```
#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.

1.2 Related Work

There are three main works that we need to compare in detail: "When Good Components Go Bad," from Abate et al., "SecurePtrs," from El-Korashy et al., and "SECOMP," from Thibault et al. These works all present mechanized proofs of some compartmentalization scheme in C, while we only have a pen-and-paper proof. We consider them in the following other dimensions:

1. How much of the C language do they cover?

2. Do they support shared memory? Are there any restrictions?

3. How strong is their specification? How does it handle non-standard C (i.e., UB)? Is it different in trusted vs. untrusted compartments? Ideally it should be strong enough to rule out bad cross-compartment behavior but not so strong as to fail when behavior is only internal.

4. What hardware assumptions do we need to make for their implementations to be supported by tags?

Good Components

1. Their language is the epitome of a toy, it isn't even really C-like.

2. There is no shared memory, only a small number of shared registers.

3. They model UB in the traditional C sense of "anything might happen", by replacing components that encounter UB with hypothetical arbitrary components. Their aim is to keep the UB restricted to the compartment in which it occurred. Therefore, their model is not useful for reasoning about the specific behavior of compartments that contain UB but that may nonetheless have consistent behavior. Their trust model is dynamic, and therefore very precise: trusted compartments become untrusted if they reach UB. This makes their specification too strong for us.

4. They propose a tag policy based on linear tags, which are unlikely to be technically feasible (in their words, "folklore.") However, without too much trouble, they could be implemented using techniques from Roessler and DeHon. Their model does not depend on protecting memory internally, and they don't support sharing, so they could implement it on tagged hardware with a relatively small number of tags, that number being linear in the number of compartments.

Δ : our language is full C, our policy supports sharing, and our property supports internal UB with more specificity.

SecurePtrs

1. They cover more of C than above, but they still are fairly restricted.
2. They fully support shared memory.
3. Their specification is RSP from a fully-defined source language with safe pointers, which is very strong. They mention in passing that they could deal with UB along the lines of the paper above, but because all pointers are safe by construction, would not include memory UB. This is even more restrictive than above. As a result of the relatively narrow gap between their machines, they don't really need to make a distinction between trusted and untrusted compartments. Everyone is equally well-behaved.
4. Their target machine is fully memory safe, so we must assume that implementing it in tagged hardware would require a number of tags linear in the number of simultaneously live allocations, a fairly high bar.

Δ : our language is full C, our policy does not require as many tags as the likely implementation of theirs, and our property property supports internal UB.

SECOMP

1. They cover all of CompCert C.
2. Shared memory is not supported.
3. *How strong is their specification?* They handle UB similarly to "Good Components", while adding new UB in the form of violations of the compartment interfaces. But, because they are focused on secure compilation rather than compiler correctness, the question of UB is primarily focused on their "blame" theorem, which states that any UB encountered by a source compartment must have originated in that compartment. As with "Good Components," we consider this too strong, as we would like to preserve a semantics that allows us to reason about what happens in the presence of UB.
4. This appears to be the same as "Good Components": it can be done with lightweight hardware, because it doesn't require individual object tags.

Δ : our policy supports sharing, and our property supports internal UB with more specificity.

Distinguishing Our Work Compared to these works in total, ours supports cross-compartment sharing with fewer constraints on available tags. Our property is tighter: the source model that it is based on gives definition to some UB, and therefore we can describe more precisely what happens in the presence of that UB. We also support cross-compartment sharing in a full C setting, not merely a toy.

2 Abstract Sharing Semantics

In the example above, we discussed the need to reason about the behavior of a compartmentalized system in terms of whether memory is being shared or is local to a compartment. We define a C semantics that builds this kind of reasoning into the memory model. This section will assume familiarity with the Tagged C semantics, and will focus on how this abstract semantics differs. Instead of a single flat memory, we separate the world into compartments, ranged over by A, B, C, etc., each with its own flat memory.

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.

Additionally, there is a shared memory that follows the rules of a standard block-offset model. Any compartment may allocate objects in its own local memory with a call to `malloc`, or in the shared memory with a call to a special `malloc_share`.

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.

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

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.]

Load and store semantics differ based on which kind of pointer is being accessed. Shared pointers access their usual memory at the block and offset that they carry, while local pointers access the memory of the compartment they carry (which will also always be the active compartment, because they can't escape.)

[TODO: sample rules]

Allocation Axioms [TODO: list out the normal Concrete C axioms here]

Allocator Oracles and Realizability [SNA: Trying something a little different: we define the flat memories as separate, but then specify that we're going to restrict ourselves to allocators that we can realize in a single memory. Then we use those allocators to index the relation between high and low states.]

