Flexible Runtime Security Enforcement with Tagged C

Sean Anderson, Allison Naaktgeboren, and Andrew Tolmach

Portland State University

Abstract. Today's computing infrastructure is built atop layers of legacy C code, often insecure, poorly understood, and/or difficult to maintain. These foundations may be shored up with dynamic security enforcement. Tagged C is a C variant with a built-in *tag-based reference monitor*. It can express a variety of dynamic security policies and enforce them with compiler and/or hardware support.
Tagged C expresses security policies at the level familiar to C developers: that of the C source code rather than the ISA. It is comprehensive in supporting different approaches to security as well as more or less restrictive policies. We demonstrate this range by providing examples of *memory safety*, *compartmentalization*, and *secure information flow* (SIF) policies. We also give a semantics and reference interpreter for Tagged C.

1 Introduction

Many essential technologies rely on new and old C code. Operating systems (Linux, Windows, OSX, BSD), databases (Oracle, sqlite3), the internet (Apache, NGNIX, NetBSD, Cisco IOS), the Internet of Things (IoT), and the embedded devices that run our homes and hospitals are built in and on C [11]. C is not a relic; more than a third of professional programmers report active developing in C today [12]. The safety of these essential technologies depends on the security of their underlying C codebases. Insecurity might take the form of undefined behavior such as memory errors (e.g. buffer overflows, heap leaks, double-free), logic errors (e.g. sql injection, input-sanitization flaws), or larger-scale architectural flaws (e.g. over-provisioning access rights.)

Although static analyses can detect and mitigate many C insecurities, the last line of defense against undetected or unfixable vulnerabilities is runtime enforcement. Ideally an engineer can tune enforcement to the security needs of the system, rather than apply conservative, one-size-fits-all restrictions. We introduce Tagged C: a general-purpose dynamic tag-based enforcement language that allows developers to define flexible security policies in terms of a familiar source language. Applications of Tagged C include defining undefined behaviors (UBs) involving memory, specifying detailed information flow policies, and enforcing arbitrary mandatory access control rules.

Tagged C's novel approach is that it is a *general-purpose* scheme for specifying security policies at the *source level*, using a tag-based reference monitor.

This style of monitor associates a metadata tag with the data in the underlying system. Throughout execution it updates these tags according to a set a predefined rules, or halts if the program would violate a rule. We are motivated by PIPE (Programmable Interlocks for Policy Enforcement) [], an ISA extension that implements such a reference monitor in hardware. While our scheme is general and could be implemented in software, we aim for compatibility with PIPE as a likely hardware target. PIPE is notable among similar systems such as ARM MTE and [that thing from Binghamton] in that its tags are very large—typically the size of words in the underlying ISA. This makes PIPE extremely flexible and able to run multiple policies at once. TODO: non-recompilation?]

Tagged C consists of an underlying semantics that establishes the baseline concrete behavior of programs with no policies, and a set of *control points* at which the semantics consult a user-defined set of *tag rules*. For convenience we build our underlying semantics on the CompCert C semantics, which are formalized as part of the CompCert verified compiler [8]. We provide a reference interpreter also based on that of CompCert, for use in executing prototype policies. Tag rules are written directly as Gallina functions.

Contributions We offer the following contributions:

- The design of a comprehensive set of *control points* at which the language interfaces with the policy
- Tagged C policies implementing (1) compartmentalization, (2) a realistic, permissive memory model from the literature (PVI), and (3) Secure Information Flow (SIF)
- A full formal semantic definition for Tagged C, formalized in Coq, describing how the control points interact with programs
- A Tagged C interpreter, implemented in Coq and extracted to Ocaml

[TODO: rework] In the next section, we give a high-level introduction of metadata-tagging: how it works, and how its use can improve security. Then in **??**, we briefly discuss the language as a whole, before moving into policies in section 4. Finally, in **??** we discuss the degree to which the design meets our goals of flexibility and applicability to realistic security concerns.

2 What is Metadata Tagging?

Consider a seemingly straightforward security requirement: "do not leak the passkey, `psk`." In the example in fig. 1, `psk` is passed to `f` as an argument, copied to a local variable, and then printed—clearly a leak of `psk`. Figure 1 maps three points in the execution of `f` to a table showing the contents of the program state, with the input value and all tags treated symbolically. Let i be the value passed to `psk` and vt_0 the corresponding tag. In each state, the first column shows the active function, the second gives the symbolic values and tags of variables in the local environment, and the third shows the origins of those tags in tag rules.

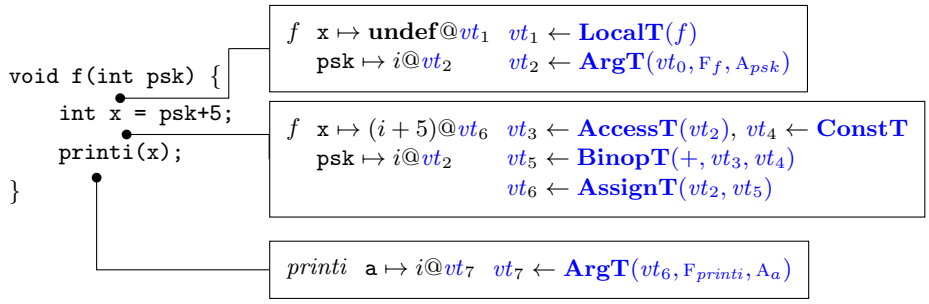

Fig. 1: Example 1

The initial tag on `x`, vt_1, comes from the **LocalT** tag rule, which can be specialized by the identity of the current function. The tag on `psk` comes from **ArgT**, which is parameterized by the tag on the argument value (vt_0) and the identities of both the function and the argument. Next, `x` is assigned `psk+5`. There are four total tag rules invoked during this statement: **AccessT** when reading from `psk`, **ConstT** for the tag on the constant, **BinopT** for the addition, and finally **AssignT** for the assignment into `x`. **BinopT** is parameterized by the identity of the operation in addition to the tags on its inputs. After the final step, the execution is in the function `printi`, where it consults **ArgT** a second time.

The system can prevent the program from outputting `psk` by instantiating these tag rules with functions that implement *secure information flow*. For the tag rules seen so far, the policy given in fig. 2 would do the trick. It defines two tags, H and L, for high security (`psk` and things derived from it) and low security (everything else.) In tag rules, the assignment operator := denotes an assignment to the named tag-rule output. "Case" statements may be abbreviated by assuming that any input that doesn't match a case causes a failstop.

