```
int* master_pwd;
int len;
bool armed = false;

void fire_missiles() {
  if (armed) {
    __sys_fire_missiles() // talk to the hardware
  }
}

void init(int _len) {
  len = _len;
  master_pwd = malloc(sizeof(int)*len);
  for (int i = 0; i < len; i++) {
    master_pwd[i] = 0x42;
  }
}

int check_pwd(int* pwd) {
  for (int i = 0; i < len; i++) {
    if (pwd[i] != master_pwd[i]) {
      return -1; // bad password
    }
  }
  armed = true; // correct password: allow missiles to fire
  return 0; // report success
}
```

Figure 1: Example Password Checker

1 Motivating Example

Consider a system in which a "gatekeeper" function takes as argument a hashed password, checks it against the hash in its own memory, and reports whether it was correct. If correct, it sets a flag that allows missiles to be launched with `fire_missiles`. Is this secure?

There are potential risks to this code in a memory-unsafe setting. An erroneous or hostile caller could cheat by changing the password in the gatekeeper's memory, or else extract and copy it. It could also directly flip the password bit.

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 code that calls it, which belongs to its own compartment, B. The two compartments' memories are kept disjoint, except for the buffer passed as `pwd`, which must be accessible to both. 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.

Abstract Sharing Semantics 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. [TODO: justify flat memory somewhere.] Additionally, there is a shared memory that follows the rules of a standard block-offset model. Any compartment may allocated objects in either its own local memory, or in the shared memory. This takes the form of two distinct calls to `malloc`, one of which we call `malloc_share`.

Regular state is now extended with a compartment id (the active compartment), a map from compartments to flat memories, a block-offset memory, and a memory oracle α (explained later):

$$M \in \mathcal{M} : comp \to mem_{flat}$$

$$\mu \in mem_{bo} : block \to int \rightharpoonup val$$

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

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 any memory, and the appropriately-sized region starting at that base address is allocated in $M[C]$ (with an appropriate functional array update). A call to `malloc_shared`, meanwhile, generates a fresh block identifier b and returns the pointer $(b, 0)$.

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.

$$v ::= \ldots \mid sptr\ b\ off \mid lptr\ addr$$

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. This means that they will only ever be accessible to the compartment that owns them.

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]

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.

```
void driver() {
  init(1);
  int* pwd = malloc(sizeof(int));
  pwd[0] = 0x42;

  if (check_pwd(pwd)) {
    return;
  } else {
    launch_missiles() {
  }
}
```

(a) Driver Code

$$
\begin{aligned}
A[B] \rightsquigarrow \ldots\ & \mathit{store}\ b\ 0\ (\mathit{int}\ \texttt{0x42})\cdot \\
& \mathit{call}\ \texttt{check\_pwd}\ [\mathit{sptr}\ b\ 0]\cdot \\
& \mathit{return}\ (\mathit{int}\ 0)\cdot \\
& \mathit{call}\ \texttt{launch\_missiles}\ [\,]\cdot \\
& \mathit{call}\ \texttt{\_\_sys\_launch\_missiles}\ [\,]\cdot\ldots
\end{aligned}
$$

(b) Partial Trace

Figure 2: Compartment B

We define the set of trace events as calls, returns, and cross-compartment loads and stores.

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

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 $[\![A[B]]\!]$:

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

Properties In Safe Semantics Now that we have a system of traces, we can attempt to define our trace property. Formally, a property π is a set of traces. A program is said to *satisfy* a property if all traces in its trace set are also in that property.

For simplicity, in our example, we selected a pretty simple (hashed) password, just the repetition of `0x42` for some number of words; in this discussion let `len = 1`. We can define the property π_{launch_safe} as the set of all traces that do not contain *call* `__sys_fire_missiles` and those in which that event is preceded by *store b ofs* `0x42` and *call* `check_pwd` [*sptr b ofs*]. See Figure 2 for an example of code that might be in B, and the resulting trace.

