Abstract

C code can be challenging to secure due to the prevalence of undefined behavior (UB), especially UB related to memory. Using an enforcement system like Tagged C, it is straightforward to enforce a memory safety security policy that eliminates all memory UB, and this is desirable as they can be major vulnerabilities. But some UB may also be supported by common compilers and used by programmers in the wild—the "defacto standard"—and fully protecting all memory objects is expensive. The solution: compartmentalization.

We present a compartmentalization policy that allows some compartments to treat memory concretely, enabling the full array of low-level pointer-manipulating idioms internally while isolating any errors that arise from such idioms inside the compartment. We prove that under this policy, standard-compliant compartments are protected from other compartments that they link with. Conversely, we conjecture that even compartments containing memory UB are protect, albeit in a more limited sense, and we give a formalization of the nature of this protection.

1 Introduction

1.1 Why Memory Safety?

1.2 Why Not Memory Safety?

Low-level Idioms The first reason not to enforce full memory safety is that code found in the wild may be memory unsafe in ways that are harmless or even intentional. These "low-level idioms" exploit the underlying structure of pointers as integers. In particular, we are interested in those idioms that create a new pointer to an existing object, either from scratch (pointer-forging) or by performing arithmetic on a pointer to a different object. Errors involving these idioms can very easily result in loads and stores to and from the wrong object, creating serious vulnerabilities. And these idioms are used in practice: for example, the Linux kernel's per-CPU-variable library maintains a table of offsets that are added to a base pointer to find one of several separately-allocated objects in memory.

```
#define per_cpu_ptr(ptr, cpu) ({ RELOC_HIDE(ptr, __per_cpu_offset[cpu]); })
// RELOC_HIDE computes ptr + offset while hiding from the compiler
```

It is also common for low-level code to directly access pre-defined addresses that play some specialized role, such as interfacing with an external piece of hardware. [TODO: example]

In each of these scenarios, a full memory-safety policy will fail on a normal (safe) execution! So, we need a policy that can be permissive enough to allow low-level idioms, but which mitigates their risks by constraining the relevant code to isolated, untrusted compartments.

Performance The second reason not to enforce full memory safety is that doing so is likely to be expensive. The details vary depending on the hardware, but in practice Tagged C policies will likely be limited to a finite number of "live" tags at any given time. In memory-safety policies, this will be correlated to the number of live objects at any given time, but by relaxing the policies, we can reduce it. Our smallest policy uses a number of tags that is linear in the number of live objects that are shared across compartment boundaries.

```
A >> int a[10];

A >> int b[10];
```

```
A >> void f() {
       for (int j=0; j<=10; ++j) {
         a[j] = 0;
       }
     }
```

Figure 1: Memory Safety Violation

1.3 Tagged C and Concrete C

Here we give a brief overview of Tagged C, discuss its baseline protections, and confirm that in the absence of other policies it can support all of the low-level idioms discussed above. This is a good place to plug Concrete C as an independent concept that is useful.

1.4 Contributions

Our contributions are:

- A compartmentalization policy that separates a program into mutually distrustful components, which may be "strict" (memory safety is enforced internally) or "lax" (permitted to treat memory fully concretely).
- A pen-and-paper proof that the compartmentalized system preserves the security of a safe component from an unsafe one, formalized as a robust safety preservation property.
- A novel security property, [TODO: give it a name] characterizing the protection offered to an unsafe component, formalized in terms of a simplified version of Tagged C. We conjecture that our compartmentalization policy enforces [NAME] as well.

2 Good and Bad Behavior, by Example

In this section we will motivate our compartmentalization policies with small examples illustrating requirements on our system's behavior. In all examples we will establish that `f` is in the compartment `A` (designated `A >> ...`), and `g` is in compartment `B`. Table 1 summarizes whether each policy under consideration fulfills the requirements.

First, consider compartment `A` in Figure 1. Regardless of any compartment it gets linked against, `f` contains an off-by-one error that overflows `a`. In a standard memory layout, `b` is contiguous to `a`, so the overflow overwrites `b[0]`. The baseline policy allows this, but this is undesirable behavior, and we require that it not occur (**strict-UB** in 1). We can enforce this by applying a memory safety policy, but we then run into other issues.