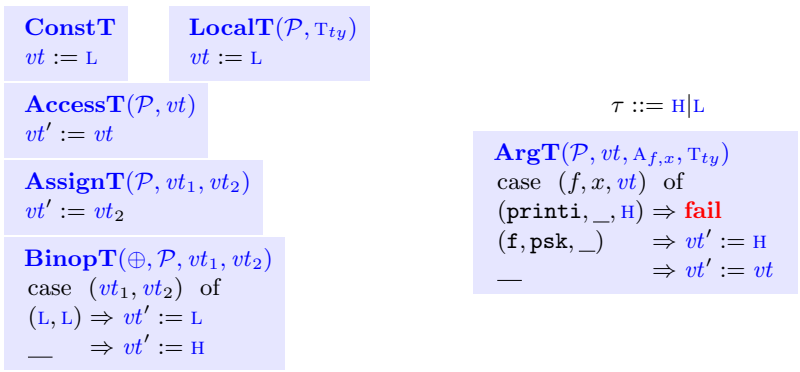

Fig. 2: Secure Information Flow (pt. 1)

The interesting rules are **BinopT** and **ArgT**. **BinopT** combines two tags, setting the result of a binary operation H if either of its arguments are. **ArgT** is parameterized by the identifier of the function argument being processed. It tags `psk` H, and maintains the security level of all other arguments. But if a H value is being passed to a parameter of `printi`, the rule will fail. So, in this example, we will be unable to generate a tag vt_5, and the tag processor will throw a failstop rather than allow execution to continue.

Example 3 adds two new wrinkles: we need to keep track of metadata associated with addresses and with the program's control-flow state. If the variable `mm` represents mmapped memory that can be seen publicly, we want to avoid storing the password there. And, by branching on the password, we risk leaking information that could eventually compromise it even if we never print it directly (an *implicit flow*.)

Global variables like `mm` are always kept in memory, as are locals with reference types like arrays. Other parameters and locals are placed in memory if they have their addresses taken. Objects in memory have additional "location tags" that are notionally associated with the memory itself, which we will range over with lt by convention. The store maps their identifier to their address, along with its own "pointer tag" which we will distinguish with the metavariable pt. To track metadata about the overall system state, we also add a special global tag called the PC Tag, ranged over by $\mathcal{P}$.

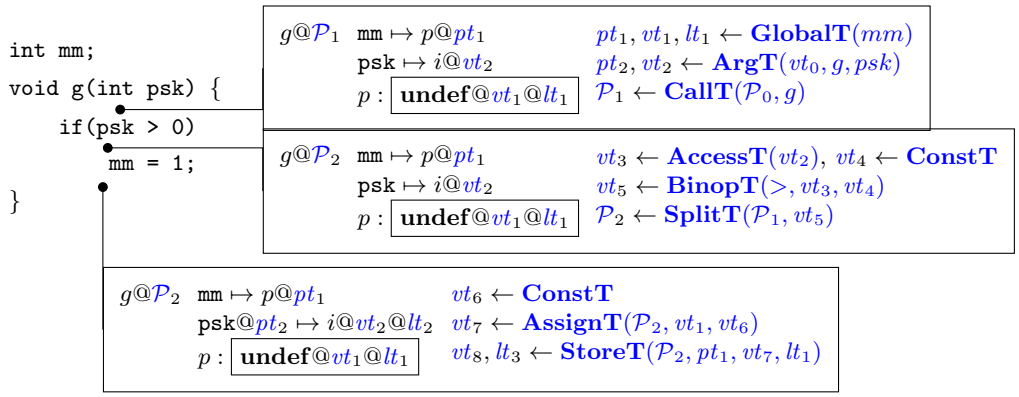

Fig. 3: Implicit Flows and Memory

Tagged C initializes the tags on `mm` with the **GlobalT** rule. The PC Tag at the point of call, $\mathcal{P}_0$, is fed to **CallT** to determine a new PC Tag inside of `g`. And the if-statement consults the **SplitT** rule to update the PC Tag inside of its branch based on the value-tag of the expression `psk < 0`. Once inside the conditional, when the program assigns to `mm`, it must consult both the **AssignT** rule as normal and the **StoreT** rule because it is storing to memory.

To upgrade our "don't print the password" policy to "don't leak the password," we can keep most of our rules the same, or extend them in natural ways. Crucially, we will tag memory locations H by default, indicating that they are allowed to contain H-tagged values, but `mm` will be tagged L. The most interesting rules are:

```
GlobalT(G_x, T_ty)
vt := L
pt := L
case x of
mm ⇒ lt := L
_  ⇒ lt := H
```

```
SplitT(P, vt)
case (P, vt) of
(L, L) ⇒ P' := L
_      ⇒ P' := H
```

```
StoreT(P, pt, vt, lt)
lt' := lt
case (P, vt, lt) of
(_, _, H) ⇒ vt' := vt
(L, L, L) ⇒ vt' := vt
_         ⇒ fail
```

In this case, **SplitT** will set the PC Tag to H, as it branches on a value derived from `psk`. Then, when it comes time to write to `mm`, **StoreT** will fail rather than write to a low address in a high context.

3 The Language, Informally

Tagged C uses the full syntax of CompCert C [8] with minimal modification (fig. 11). There are two notable syntactical differences in the language, relative to CompCert C: conditionals and loops take an optional *join point* label, and parenthetical expressions an optional "context tag."

Our semantics are a small-step reduction semantics which differ from CompCert C's in two key respects. These are given in full in the appendix. First, Tagged C's semantics contain *control points*: hooks within the operational semantics at which the tag policy is consulted and either tags are updated, or the system failstops. (Control points resemble "advice points" in aspect-oriented programming, but narrowly focused on the manipulation of tags.) A control point consists of the name of a *tag rule* and the bindings of its inputs and outputs; a tag rule is a partial function. The names and signatures of the tag rules, and their corresponding control points, are listed in Table 1.

Second, there is no memory-undefined behavior: the source semantics reflect a concrete target-level view of memory as a flat address space. Without memory safety, programs that exhibit memory-undefined behavior will act as their compiled equivalents would, potentially corrupting memory; we expect that a memory safety policy will be a standard default, but that the strictness of the policy may need to be tuned for programs that use low-level idioms.

The choice of control points and their associations with tag rules, as well as the tag rules' signatures, are a crucial design element. Our proposed design is sufficient for the three classes of policy that we explore in this paper, but it may not be complete.