In these abstract semantics, we quantify over all allocators that satisfy the above axioms. Many such allocators are not actually realizeable in a given concrete system: most obviously, because they allow more memory to be allocated than the system has. A related obstacle is that we cannot realize in a single address space an allocator that would assign the same internal address. To simplify our model, we will restrict all of our proofs to deal with a specific subset of allocators that correspond to realistic allocation systems and therefore are guaranteed to be realizable.

Formally, we assume that such an allocator instantiates a “live list” that associates every allocation—local allocations in every compartment and shared allocations alike—to a range of addresses that are guaranteed not to overlap.

Abstract Events and Traces One advantage of defining a property in terms of this abstract machine is that we have a clear distinction between loads and stores that represent compartment interactions (and should appear in a trace) and those that are internal to a compartment and don't matter.

We define the set of trace events as calls, returns, and cross-compartment loads and stores, as well as the terminal *stuck* event representing undefined behavior.

$$\begin{aligned} e \in EvA ::= & \, \mathit{call}\ f\ \overline{v} \\ & \mathit{return}\ v \\ & \mathit{store}\ b\ \mathit{off}\ v \\ & \mathit{load}\ b\ \mathit{off}\ v \\ & \mathit{stuck} \end{aligned}$$

A trace is a (possibly infinite) sequence of events. We write that a particular combination of program $A[B]$ and oracle α, produces a trace t as $\alpha \vdash A[B] \rightsquigarrow t$.

We describe the behavior of a given program via its trace set $[\![A[B]]\!]$:

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

3 Implementing Compartmentalization in Tagged C

[This is where I would put the description of the policy, but right now I'm not focused on that.]

4 Proving Correctness

Our correctness proofs relate the abstract machine defined above to the Tagged C semantics, instantiated with the policy above. We give Tagged C a trace semantics as well, with a separate type of traces reflecting the more concrete setting. We will need to show that we can reproduce any property from the abstract traces semantics in the concrete one; the remainder of this writeup is dedicated to defining what it means to reproduce a property in this compartmentalized setting, and describing how we prove it.

Tagged C Events and Traces Tagged C does not distinguish between cross-compartment loads and stores and local ones. So, its trace model must be somewhat different. Specifically, loads and stores now access concrete addresses, and load and store events are generated on every load and store, not just the cross-compartment ones.

We define the set of trace events as calls, returns, loads, stores, stuckness/UB, and now an explicit *fail* event representing failstop behavior.

$$\begin{aligned} \hat{e} \in EvC ::=\ & call\ f\ \overline{v} \\ & return\ v \\ & store\ addr\ v \\ & load\ addr\ v \\ & stuck \\ & fail \end{aligned}$$

When a Tagged C program under policy ρ produces a trace $\hat{t}$, we write it $\alpha \vdash A[B] \rightsquigarrow \hat{t}$, and we define $\llbracket A[B] \rrbracket_\rho$ similarly to above.

The intuition behind our notion of safe compartmentalization is that for any compartment A and any property that it robustly satisfies in the abstract machine, it should also robustly satisfy that property in Tagged C with our policy. Additionally, we should sanity-check our semantics by proving forward simulation. In fact, we will prove a bisimulation, from which both forward simulation and robust property preservation follow naturally. However, the differences in the events that comprise traces mean that they can't actually be the same traces; we instead need to describe the relationship between them that we consider to capture an important notion of sameness.

Relating traces The notion of sameness that we choose is: all loads and stores to shared blocks in the abstract trace have counterparts in the concrete trace, and none of them overlap with any additional loads or stores that appear in the concrete trace.

We define a relation $\sim$ indexed by an assignment of blocks to their base addresses and bounds (the address immediately following their last byte), $\Gamma \in block \rightharpoonup (int \times int)$, which we write $\Gamma \vdash e \sim \hat{e}$. Defined inductively on values:

$$\frac{\Gamma\ b = (base, bound)}{\Gamma \vdash sptr\ b\ off \sim long\ (base + off)} \qquad \frac{}{\Gamma \vdash lptr\ C\ addr \sim long\ addr} \qquad \frac{}{\Gamma \vdash v \sim v}$$

Which extends naturally to events:

$$\frac{\Gamma \vdash v_1 \sim \hat{v}_1 \qquad \Gamma \vdash v_2 \sim \hat{v}_2}{\Gamma \vdash load\ b\ off\ v_2 \sim load\ addr\ \hat{v}_2} \qquad \frac{\Gamma \vdash (sptr\ b\ off) \sim \hat{v}_1 \quad \Gamma \vdash v_2 \sim \hat{v}_2}{\Gamma \vdash store\ b\ off\ v_2 \sim store\ addr\ \hat{v}_2}$$

$$\frac{\Gamma \vdash \overline{v} \sim \hat{\overline{v}}}{\Gamma \vdash call\ f\ \overline{v} \sim call\ f\ \hat{\overline{v}}} \qquad \frac{\Gamma \vdash v \sim \hat{v}}{\Gamma \vdash return\ v \sim return\ \hat{v}}$$

And finally, to traces, via the possible addition of (non-shared) loads and stores. These must not overlap with any blocks that are mapped in Γ. Note that these are coinductive definitions. Also, the *fail* event in Tagged C relates to any abstract trace.