The example B compartment in Figure 2 satisfies the property when linked with A. But if we don't know much about the code that we link against, we want to know that *any* compartment B will satisfy the property. This is called *robust satisfaction*. A policy π is *robustly satisfied* by a compartment A if, for all B, $[\![A[B]]\!] \subseteq \pi$. Ideally, we would want to know that A robustly satisfies the property.

Writing properties out in the ad-hoc format above is challenging. More user-friendly approaches include domain-specific languages like linear temporal logic. It's also easy to get them wrong! Proving or otherwise enforcing a weak property will be little comfort on seeing the missiles fly.

But even without proving or even rigorously defining a specific property of interest, the abstract semantics matches the intuitive behavior of programs, so as long as the abstract semantics are reproduced in our concrete implementation, observers examining this code can reason about its behavior in different scenarios and convince themselves that it is secure.

2 Implementing Compartmentalization in Tagged C

The concrete machine is Tagged C with a compartmentalization policy. We give it 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, and cross-compartment loads and stores.

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

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 $[\![A[B]]\!]_\rho$ similarly to above.

The intuition behind our notion of correct 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. 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), $\Gamma \in \mathit{block} \rightharpoonup (\mathit{int} \times \mathit{int})$, which we write $\Gamma \vdash e \sim \hat{e}$. Defined inductively on values:

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

Which extends naturally to events:

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

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

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.

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

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

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

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 placing internal accesses in the memory. 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 Property Preservation The following definitions are based on those of Abate et al. (TODO: cite). Abstract and concrete properties are related if, for every trace in one, all related traces are in the other:

$$\pi \approx \hat{\pi} \triangleq \forall t\ \hat{t}.\ t \approx \hat{t} \Rightarrow (t \in \pi \Leftrightarrow \hat{t} \in \hat{\pi})$$

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 $\pi \approx \hat{\pi}$.

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

Taking The Contrapositive It's a major hassle to quantify over properties, so we prefer to prove a different metaproperty, RPC. This is the "property-free" variant of RPP. It is provably equivalent.

$$\begin{aligned} RPC \triangleq \forall A\ B\ \alpha\ \hat{t}.\ \alpha \vdash A[B] \rightsquigarrow_\rho \hat{t} \Rightarrow \\ \exists B'\ \alpha'\ t.\ \alpha' \vdash A[B'] \rightsquigarrow t \wedge t \approx \hat{t} \end{aligned}$$

This means that if we have a function that, given any A, B, α, and $\hat{t}$, constructs a B', α', and t such that $A[B'] \rightsquigarrow t$ and $t \approx \hat{t}$, that will be sufficient to prove RPC and by extension RPP. Such a function is called a *backtranslation*. We will specifically implement it using a technique called *dataflow backtranslation*. The basic proof technique is that we build an even richer trace model, one that also tracks the flow of data through private variables, and consider the trace produced by $A[B]$ under that model. We construct a B' that duplicates (a finite prefix of) that trace in the abstract machine.

Consulting the Oracle So far I've been very abstract about the role that the oracle plays in all of this. This section is a stub to give a very lightweight description of how I'm thinking about the oracle. We model an oracle as a (co-inductive) stream of pairs of integers.

$$\alpha ::= (sz, addr) : \alpha'$$

The first integer is the size of the next block that will be allocated, and the second is its base address. The oracle therefore corresponds to the behavior of the compiler and allocator in a particular run of a program—it knows the order in which blocks of various sizes will be allocated in that run. Just to avoid any weirdness, any universal quantification over oracles that will then produce a trace representing an execution should be restricted to those oracles that actually capture the ordering of that execution. Fortunately if we ever need to produce such an oracle, we can simply define it as the one that corresponds to the allocator's actual behavior on a given program.

But in our actual proof, we are just going to be given an arbitrary allocator. The concern is what happens if our backtranslation involves allocating additional memory. Provided that there is sufficient memory available, we simply insert a new pair at the head of the allocator stream, in a usable location. It is far from clear that we actually will need to allocate the additional memory, though.

Other TODO: Justify semantics vs. stack-safety-style trace properties