Rule Name	Inputs	Outputs	Control Points
LoadT	$\mathcal{P}, pt, vt, \overline{lt}$	vt'	Memory Loads
StoreT	$\mathcal{P}, pt, vt, lt$	$\mathcal{P}', vt', lt'$	Memory Stores
UnopT	$\odot, \mathcal{P}, vt$	vt	Unary Operation
BinopT	$\oplus, \mathcal{P}, vt_1, vt_2$	vt'	Binary Operation
ConstT		vt	Applied to Constants/Literals
ExprSplitT	$\mathcal{P}, vt$	$\mathcal{P}'$	Control-flow split points in expressions
ExprJoinT	$\mathcal{P}, \mathcal{P}', vt$	$\mathcal{P}'', vt'$	Join points in expressions
SplitT	$\mathcal{P}, vt, L$	$\mathcal{P}'$	Control-flow split points in statements)
LabelT	$\mathcal{P}, \mathrm{L}_L$	$\mathcal{P}'$	Labels/arbitrary code points
CallT	$\mathcal{P}, \mathrm{F}_f, \mathrm{F}_{f'}$	$\mathcal{P}'$	Call
ArgT	$\mathcal{P}, vt, \mathrm{A}_{f,x}, \mathrm{T}_{ty}$	$\mathcal{P}', pt, vt', \overline{lt}$	Call
RetT	$\mathcal{P}_{CLE}, \mathcal{P}_{CLR}, vt, \mathrm{F}_f$	$\mathcal{P}', vt'$	Return
GlobalT	$\mathrm{G}_x, \mathrm{T}_{ty}$	$pt, vt, \overline{lt}$	Program initialization
LocalT	$\mathcal{P}, \mathrm{T}_{ty}$	$\mathcal{P}', pt, vt, \overline{lt}$	Call
DeallocT	$\mathcal{P}, \mathrm{T}_{ty}$	$\mathcal{P}', vt, \overline{lt}$	Return
ExtCallT	$\mathcal{P}, \mathrm{F}_f, \mathrm{F}_{f'}, \overline{vt}$	$\mathcal{P}'$	Call to linked code
MallocT	$\mathcal{P}, \mathrm{F}_f, \mathrm{F}_{f'}, vt$	$\mathcal{P}', pt, vt, \overline{lt}$	Call to `malloc`
FreeT	$\mathcal{P}, vt$	$\mathcal{P}', pt, vt, \overline{lt}$	Call to `free`
FieldT	$pt, \mathrm{T}_{ty}, \mathrm{G}_x$	pt'	Field Access
PICastT	$\mathcal{P}, pt, vt, \overline{lt}$	vt	Cast from pointer to scalar
IPCastT	$\mathcal{P}, vt_1, vt_2, \overline{lt}$	pt	Cast from scalar to pointer
PPCastT	$\mathcal{P}, pt, vt, \overline{lt}$	pt'	Cast between pointers
IICastT	$\mathcal{P}, vt_1$	pt	Cast between scalars

Table 1: Full list of tag-rules in control points

Parts of a policy A policy consists of instantiations of the tag type and each of the tag rules associated with control points in the semantics. Table **??** identifies the full collection of control points, their tag rules, and the inputs and outputs of the tag rules. The tag type τ must be inhabited by a default tag.

Identifiers Identifiers are a C source-level construct, and a policy designer might want to operate on them, so we embed them in tags. These are called *name tags*. We give name tags to the following constructions and identify them as follows:

- Function identifiers, F_f
- Function arguments, $\mathrm{A}_{f,x}$
- Global variables, G_x
- Labels, L_L
- Types, T_{ty}

4 Tags and Policies

The heart of Tagged C is in its security policies. Using the control points shown in section 3, we will walk through three example policies: PVI memory safety, compartmentalization, and secure information flow.

[TODO: discussion of combining policies]

4.1 PVI Memory Safety

Memory safety is defined relative to a *memory model*—a formal or informal description of how a high-level language handles memory. The memory model associated with the C standard leaves many behavior undefined, including some that are in use in practice [10]. Memarian et al. have proposed two alternatives that define useful subsets of these behaviors [9]. We choose their "provenance via integer" (PVI) memory model as our example.

Variations of memory safety have been enforced in PIPE already, but usually using an ad hoc memory model. PVI has the virtue of giving definition to many memory UBs in which a pointer is cast to an integer, subjected to various arithmetic operations, and cast back to a pointer. Their second memory model, *PNVI* (provenance not via integer), is even more permissive. We can also enforce it in Tagged C, though its security value is questionable, and we will not describe it in this paper.

For a policy to enforce PVI, it should not failstop on any program that is defined in PVI. However, it should failstop if and when a program reaches UB. For example, low-level code that uses the lower-order bits of a pointer to store a flag (**??**) should execute successfully. This idiom appears often in code where a pointer is associated with a flag, such as Cheney's garbage collection algorithm [] and is generally considered harmless. But violations such as buffer overflows (**??**) in which a load, store, or free would occur outside of the proper range are undefined in PVI and must failstop. In **??**, memory is arranged so that the overflow ends up overwriting the variable `y`.

```
void overflow() {
    int[3] x; int y;
    *x = 222222;
    *(x+3) = 333333;
}
```

$overflow@\mathcal{P}_2$ $\quad \texttt{x} \mapsto \underline{84}@pt_1)$
$\qquad\qquad\qquad\quad \texttt{y} \mapsto \underline{96}@pt_2$

84				88				92				96			
undef @ vt_1				**undef** @ vt_1				**undef** @ vt_1				**undef** @ vt_2			
lt_1	lt_1	lt_1	lt_1	lt_1	lt_1	lt_1	lt_1	lt_1	lt_1	lt_1	lt_1	lt_2	lt_2	lt_2	lt_2

84				88				92				96			
222222 @ vt_3				**undef** @ vt_1				**undef** @ vt_1				333333 @ vt_4			
lt_3	lt_3	lt_3	lt_3	lt_1	lt_1	lt_1	lt_1	lt_1	lt_1	lt_1	lt_1	lt_4	lt_4	lt_4	lt_4

?? shows the initial state of memory on entry to `overflow` and the state after the assignment to `*(x+3)`, assuming that the PC Tag is $\mathcal{P}_0$ at the call. The tags associated with `x` $(\mathcal{P}_1, pt_1, vt_1, \overline{lt}_1)$ come from **LocalT**$(\mathcal{P}_0, \mathrm{T}_{int})$ and those associated with `y` come from **LocalT**$(\mathcal{P}_1, \mathrm{T}_{int})$.

The write to `*(x+3)` is of particular interest. We tag the result of `x+3` with $pt' \leftarrow \mathbf{BinopT}(\mathcal{P}_2, pt_1, \mathbf{ConstT})$; then for the store itself, we use the rule $vt_4, \overline{lt}_4 \leftarrow \mathbf{StoreT}(\mathcal{P}_2, pt', vt_2, \mathbf{ConstT}, \overline{lt}_2)$.