$$\frac{\forall\ b.\Gamma\ b = (base, bound) \rightarrow \neg(base \leq addr < bound) \quad \Gamma \vdash t \sim \hat{t}}{\Gamma \vdash t \sim load\ addr\ \hat{v} \cdot \hat{t}}$$

$$\frac{\forall\ b.\Gamma\ b = (base, bound) \rightarrow \neg(base \leq addr < bound) \quad \Gamma \vdash t \sim \hat{t}}{\Gamma \vdash t \sim store\ addr\ \hat{v} \cdot \hat{t}}$$

$$\frac{\Gamma \vdash v \sim \hat{v} \quad \Gamma \vdash t \sim \hat{t}}{\Gamma \vdash v \cdot t \sim \hat{v} \cdot \hat{t}}$$

$$\frac{}{\Gamma \vdash t \sim \mathit{fail}}$$

Which finally brings us to defining the overall trace relation $\approx$ by quantifying over Γ.

$$t \approx \hat{t} \triangleq \exists \Gamma . \Gamma \vdash t \sim \hat{t}$$

In short, $\hat{t}$ is a plausible result of starting with t, assigning concrete addresses to blocks, and recording additional internal memory accesses, possibly failing prematurely. Conversely, t is the result of stripping internal loads and stores out of $\hat{t}$ and mapping shared ones to their block-offset addresses.

Trace-relating Correctness In order for the abstract machine to be at all useful in reasoning about the behavior of Tagged C, we need to show that Tagged C (with a compartmentalization policy) actually implements it! This is akin to *forward correctness*, albeit with an identity compiler and a target machine that might failstop. Formally, Tagged C with policy ρ enjoys forward correctness if, for all programs P and realizable allocator α, if $\alpha \vdash P \rightsquigarrow t$, then there exists some $\hat{t}$ such that $t \approx \hat{t}$ and $\alpha \vdash P \rightsquigarrow_\rho \hat{t}$.

The possibility of failstop makes this form of correctness quite weak. It might be desirable to prove that it only failstops under certain conditions, particularly that a fully memory-safe program never failstops. [SNA: I'm considering whether I can just match stuck events in the source with failstops in the target, so we only failstop on abstract UB.]

Trace-relating Property Preservation Abstract and concrete properties are related if, for every trace in one, the other produces a related trace.

$$\begin{aligned} \pi \approx \hat{\pi} \triangleq & (\forall t.\ t \in \pi \Rightarrow \exists \hat{t}.\ \hat{t} \in \hat{\pi} \wedge t \approx \hat{t}) \wedge \\ & (\forall \hat{t}.\ \hat{t} \in \hat{\pi} \Rightarrow \exists t.\ t \in \pi \wedge t \approx \hat{t}) \end{aligned}$$

A policy ρ enjoys *trace-relating robust property preservation* with respect to the abstract machine if, for all A and all π robustly satisfied by A, the concrete machine with policy ρ robustly satisfies any $\hat{\pi}$ when $\pi \approx \hat{\pi}$.

$$\begin{aligned} \mathit{RPP} \triangleq & \forall A\ \pi\ \hat{\pi}.\ \pi \approx \hat{\pi} \Rightarrow \\ & \quad (\forall\ B.\ [\![A[B]]\!] \subseteq \pi) \Rightarrow \\ & \quad (\forall\ B.\ [\![A[B]]\!]_\rho \subseteq \hat{\pi}) \end{aligned}$$

Proving both Correctness and Property-Preservation Using Bisimulation We define the bisimulation relation $\mathfrak{R}$ between states of the abstract machine and states of Tagged C, indexed by an allocator α. 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.

[SNA: And then I started to write out the relation itself, but I need to do another pass for it to make sense, so I'm commenting it out for now. That's next on my list, though: including enough concrete state stuff that it will make sense.]