Suppose that we fix `f`, and link `A` against compartment `B`, which contains the function `g` (Figure 2). `g` performs inter-object pointer arithmetic that violates memory safety, even though this usage is safe, so we will see a we see a failstop in `g` under the memory safety policy. That's a problem. Since we can't guarantee that no errors exist in general, we would like to find a middle ground that enables the low-level pointer manipulations of `B` while still preventing any memory UB in `A` (**lax-internal-UB** in 1).

```
A >> int a[10];

A >> int b[10];

A >> void f() {
       for (int j=0; j<10; ++j) {
         a[j] = 0;
       }
     }
```

```
B >> int g() {
       int* x = malloc(sizeof(int)*4);
       int* y = malloc(sizeof(int)*4);
       int off = y - x;

       f();
       x[off] = 42;
       return off;
     }
```

Figure 2: Safe Despite UB

```
A >> int f() {
       int* x = malloc(sizeof(int)*4);
       g();
       return x[0];
     }
```

```
B >> int g() {
       int* y = malloc(sizeof(int)*4);
       p[sizeof(int)*(-4)] = 5;
     }
```

Figure 3: External Pointer Arithmetic

One simple solution to this problem is to separate our compartments into "strict" and "lax" enforcement categories. `A` should get strict enforcement, while `B` needs lax enforcement to run. Now the program in Figure 1 will failstop in `A`, while the one in Figure 2 will execute successfully.

But we need to be careful with lax enforcement! Consider the program in Figure 3: if `x` and `y` are allocated contiguously, then `g` will write to `x` despite never having a valid pointer—and `A` is not expecting its memory to be accessed in this way. We term this an "external" UB, because it spans both compartments: `B` is doing the UB, but it's impacting `A`'s memory. We want these to be illegal (**lax-external-UB**).

The most straightforward way of implementing the division between lax and strict compartments is to decide which set of rules to use at any given time based on the status of the current compartment. This in turn means that cannot permit compartments to access one anothers' pointers. We call this policy NOSHARE.

To allow compartments to share memory with one another (**sharing**), we define a more sophisticated policy, SHARE. In SHARE, rather than separate whole compartments into lax and strict, we separate individual objects into "shared" and "local". Shared objects follow the full memory safety rules, as in a strict compartment, while local objects are guaranteed not to escape their compartments and can be accessed via pointer arithmetic by the compartment that owns them. A compartment that allocates all of its objects locally is equivalent to a lax compartment, and one that allocates them all as shared is equivalent to a strict compartment.

3 Separating Strict and Lax Compartments

As a first step toward our ultimate goal, let's think about a simpler kind of protection: separating the world into strict and lax compartments that do not share memory.

Policy	**strict-UB**	**lax-internal-UB**	**lax-external-UB**	**sharing**
$\perp$ (baseline)	N	Y	N	Y
MS	Y	N	Y	Y
NOSHARE	Y	Y	Y	N
SHARE	Y	Y	Y	Y

Table 1: Policies and their requirements

3.1 Policy In Detail

A Tagged C policy consists of a tag type τ, a *policy state* type σ, and instantiations of 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.

Relevant Policy Rules Without further ado, let's define our policy. Tags consist of compartment identifiers $\mathbf{comp}(C)$ and per-compartment allocation colors $\mathbf{clr}(C, a)$, where a is a natural number. The policy state is a counter that tracks the next allocation color.

$$\tau ::= \mathbf{comp}(C) | \mathbf{clr}(C, a) | \emptyset$$

$$\sigma := \mathbb{N}$$

We assume a mapping *owner* from function and global identifiers to compartments, and initialize tags such that the function pointer tag for each function f is $owner(f)$. The trusted compartment set *strict* C contains all of the compartments that we wish to enforce memory safety within. We start by keeping track of which compartment we're in.