We can prevent overflows like this using a *memory safety* policy. In brief, whenever an object is allocated, it is assigned a unique "color," and its memory locations as well as its pointer are tagged with that color. Pointers maintain their tags under arithmetic operations, and loads and stores are legal if the pointer matches the target memory location. The rules for the PVI memory safety policy are given in fig. 4. In this case, we will have $pt_1 = lt_1 = 0$ and $pt_2 = lt_2 = 1$. When we try to write to `x+3`, we compare $pt' = 0$ with lt_2, and failstop because they differ.

$\tau ::= clr \mid N$ $\qquad clr \in \mathbb{N}$

BinopT$(\oplus, \mathcal{P}, vt_1, vt_2)$
case (vt_1, vt_2) of
$(t, N) \Rightarrow vt' := t$
$(N, t) \Rightarrow vt' := t$
$(clr_1, clr_2) \Rightarrow vt' := N$

UnopT$(\odot, \mathcal{P}, vt)$
$vt' := vt$
$\{ vt \}$

LocalT$(\mathcal{P}, \mathrm{T}_{ty})$
$\mathcal{P}' := \mathcal{P} + 1$; $pt := \mathcal{P}$
$vt := N$; $\overline{lt} := [\mathcal{P}]$

MallocT$(\mathcal{P}, \mathrm{F}_f, \mathrm{F}_{f'}, vt)$
$\mathcal{P}' := \mathcal{P} + 1$; $pt := \mathcal{P}$
$vt := N$; $\overline{lt} := [\mathcal{P}]$

LoadT$(\mathcal{P}, pt, vt, \overline{lt})$
assert $\forall lt \in \overline{lt}.pt = lt$
$vt' := vt$

StoreT$(\mathcal{P}, pt, vt, lt)$
assert $\forall lt \in \overline{lt}.pt = lt$
$\mathcal{P}' := \mathcal{P}$; $vt' := vt_2$; $\overline{lt}' := \overline{lt}$

Fig. 4: PVI Memory Safety Policy

The cast rules, meanwhile, have no effect on the tag of the value being cast at all. So, we can cast a pointer to a scalar value, perform any operation that is defined on that type on it, and cast it back, and it will retain its pointer tag. As long as it ends up pointing at the same object, loads and stores will be successful. Function pointers are an exception: Tagged C's underlying control-flow protections prevent them from being tampered with.

4.2 Compartmentalization

In a perfect world, all C programs would be memory safe. This world is imperfect and it is common for a codebase to contain undefined behavior that will not be fixed. Developers intentionally use low-level idioms that are UB [10], or the cost and risk of regressions may make it undesirable to fix bugs in older code [3].

A compartmentalization policy can isolate potentially risky code, such as code with known UB, from safety-critical code, and enforce the principle of least privledge. It limits the possible damage from exploitation to the containing compartment. It may also restrict how code in one compartment may interact with another even in the absence of language-level errors. Ideally, each component has only the *least privilege* necessary to complete its task. This popular defense

can be implemented at many levels. It is often built into a system's fundamental design, like a web browser sandbox untrusted javascript. For our use-case, we consider a compartmentalization scheme being added to the system after development and we have a set of identifiers and a map provided by a security engineer. The compartmetn identiifers are ranged over by C, and function identifiers are mapped to compartments by $comp(f)$.

Coarse-grained Protection The core of a compartmentalization scheme is typicall memory protection. In the simplest version, memory allocated by a function is only accessible by functions within its compartment. Thus the system keeps track of the active compartment, using the PC tag.

Calls and returns each take two steps: first to an intermediate call or return state, and then to the normal execution state, as shown in fig. 5 with some function f calling f'. In the initial call step, **CallT** uses the name-tags of the caller and callee to update the PC Tag. Then, in the step from the call state, we place the function arguments in the temp environment, tagging their values with the results of **ArgT**, and we allocate our stack locals, tagging their values and locations with the results of **LocalT**. And on return, we deallocate locals and update their location tags with **DeallocT** when stepping to a return state, and from there **RetT** updates both the PC Tag and the tag on the returned value.

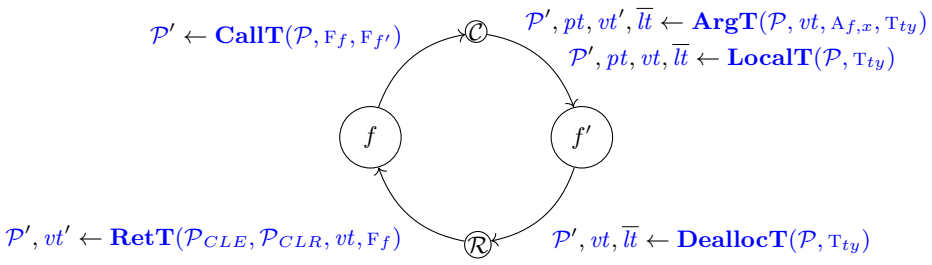

Fig. 5: Structure of a function call

In our compartmentalization policy (fig. 6), we define a tag to be a compartment identifier or the default N tag. At any given time, the PC Tag carries the compartment of the active function, kept up to date by the **CallT** and **RetT** rules.

The remainder of the policy works much like memory safety, except that coarse-grained protection means that the "color" we assign to an allocation is the active compartment, and during a load or store, we compare the location tags to the PC Tag, not the pointer.

Sharing Memory The above policy works if our compartments only ever communicate by passing non-pointer values. In practice, this is far too restrictive!

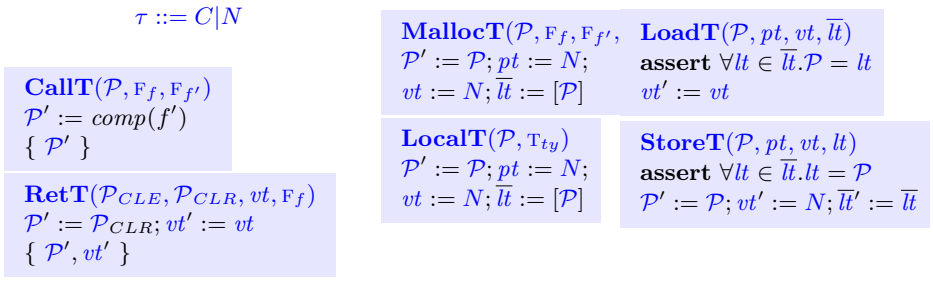

Fig. 6: Compartmentalization Policy

Many library functions take pointers and operate on memory shared with the caller. External libraries are effectively required for most software to function yet represent a threat. Isolating external libraries from critical code prevents vulnerabilities in the library from compromising critical code and deprives potential attackers of ROP gadgets and other tools if there is an exploit in the critical code.

