$

CallT$(\mathcal{P}, pt)$
  case pt of
  $\mathbb{F}(\text{f}_{fun}) \quad \Rightarrow \quad \mathcal{P}' := comp(\text{f}_{fun})$
  $\_ \qquad\qquad \Rightarrow \quad$ **fail**

LocalT$(\mathcal{P}, \text{x}_{loc}, \text{ty})$
$vt' := \emptyset;$
if $share(\text{x})$
then $pt' := \text{S}(s);$
       $lt' := \text{S}(s);$
       $s := s + 1$
else $pt' := \text{L}(C);$
      $lt' := \text{L}(C);$

MallocT$(\mathcal{P}, pt, vt)$
$vt' := \emptyset;$
if $share\ C$
then $pt' := \text{S}(s);$
       $lt' := \text{S}(s);$
       $s := s + 1$
else $pt' := \text{L}(C);$
      $lt' := \text{L}(C);$

BinopT$(\oplus, \mathcal{P}, vt_1, vt_2)$
  case vt_1, vt_2 of
  $\emptyset, \emptyset \quad \Rightarrow \quad \emptyset$
  $vt, \emptyset$
  $\emptyset, vt \quad \Rightarrow \quad vt' := vt$
  $\_, \_ \quad \Rightarrow \quad$ **fail**

LoadT$(\mathcal{P}, pt, vt, lt)$
  case pt, lt of
  $\text{S}(s), \text{S}(s)$
  $\text{L}(\mathcal{P}), \text{L}(\mathcal{P}) \quad \Rightarrow \quad vt' := vt$
  $\_, \_ \qquad\qquad \Rightarrow \quad$ **fail**

StoreT$(\mathcal{P}, pt, vt, lt)$
  case pt, lt, vt of
  $\text{S}(s), \text{S}(s), \text{L}(C) \quad \Rightarrow \quad$ **fail**
  $\text{S}(s), \text{S}(s), \_$
  $\text{L}(\mathcal{P}), \text{L}(\mathcal{P}), \_ \quad \Rightarrow \quad vt' := vt$
  $\_, \_ \qquad\qquad\qquad \Rightarrow \quad$ **fail**

ArgT$(\mathcal{P}, vt, \text{f}_{fun}, \text{x}_{arg})$
  case $pub(\text{f}_{fun}), vt$ of
  true$, L(C) \quad \Rightarrow \quad$ **fail**
  $\_, \_ \qquad\qquad \Rightarrow \quad \begin{array}{l} vt' := vt; \\ \mathcal{P}' := \mathcal{P}_{CLR} \end{array}$

RetT$(\mathcal{P}_{CLE}, \mathcal{P}_{CLR}, vt, \text{f}_{fun})$
  case $pub(\text{f}_{fun}), vt$ of
  true$, L(C) \quad \Rightarrow \quad$ **fail**
  $\_, \_ \qquad\qquad \Rightarrow \quad \begin{array}{l} vt' := vt; \\ \mathcal{P}' := \mathcal{P}_{CLR} \end{array}$

Figure 7: Policy Definition

$$val ::= \ldots \mid Vptr \ r \ a \qquad r \in 2^{\mathcal{C}}$$

$$e \in ident \rightharpoonup (2^{\mathcal{C}} \times int)$$

$$M \in \mathcal{M} \subseteq 2^{\mathcal{C}} \rightharpoonup mem$$

$$heap\_alloc \in \mathcal{M} \rightarrow 2^{\mathcal{C}} \rightarrow int \rightharpoonup (int \times \mathcal{M})$$

$$heap\_free \in \mathcal{M} \rightarrow 2^{\mathcal{C}} \rightarrow int \rightharpoonup \mathcal{M}$$

$$stk\_alloc \in \mathcal{M} \rightarrow 2^{\mathcal{C}} \rightarrow int \rightharpoonup (int \times \mathcal{M})$$

$$stk\_free \in \mathcal{M} \rightarrow 2^{\mathcal{C}} \rightharpoonup \mathcal{M}$$

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

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

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

Figure 8: Selected Rules for Explicit Sharing

support programs with thousands of shared objects.

That said, it only takes a minor adjustment for our abstract machine 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 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 $\mathtt{malloc}_r$. Then we replace the relevant definitions and step rules with those in Figure 8, 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.

[SNA: I could give a policy for this variant, too, but not sure it's important enough.]

References

[1] ABATE, C., AZEVEDO DE AMORIM, A., BLANCO, R., EVANS, A. N., FACHINI, G., HRITCU, C., LAURENT, T., PIERCE, B. C., STRONATI, M., AND TOLMACH, A. When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2018), CCS '18, ACM, pp. 1351–1368.

[2] AZEVEDO DE AMORIM, A., HRIŢCU, C., AND PIERCE, B. C. The Meaning of Memory Safety. In *Principles of Security and Trust* (Cham, 2018), L. Bauer and R. Küsters, Eds., Springer International Publishing, pp. 79–105.

[3] EL-KORASHY, A., BLANCO, R., THIBAULT, J., DURIER, A., GARG, D., AND HRITCU, C. Secureptrs: Proving secure compilation with data-flow back-translation and turn-taking simulation. In *2022 2022 IEEE 35th Computer Security Foundations Symposium (CSF) (CSF)* (Los Alamitos, CA, USA, aug 2022), IEEE Computer Society, pp. 64–79.

[4] 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).