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

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

The memory locations of global variables are tagged with the compartment that owns them. Dynamic memory is more complicated. We first check whether the active compartment is strict. 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 *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 *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$(\mathtt{x}_{glb}, \mathtt{ty}_{typ})$
$vt' := \emptyset$;
$lt' := owner(\mathtt{x})$

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.

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

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

Note that this policy allows pointers to be passed between compartments, but they can only be accessed by the compartment that allocated them. This is a simplification to introduce the structure of the safety properties, and we will relax it later.

When we deallocate any object, we clear its location tags, so old floating pointers can no longer read or write to it.

3.2 Proving Protection

Events and Traces Compartments interact via calls and returns, and via visible loads and stores. A load or a store is visible if it is made via a valid pointer. To each allocation we associate a provenance symbol ϕ, and we track the provenance of every pointer within each compartment. Then, when a load or store is performed by a pointer of provenance ϕ, we record both ϕ and the concrete address of the target object in the trace.

Formally, an event value is a value with an optional provenance. An event is a call, return, alloc, free, load, or store. An alloc records the range of addresses that are allocated and gives them a unique provenance. A load or a store always records the provenance of the pointer being accessed (which means that it needs to be a valid pointer so as to have a provenance); a load or store with no provenance is not visible in the trace. It also gives the range of addresses affected.

```
A >> int f() {                          B >> int g(int* p, int i) {
2       int* x = malloc(sizeof(int)*4);  12      p[i] = 5;
3       int* y = malloc(sizeof(int)*4);  13    }
4       int off = y - x;
5       x[0] = 0;                       B >> int main() {
6       x[off] = 42;                     16      return f();
7       g(x,0);                          17    }
8       g(x,off);
9
10      return y[0];
11    }
```

$$
\begin{aligned}
A[B] \rightsquigarrow & \mathit{call}\ f\ [] \cdot \mathit{alloc}\ \phi_0\ \texttt{0xAB00} \ldots \texttt{0xAB0F} \cdot \\
& \mathit{alloc}\ \phi_1\ \texttt{0xAB20} \ldots \texttt{0xAB2F} \cdot \mathit{store}\ \phi_0\ \texttt{0xAB20} \ldots \texttt{0xAB23}\ 42 \mid \textsc{ms} \\
& \mathit{call}\ g\ [(\phi_0, \texttt{0xAB00}); 0] \cdot \mathit{load}\ \phi_0\ \texttt{0xAB00} \ldots \texttt{0xAB03}\ 0 \mid \textsc{noshare} \\
& \mathit{call}\ g\ [(\phi_0, \texttt{0xAB00}); 32] \cdot \mathit{load}\ \phi_0\ \texttt{0xAB20} \ldots \texttt{0xAB23}\ 42 \mid \textsc{share} \\
& \mathit{return} \cdot \mathit{return}\ 42
\end{aligned}
$$

Figure 4: Example With Cross-compartment Sharing and Pointer Arithmetic

$$
\begin{aligned}
ev ::= & \ v \mid (\phi, v) \\
e ::= & \ \mathit{call}\ f\ \overline{ev} \\
\mid & \ \mathit{return}\ ev \\
\mid & \ \mathit{alloc}\ \phi\ a_0 \ldots a_n \\
\mid & \ \mathit{free}\ \phi \\
\mid & \ \mathit{load}\ \phi\ a_0 \ldots a_n\ ev \\
\mid & \ \mathit{store}\ \phi\ a_0 \ldots a_n\ ev
\end{aligned}
$$

A trace is a (possibly infinite) sequence of event values.

Let A and B be components such that linking them produces a complete program. We write the linked program $A[B]$, and when such a program run under tag policy ρ produces a trace t, we write $A[B] \rightsquigarrow_\rho t$. We represent the "baseline" policy, which never failstops, with $\bot$.

Trace Example Let's look at an example of a program and the trace it produces. In Figure 4, we see multiple kinds of undefined behavior, as `f` has internal `UB` and `g` has external. We give a full execution trace of the program (one of many, selecting arbitrary addresses for allocations), marking where it will be truncated by a failstop under each policy in red.