To allow intentional sharing of memory across compartments, a more flexible policy is needed. Suppose for example the hostname needs to conform to an expected pattern, such as in an enterprise network, to differentiate between different classes of computers (employee, server, contractor, etc). The standard library, over in its own compartment, has helpful functions, provided the caller provides the buffers from which to set or get the hostname.

```
void configure_enterprise(char* intended_name) {
  int ret = 0;
  char* curr_name = malloc(HOST_NAME_MAX + 1);
  ret = gethostname( &curr_name, HOST_NAME_MAX + 1 );
  if (! ret && !(strcmp(curr_name, intended_name))) {
    ret = sethostname(intended_name), strlen(intended_name);
    ....
  }
  ....
}
```

The literature contains two main approaches to this problem: *mandatory access control* and *capabilities*. The former explicitly enumerates the access rights of each compartment, while the latter turns passed pointers into unforgeable tokens of privilege, so that the act of passing one implicitly grants the recipient access.

Tagged C can enforce either; here we will demonstrate a capability approach in which we delineate allocations that may be passed and those that must not. At the syntactic level we separate these by creating a variant identifier for `malloc`, `malloc_share`. This identifier maps to the same address (i.e., it is still calling the same function) but its name tag differs and can therefore parameterize the

tag rule. The source must have the malloc name changed for every allocation that might be shared. The annotation could be performed manually, or perhaps automatically using some form of escape analysis.

Seen in fig. 7, the policy works by gluing compartmentalization and memory safety together. The PC Tag carries the current compartment and the next color for shared allocations, and **MallocT** uses the function tag to determine which to attach to the pointer and allocated region. During loads and stores, the location tag of the target address determines whether access is restricted via the identity of the active compartment or the validity of the pointer.

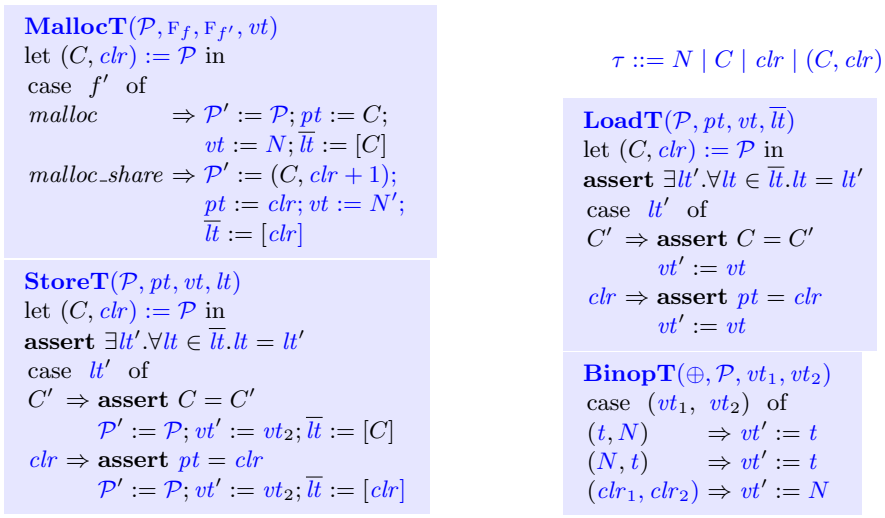

Fig. 7: Compartmentalization with Shared Capabilities

4.3 Secure Information Flow

Our final example policy will be a more realistic version of our first: *secure information flow (SIF)*. SIF is described in the venerable Denning and Denning [5], and part of a larger family of policies known as *information flow control* (IFC.) This family of policies deal entirely with enforcing higher-level security concerns, regardless of whether the code that they protect contains errors or undefined behaviors. We will give an example of a single policy in the family. Our introductory example was an instance of *confidentiality*, so now we will discuss *integrity*: preventing insecure input from influencing secure behavior. In this code, a malformed user input is accidentally appended to an sql query without sanitization:

```
void sanitize(char* in, char* out);
```

```
char* sql_query(char* query);

void get_data() {
  char[20] name;
  char[20] name_san;
  char[100] query = "select address where name =";

  scanf("%19f", name);
  sanitize(name, name_san);
  strncat(query, name, strlen(name));

  sprintf(buf, sql_query(query));
  return;
}
```

This function sanitizes its input `name`, then appends the result to an appropriate SQL query, storing the result in `buf`. But, in the default case, the programmer has accidentally used the unsanitized string! This creates the opportunity for an SQL injection attack: a caller to this function could (presumably at the behest of an outside user) call it with `field` of 3 and `name` of "Bobby; drop table;".

The fact that the input can be sanitized also makes this an *intransitive* policy: information may flow from `scanf` to `sanitize`, and from `sanitize` to `sql_query`, but not directly from `scanf` to `sql_query`.

$$\tau ::= \mathrm{H} \mid \mathrm{L} \mid pc\ f\ e\ \overline{L} \quad e \in list\ \mathbb{B}$$

$$|t| \triangleq \begin{cases} \mathrm{L} & \text{if } t = \mathrm{L} \text{ or } t = pc\ f\ e\ \emptyset \text{ where every } b \in e \text{ is } \mathbf{f} \\ \mathrm{H} & \text{otherwise} \end{cases}$$

$$t_1 \sqcup t_2 \triangleq \begin{cases} \mathrm{L} & \text{if } |t_1| = |t_2| = \mathrm{L} \\ \mathrm{H} & \text{otherwise} \end{cases}$$

StoreT$(\mathcal{P}, pt, vt, lt)$
let $pc\ f\ ets\ \overline{L} := \mathcal{P}$ in
case f of
scanf $\Rightarrow \mathrm{H}$
sanitize $\Rightarrow \mathrm{L}$
_____ $\Rightarrow \mathcal{P}' := \mathcal{P}$
$vt' := \mathcal{P} \sqcup vt \sqcup pt$
$\overline{lt}' := \overline{lt}$

LoadT$(\mathcal{P}, pt, vt, \overline{lt})$
let $pc\ f\ ets\ \overline{L} := \mathcal{P}$ in
case f of
sql_query $\Rightarrow$ **assert** $vt \sqcup pt = \mathrm{L}$
$vt' := vt$
_____ $\Rightarrow vt' := vt$

