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.

```
int f() {
  int *x = malloc(sizeof(int)*8);
  int *y = malloc(sizeof(int)*8);

  for (int i = 0; i<8; ++i) {
    x[i] = 0;
    y[i] = 0;
  }

  g();
  return x[0]; // always returns 0
}
```

Figure 1

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 Compartments by Example

In Figure 1, the function `f` is assumed to always return 0, because `x[0]` is assigned 0 and not updated before its value is returned. This assumption is dependent on the system being memory safe—in the absence of memory safety, `g` might overwrite `x[0]`. The fact that `g` will not overwrite `x` follows from a set of principles that we might term "capability reasoning"—working backwards from the fact that every permitted load or store to an object must be made through pointer that traces its provenance to that object's initial allocation, we can rule out memory accesses by proving that no pointer of the relevant provenance is accessible to a given piece of code.

But, there are several reasons why we might not want to enforce full memory safety. For one thing, it's expensive. For another, the definition of `g` might contains some undefined memory behavior that doesn't directly impact `x`.

```
int f() {
  int *x = malloc(sizeof(int)*8);
  int *y = malloc(sizeof(int)*8);
  int *z = malloc(sizeof(int)*8);

  for (int i = 0; i<8; ++i) {
    x[i] = 0;
    y[i] = 0;
    z[i] = i;
  }

  g(z);
  return x[0]; // always returns 0
}
```

Figure 2

Instead of memory safety, we can apply a compartmentalization policy that separates `g` from `f`, so that they cannot both access the same memory locations. This essentially merges `x` and `y`, from a protection perspective: they can both be accessed by `f`, even if `f` should contain some undefined behavior, and only by `f`. Therefore, as before, `x[0]` will not change during the call. In performance terms, we halve the number of tags necessary to protect the two arrays.

But, now consider the version of `f` in Figure 2. It's now passing a pointer, `z`, to `g`. Our strict separation no longer works! This program will still run, and `x` will still be protected, under a full memory safety policy. But under our compartmentalization policy, if `g` attempts to read or write `z`, execution will failstop.

We need a hybrid model that allows some objects to be shared, while still allowing us to merge `x` and `y` for efficiency. Escape analysis can determine which objects need to be shared, by allocation site; we assume that the code is rewritten so that calls to `malloc` that should be shared are replaced by a special `malloc_share` call, which behaves identically except in terms of how the policy treats it.

Now, the question that we have to answer is: how fine-grained to we need our protection of the shared objects to be? We have a few different options.

- All shared objects are grouped together
- All objects are grouped according to which compartment(s) they are shared with
- Objects are grouped by allocation-site
- Each object is kept separate, with no grouping at all

Our guiding principle here is that we want to preserve all of the properties of a memory safe system, so keeping each object separate will definitely work. It would be nice to be able to compress them further, but it turns out that each kind of grouping will cost us something.

First, in Figure 3a, if `g` and `h` are in separate compartments that do not call one another, we should be able to rely on `x[0]` staying constant during the call to `h`. But if we group all shared objects together, then sharing `y` with compartment `B` implicitly means sharing `x` as well.

```
A >> int f() {
  int* x = malloc_share(sizeof(int));
  int* y = malloc_share(sizeof(int));
  g(x);
  x[0] = 0;
  h(y);
  return x[0];
}

B >> void g(int*);

C >> void h(int*);
```

(a)

```
A >> int f() {
  int* x;
  for (int i = 0; i < 100; i++) {
    x = malloc_share(sizeof(int));
    if (i % 2) {
      g(x);
    } else {
      h(x);
    }
  }
  return x[0];
}
```

(b)

Figure 3

We might consider grouping together objects based on which compartment(s) they're going to be shared with. One could imagine approximating the set of compartments that the objects from a given allocation site might escape to, and grouping together all objects that escape to the exactly the same set of compartments. This is unworkable; it is simple to construct an example like Figure 3b, where objects' sharing behavior is finer-grained than their allocation-site. There are also temporal factors to consider: if a pointer is shared to `B` early in its lifetime, and `C` much later, can we rely on its safety from `C` in the interim? This last concern seems to be the final nail in the coffin of any meaningful groupings. So, we must tag every shared object uniquely unless we're willing to give up capability-reasoning for all shared objects.

If we are willing to give up capability reasoning for all shared objects, then it isn't clear that we gain anything from grouping them more finely. This might not be so bad; in our original motivating example, we were concerned with `x` being protected, and as `x` does not escape at all, its allocation site will not be marked shared. This is a narrower form of reasoning, one that we might call "escape-local" reasoning.

3 Compartmentalization Without Sharing

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.

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. [SNA: This is going to be a real pain to explain so that it can be followed without reading the last paper.]

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.

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

```
$\mathbf{MallocT}(\mathcal{P}, pt, vt)$
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  $pt' := \mathbf{comp}(C)$;
      $lt' := \mathbf{comp}(C)$;
```

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

```
$\mathbf{BinopT}(\oplus, \mathcal{P}, vt_1, vt_2)$
  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.

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

```
$\mathbf{StoreT}(\mathcal{P}, pt, vt, lt)$
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

Concrete C Semantics A program's behavior is determined by its Tagged C semantics, which is in turn derived from the Concrete C semantics. Let's first ignore tags and describe how the Concrete C semantics behaves with memory.

Concrete C promises to fulfill a set of memory axioms, but the means by which it does so is nondeterministic. So, for example, a call to malloc may failstop (indicating that the system it out of memory) or it may return a pointer, with the guarantee that the block beginning at that pointer does not overlap with any allocated block. Subject to that guarantee, the block might be anywhere in the address space. This is one source of nondeterminism.