As we truncate the trace, we eliminate interactions between compartments. [TODO: say something intelligent here about what kinds of interactions.] Quantifying over all programs, some of our policies eliminate whole classes of interactions between compartments. Full memory safety is the most restrictive, and as it corresponds to the C standard, it forms the basis of our overall security property: for a particular interesting class of inter-compartment interactions, our compartmentalization policies should not be any more permissive than MS.

Robust Safety Preservation This brings us to the definition of *robust safety preservation*. First, we define the class of *safety properties* as those properties π which can always be falsified by a finite prefix of a trace. Now, for any compartment "under focus", C, consider the set of safety properties *robustly satisfied* by C under some policy ρ:

$$\overline{\pi}(C)_\rho \triangleq \{\pi \mid \forall A\ t.C[A] \rightsquigarrow_\rho t \rightarrow t \in \pi\}$$

For any pair of policies ρ and ρ', we say that ρ' enjoys robust safety preservation with respect to ρ if, for all C, $\overline{\pi}(C)_\rho \subseteq \overline{\pi}(C)_{\rho'}$.

3.3 Proof

4 Safely Sharing Memory

Now we extend the policy above to allow sharing memory. The crucial difference is that safe pointers are no longer tied to particular compartments; instead, we distinguish between *allocation points*—that is, calls to malloc, grouped according to whether or not the allocated object should be shared between compartments. So, we add another tag, **share**, which is attached to the function pointer of each call to malloc that is meant to be shared.

$$\tau ::= \mathbf{comp}(C) | \mathbf{clr}(a) | \emptyset | \mathbf{share}$$

For simplicity, we focus exclusively on malloc and disallow sharing of stack pointers; these are therefore tagged with **comp** for every compartment. The malloc rule checks the tag on the function pointer being called to determine how to proceed.

LocalT$(\mathcal{P}, \mathtt{ty}_{typ})$
let $\mathbf{comp}(C) := \mathcal{P}$ in
$vt' := \emptyset$;
$pt' := \mathbf{comp}(C)$;
$lt' := \mathbf{comp}(C)$;

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

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

Safe pointer tags are propagated similarly to before, as are loads and stores.

LoadT$(\mathcal{P}, pt, vt, lt)$
if $lt = \mathbf{comp}(\_)$
then **assert** $\mathcal{P} = lt$
$vt' := vt$
else **assert** $pt = lt$
$vt' := vt$

StoreT$(\mathcal{P}, pt, vt, lt)$
if $lt = \mathbf{comp}(\_)$
then **assert** $\mathcal{P} = lt$
$\mathcal{P}' := \mathcal{P}$; $vt' := vt$; $lt' := lt$
else **assert** $pt = lt$
$\mathcal{P}' := \mathcal{P}$; $vt' := vt$; $lt' := lt$

4.1 Proving Safety with Sharing

We need to axiomatize the fact that only the lax compartment is allowed to do memory unsafe things.

5 Trust While Using Unsafe Idioms

While we have proven that our compartmentalization policy protects our strict compartment from lax ones, we conjecture that it can also protect a lax compartment from an attacker that aims to exploit its non-standard behavior. But what does it mean for a compartment to be protected when it contains UB? To demonstrate the difficulty, consider the following example:

```
C >> f() {
  int x[10];
  x[10] = 42;
  return 5;
}
```

Since `f` writes out of bounds, its behavior is undefined, and under a full memory safety policy it will always failstop—which in turn means that it will vacuously satisfy all safety properties robustly. Under our compartmentalization policy, the write to `x[10]` will either be successful (but unstable) or it will failstop. If it doesn't failstop, `f` returns 5, which means that there are a large number of safety properties no satisfied. Clearly, we cannot expect our policy to preserve arbitrary safety properties that are satisfied in a memory safe setting.

Instead, we define a new partial memory model in which each compartment has its own private (concrete) address space. The model is axiomatized such that, in each private address space, in-bounds loads and stores behave as expected, while out-of-bounds accesses are unstable and may failstop. This memory model forms the specification from which we derive a robust safety preservation property.