BinopT$(\oplus, \mathcal{P}, vt_1, vt_2)$
$vt' := vt_1 \sqcup vt_2$

Fig. 8: SIF Policy: Tags and Selected Rules

Since we care about a single source, we can once again use the standard division of H and L. We additionally need to carry significant information on the

```
int f(bool secret) {
    int public1, public2;

S:  if (secret) {
b1:     public1 = 1;
    } else {
b2:     public1 = 0;
    }

J:  public2 = 42;
    return public2;
}
```

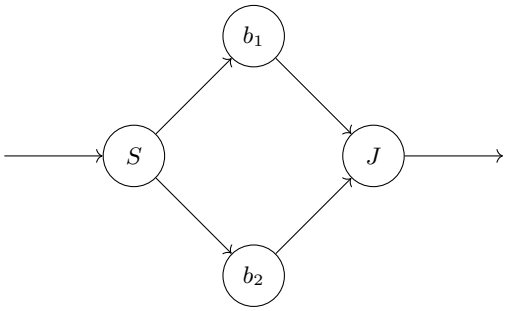

Fig. 9: Leaking via if statements

PC Tag. First, we track the current function identifier. Beyond that, we need two pieces of information to deal with *implicit flows*, situations where the control-flow of the program is influenced by a secret. In our initial example we simplified these. To keep track of tainted scopes, the PC Tag carries a list of booleans, e, and a set of label identifiers, L. We will discuss these in detail below. Initially, the PC Tag is $pc\ f\ \varepsilon\ \emptyset$, and all values and memory locations are tagged L.

The detailed tag rules for expressions are given in fig. 8. We define two operators on tags: the "join" operator $\sqcup$ takes the higher of two security levels, and the "reduce" operator $|\cdot|$ converts a PC Tag into a security level, H or L. The function `scanf` taints all of its writes by marking them H. This extends to its return value in **RetT** as well (not shown.) By the same token, all outputs of `sanitize` are tagged L, so that when it copies H-tagged data into its output buffer, we consider those data safe.

In this scenario, our policy aims to prevent `sql_query` from recieving tainted data. For this reason, we failstop if `sql_query` would load a tainted value.

Implicit Flows Things become trickier when we consider that the program's control-flow itself can be tainted. This can occur in any conditional, including loops, conditional statements, and conditional expressions. In general, anytime a "split" is conditioned on a tainted value, subsequent assignments must also be tainted. An example can be seen in fig. 9, where labels in the code indicate the split point, branches, and *join point* J.

A join point is the node in the program's control-flow graph where all possible routes from the split to a return have re-converged; its its immediate post-dominator []. [TODO: this is the Denning cited in Bay and Askarov] At this point, an observer can no longer deduce which path execution took, except through the assignments that happened in b_1 or b_2, which are already tainted. It is therefore safe to tag future assignments L. In the example, `public1` should be tagged H, while `public2` is tagged L.

We assume a *termination-insensitive* setting [2], in which we allow an observer to glean information by the termination or non-termination of the pro-

gram. This is a necessary limitation of an enforcement mechanism that halts execution. Having accepted this limitation, we may apply the same analysis to loops as well as conditionals.

?? gives the tag rules for handling conditionals and loops. All branching statements are governed by the same **SplitT** rule, and their join points by the **LabelT** rule when the label is placed on a join point. Branching expressions are likewise handled by the **ExprSplitT** rule, and the **ExprJoinT** rule. This last needs no label, due to the predictable structure of nested expressions.

SplitT$(\mathcal{P}, vt, L)$
let $pc\ f\ e\ \overline{L} := \mathcal{P}$ in
case vt of
$\text{H} \Rightarrow \mathcal{P}' := pc\ f\ e\ (L \cup \overline{L})$
$\text{L} \Rightarrow \mathcal{P}' := \mathcal{P}$

LabelT$(\mathcal{P}, \text{L}_L)$
let $pc\ f\ e\ \overline{L} := \mathcal{P}$ in
$\mathcal{P}' := pc\ f\ e\ \overline{L} - L$
$\{\ \mathcal{P}'\ \}$

ExprSplitT$(\mathcal{P}, vt)$
let $pc\ f\ e\ \overline{L} := \mathcal{P}$ in
case vt of
$\text{H} \Rightarrow \mathcal{P}' := pc\ f\ \mathbf{t} :: e\ \overline{L})$
$\text{L} \Rightarrow \mathcal{P}' := pc\ f\ (\mathbf{f} :: e\ \overline{L})$
$\{\ \mathcal{P}'\ \}$

ExprJoinT$(\mathcal{P}, \mathcal{P}', vt)$
let $pc\ f\ b :: e\ \overline{L} := \mathcal{P}$ in
$\mathcal{P}' := pc\ f\ e\ \overline{L}$
$\{\ \mathcal{P}'', vt'\ \}$

Fig. 10: SIF Conditionals

Of course, code does not generally come with labeled join points, and they must be associated with their split points. We introduce additional forms of the if, while, do-while, for, and switch statements which carry an additional label, and perform some preprocessing to associate these with new labels in the source code. This preprocessing step generates the program's control flow graph and, for each branch, identifies its immediate post-dominator. That node is labeled with a fresh identifier, and the same identifier is added to the original conditional statement.

5 Implementing Tagged C with PIPE

Chhak et al. [4] introduce a verified compiler from a toy high-level language with tags to a control-flow-graph-based intermediate representation of a PIPE-based ISA. It is a proof-of-concept of compilation from a source language's tag policy to realistic hardware. Everything in a PIPE system carries tags, including instructions. Instruction tags are statically determined at compile-time, so they can carry data about source-level control points in the corresponding assembly. This means that PIPE can emulate any given Tagged C policy by running two policies in parallel: a basic stack-and-function-pointer-safety policy to mimic Tagged C's high-level control-flow, and the source-level policy as written.

Chhak et al.'s general strategy for mapping Tagged C's tag rules sometimes requires adding extra instructions to the generated code. A Tagged-C control

point may require a tag from a location that is not read under a normal compilation scheme, or must update tags in locations that would otherwise not be written. Such instructions are unnecessary overhead if the policy doesn't meaningfully use the relevant tags.

To mitigate this, control points whose compilation would add potentially extraneous instructions take optional parameters or return optional results. We will explain how the rule should be implemented in the target if the options are used. [SNA: I think we have been leaning toward doing so in more detail here, not threaded through the rest of the paper. So that's a TODO.] Optional inputs and outputs are marked with boxes. If a policy does not make use of the options, it will be sound to compile without the extra instructions.

6 Evaluation

Tagged C aims to combine the flexibility of tag-based architectures with the abstraction of a high-level language. How well have we achieved this aim?