The behavior of loads and stores differs depending on whether they are accessing allocated memory. A load or store that accesses allocated memory guarantees the usual set of properties of memory: a load produces the last stored value, etc. A load or store outside of allocated memory has fewer guarantees. It may failstop, for instance because the accessed address is reserved for the compiler, or it may proceed, in which case the normal properties of memory apply only up to a call or return, when the compiler is free to scramble unallocated memory however it sees fit.

Isolated Semantics Our specification for the no-sharing case is derived from a machine in which memory is isolated by construction. Where previously we had a single flat memory, we now have a map from compartments to flat memories. The state tracks at any given time which compartment is active.

$$M \in \mathcal{M} : comp \rightarrow mem$$

Regular state now extended with a compartment id and a memory map:

$$\mathcal{S}(C, M, le, te \mid s \gg k@\mathcal{P})$$

We adjust all of the step relation rules appropriately. [SNA: It's unclear to me whether this semantics should have tags. Probably not, in which case it really does look a lot like Concrete C.]

We add one additional axiom, the *compatible_alloc* axiom, which states that at no point during execution is an address allocated in more than one compartment. [SNA: Without this, we will have traces that can appear in the source but not the target. That's a problem if some interesting behavior only occurs in those traces (seems unlikely, but maybe possible when we've used a lot of memory.)]

Events and Traces Compartments interact via calls and returns, and via loads and stores.

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.

$$ev ::= v|(\phi, v)$$

$$\begin{aligned} e ::= & C \gg \textit{call } f\ \overline{ev} \\ & |C \gg \textit{return } ev \\ & |C \gg \textit{alloc } a_0 \dots a_n \\ & |C \gg \textit{free } a_0 \dots a_n \\ & |C \gg \textit{load } a_0 \dots a_n\ ev \\ & |C \gg \textit{store } a_0 \dots a_n\ ev \end{aligned}$$

A trace is a (possibly infinite) sequence of events.

Let A and B be components such that linking them produces a complete program. We write the linked program $A[B]$. Now we need to capture its traces. Execution is deterministic given a certain compiler/allocation strategy. We range over these with the metavariable α, and write that a particular combination of program $A[B]$, α, and tag policy ρ produces a trace t as $\alpha \vdash A[B] \rightsquigarrow_{tag(\rho)} t$. To write the same given the multi-concrete model, we write $\alpha \vdash A[B] \rightsquigarrow_{mc} t$. Producing a trace is defined inductively in terms of the step function such that if $\alpha \vdash A[B] \rightsquigarrow_X t$, then $\alpha \vdash A[B] \rightsquigarrow_X s$ for any s that prefixes t.

We frequently want to talk about all possible traces of a program under a given policy or the multi-concrete model regardless of the memory layout. In this case we define the trace set $[\![A[B]]\!]_X$ where X is mc or $tag(\rho)$.

$$[\![A[B]]\!]_X \triangleq \bigcup_{\alpha} \{t \mid \alpha \vdash A[B] \rightsquigarrow_X t\}$$

Properties and the Big Property A property π is a set of traces, members of which are said to satisfy π.

A property π is *robustly satisfied (def 1)* by a compartment A under context X if, for all B, $[\![A[B]]\!]_X \subseteq \pi$.

A policy ρ enjoys *robust property satisfaction (def 1)* with respect to ms if, for all A and all π robustly satisfied by A under ms, A robustly satisfies π under $tag(\rho)$.

By the contrapositive, this is equivalent to: for all A, B, and t such that $t \in [\![A[B]]\!]_{tag(\rho)}$,

Trace Examples Let's look at an example of a program and the trace it produces. In Figure 5, we see multiple kinds of undefined behavior, as `f` has internal `UB` and `g` has external. We give

```
A >> f() {
  int* x = malloc(sizeof(int));

  return (int) x;
}
```

$$[\![A[B]]\!] = \{\ldots\ \mathit{call}\ f\ []\cdot \mathit{return}\ 8 * i \ldots \mid 0 \leq i \leq 2^{61}\}$$

```
A >> f() {
  int* x = malloc(sizeof(int));
  int* y = malloc(sizeof(int));

  return (int) x + (int) y);
}
```

$$[\![A[B]]\!] = \{\ldots\ \mathit{call}\ f\ []\cdot \mathit{return}\ 8 * (i + j) \ldots \mid 0 \leq i \leq 2^{61}, 0 \leq i \leq j, i \neq j\}$$

```
A >> f() {
  int* x = malloc(sizeof(int));
  int* y = malloc(sizeof(int));

  if ((int) x < (int) y && (int) x > (int) y) {
    return 1;
  } else {
    return 0;
  }
}
```

$$[\![A[B]]\!] = \{\ldots\ \mathit{call}\ f\ []\cdot \mathit{return}\ 0 \ldots\}$$

Figure 4: Addresses not in Traces

```
A >> int f() {
       int* x = malloc(sizeof(int)*4);
       int* y = malloc(sizeof(int)*4);
       int off = y - x;
       x[0] = 0;
       x[off] = 42;
       g(x,0);
       g(x,off);

       return y[0];
     }
```

```
B >> int g(int* p, int i) {
       p[i] = 5;
     }

B >> int main() {
       return f();
     }
```

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

Figure 5: Example With Cross-compartment Sharing and Pointer Arithmetic

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.

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

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

```
$\mathbf{BinopT}(\oplus, \mathcal{P}, vt_1, vt_2)$
  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.

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

```
$\mathbf{StoreT}(\mathcal{P}, pt, vt, lt)$
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.

[SNA: After some consideration, I've come to the conclusion that this is the approach most likely to work. One con: this means I have to actually define that memory model (but I envision it being basically n Concrete C memories stacked on top of one another).]