[Here we list criteria and evaluate how we fulfilled them]

- Flexibility: we demonstrate three policies that can be used alone or in conjunction
- Applicability: we support the full complement of C language features and give definition to many undefined C programs
- Practical security: our example security policies are based on important security concepts from the literature

6.1 Known Limitations

By committing to a tag-based mechanism, we do restrict the space of policies that Tagged C can enforce. In general, a reference monitor can enforce any policy that constitutes a *safety property*—any policy whose violation can be demonstrated by a single finite trace. This class includes such policies as "no integer overflow" and "pointers are always in-bounds," which depend on the values of variables. Tag-based monitors cannot enforce any policy that depends on the value of a variable rather than its tags.

Due to our approach to tagging memory uniformly at allocation time, Tagged C cannot easily enforce substructural memory safety. Versions of memory safety that protect fields of a struct from overflows within the same struct would be very useful, but currently enforcing them will require manual initialization.

6.2 Unknown Limitations

[TODO]

7 Related Work

Reference Monitors The concept of a reference monitor was first introduced fifty years ago in [1]: a tamper-proof and verifiable subsystem that checks every security-relevant operation in a system to ensure that it conforms to a security *policy* (a general specification of acceptable behavior; see [7].)

A reference monitor can be implemented at any level of a system. An *inline reference monitor* is a purely compiler-based system that inserts checks at appropriate places in the code. Alternatively, a reference monitor might be embedded in the operating system, or in an interpreted language's runtime. A *hardware reference monitor* instead provides primitives at the ISA-level that accelerate security and make it harder to subvert.

Programmable Interlocks for Policy Enforcement (PIPE) [6] is a hardware extension that uses *metadata tagging*. Each register and each word of memory is associated with an additional array of bits called a tag. The policy is decomposed into a set of *tag rules* that act in parallel with each executing instruction, using the tags on its operands to decide whether the instruction is legal and, if so, determine which tags to place on its results. PIPE tags are large relative to other tag-based hardware, giving it the flexibility to implement complex policies with structured tags, and even run multiple policies at once.

Other hardware monitors include Arm MTE, [Binghamton], and CHERI. Arm MTE aims to enforce a narrow form of memory safety using 4-bit tags, which distinguish adjacent objects in memory from one another, preventing buffer overflows, but not necessarily other memory violations. [TODO: read the Binghamton paper, figure out where they sit here.]

CHERI is capability machine [TODO: cite OG CHERI]. In CHERI, capabilities are "fat pointers" carrying extra bounds and permission information, and capability-protected memory can only be accessed via a capability with the appropriate privilege. This is a natural way to enforce spatial memory safety, and techniques have been demonstrated for enforcing temporal safety [14], stack safety [13], and compartmentalization [TODO: figure out what to cite], with varying degrees of ease and efficiency. But CHERI cannot easily enforce notions of security based on dataflow, such as Secure Information Flow.

In this paper, we describe a programming language with an abstract reference monitor. We realize it as an interpreter with the reference monitor built in, and envision eventually compiling to PIPE-equipped hardware. An inlining compiler would also be plausible. As a result of this choice, our abstract reference monitor uses a PIPE-esque notion of tags.

Aspect Oriented Programming [TODO: do forward search from original AOP paper]

8 Future Work

We have presented the language and a reference interpreter, built on top of the CompCert interpreter [8], and three example policies. There are several significant next-steps.

Compilation An interpreter is all well and good, but a compiler would be preferable for many reasons. A compiled Tagged C could use the hardware acceleration of a PIPE target, and could more easily support linked libraries, including linking against code written in other languages. The ultimate goal would be a fully verified compiler, but that is a very long way off.

Language Proofs There are a couple of properties of the language semantics itself that we would like to prove. Namely (1) that its behavior (prior to adding a policy) matches that of CompCert C and (2) that the behavior of a given program is invariant under all policies up to truncation due to failstop.

Policy Correctness Proofs For each example policy discussed in this paper, we sketched a formal specification for the security property it ought to enforce. A natural continuation would be to prove the correctness of each policy against these specifications.

Policy DSL Currently, policies are written in Gallina, the language embedded in Coq. This is fine for a proof-of-concept, but not satisfactory for real use. We plan to develop a domain-specific policy language to make it easier to write Tagged C policies.

References

1. Anderson, J.P.: Computer Security Technology Planning Study. Tech. rep., U.S. Air Force Electronic Systems Division (10 1972)
2. Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. In: Jajodia, S., Lopez, J. (eds.) Computer Security - ESORICS 2008. pp. 333–348. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
3. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.: A few billion lines of code later: Using static analysis to find bugs in the real world. Commun. ACM **53**(2), 66–75 (feb 2010). https://doi.org/10.1145/1646353.1646374, https://doi.org/10.1145/1646353.1646374
4. Chhak, C., Tolmach, A., Anderson, S.: Towards formally verified compilation of tag-based policy enforcement. In: Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 137–151. CPP 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3437992.3439929, https://doi.org/10.1145/3437992.3439929
5. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7), 504–513 (jul 1977). https://doi.org/10.1145/359636.359712, https://doi.org/10.1145/359636.359712

6. Dhawan, U., Vasilakis, N., Rubin, R., Chiricescu, S., Smith, J.M., Knight Jr., T.F., Pierce, B.C., DeHon, A.: PUMP: A Programmable Unit for Metadata Processing. In: Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy. p. 8:1–8:8. HASP '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2611765.2611773, http://doi.acm.org/10.1145/2611765.2611773

7. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy. pp. 11–20. IEEE Computer Society (1982), http://dblp.uni-trier.de/db/conf/sp/sp1982.html#GoguenM82a

8. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (jul 2009). https://doi.org/10.1145/1538788.1538814, https://doi.org/10.1145/1538788.1538814

9. Memarian, K., Gomes, V.B.F., Davis, B., Kell, S., Richardson, A., Watson, R.N.M., Sewell, P.: Exploring c semantics and pointer provenance. Proc. ACM Program. Lang. **3**(POPL) (jan 2019). https://doi.org/10.1145/3290380, https://doi.org/10.1145/3290380

10. Memarian, K., Matthiesen, J., Lingard, J., Nienhuis, K., Chisnall, D., Watson, R.N.M., Sewell, P.: Into the depths of c: Elaborating the de facto standards. SIGPLAN Not. **51**(6), 1–15 (jun 2016). https://doi.org/10.1145/2980983.2908081, https://doi.org/10.1145/2980983.2908081

11. Munoz, D.: After all these years, the world is still powered by c programming, https://www.toptal.com/c/after-all-these-years-the-world-is-still-powered-by-c-programming

12. Overflow, S.: 2022 stack overflow annual developer survey (2022), https://survey.stackoverflow.co/2022/

13. Skorstengaard, L., Devriese, D., Birkedal, L.: StkTokens: Enforcing Well-bracketed Control Flow and Stack Encapsulation using Linear Capabilities. Proceedings of the ACM on Programming Languages **3**(POPL), 1–28 (2019)

14. Wesley Filardo, N., Gutstein, B.F., Woodruff, J., Ainsworth, S., Paul-Trifu, L., Davis, B., Xia, H., Tomasz Napierala, E., Richardson, A., Baldwin, J., Chisnall, D., Clarke, J., Gudka, K., Joannou, A., Theodore Markettos, A., Mazzinghi, A., Norton, R.M., Roe, M., Sewell, P., Son, S., Jones, T.M., Moore, S.W., Neumann, P.G., Watson, R.N.M.: Cornucopia: Temporal safety for cheri heaps. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 608–625 (2020). https://doi.org/10.1109/SP40000.2020.00098

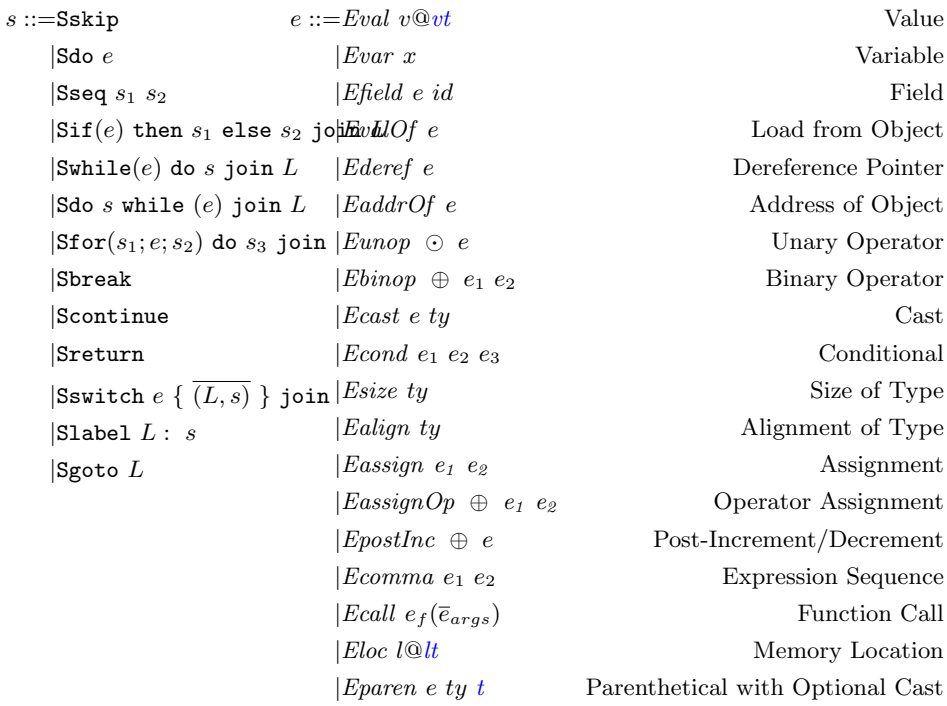

Fig. 11: Tagged C Abstract Syntax

A Syntax

B Continuations

$$
\begin{aligned}
k ::= & \mathit{Kemp} \\
& | \mathit{Kdo};\ k \\
& | \mathit{Kseq}\ s;\ k \\
& | \mathit{Kif}\ s_1\ s_2\ L;\ k \\
& | \mathit{KwhileTest}\ e\ s\ L;\ k \\
& | \mathit{KwhileLoop}\ e\ s\ L;\ k \\
& | \mathit{KdoWhileTest}\ e\ s\ L;\ k \\
& | \mathit{KdoWhileLoop}\ e\ s\ L;\ k \\
& | \mathit{Kfor}\ (e, s_2)\ s_3\ L;\ k \\
& | \mathit{KforPost}\ (e, s_2)\ s_3\ L;\ k
\end{aligned}
$$

C States

States can be of several kinds, denoted by their script prefix: a *general state* $\mathcal{S}(\dots)$, an *expression state* $\mathcal{E}(\dots)$, a *call state* $\mathcal{C}(\dots)$, or a *return state* $\mathcal{R}(\dots)$. Finally, the special state *failstop* ($\mathcal{F}(\dots)$) represents a tag failure, and carries the state that produced the failure. [Allison: to whatever degree you've figured out what is useful here by publication-time, we can tune this to be more specific.]

$$
\begin{aligned}
S ::= & \mathcal{S}\,(m \mid s \gg k@\mathcal{P}) \\
& | \mathcal{E}\,(m \mid e \gg k@\mathcal{P}) \\
& | \mathcal{C}\,(\mathcal{P} \mid m(le) \gg f'@f)\ \overline{\mathit{Eval}\ v@vt}k \\
& | \mathcal{R}\,(m \mid ge \gg le@\mathcal{P})\ \mathit{Eval}\ v@vtk \\
& | \mathcal{F}\,(S)
\end{aligned}
$$

D Initial State

Given a list xs of variable identifiers id and types ty, a program's initial memory is defined by iteratively allocating each one in memory and updating the global environment with its base address, bound, type, and a static identity tag. Let $|ty|$ be a function from types to their sizes in bytes. The memory is initialized $\mathbf{undef}@vt@\overline{lt}$ for some vt and $\overline{lt}$, unless given an initializer. Let m_0 and ge_0 be the initial (empty) memory and environment. The parameter b marks the start of the global region.

$$globals\ xs\ b = \begin{cases} (m_0, ge_0) & \text{if } xs = \varepsilon \\ (m[p \ldots p + |ty| \mapsto \mathbf{undef}@vt@\overline{lt}]_{|ty|}, & \text{if } xs = (id, ty) :: xs' \\ \ ge[id \mapsto (p, p + |ty|, ty, pt)]) & \text{and } pt, vt, \overline{lt} \leftarrow \mathbf{GlobalT}(\mathrm{G}_x, \mathrm{T}_{ty}) \\ & \text{where } (m, ge) = globals\ xs'\ (b + |ty|) \end{cases}$$

E Step Rules

E.1 Sequencing rules

$$\frac{}{\mathcal{S}\,(m \mid \texttt{Sdo}\ e \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\,(m \mid e \gg Kdo;\ k@\mathcal{P})}$$

$$\frac{}{\mathcal{E}\,(m \mid Eval\ v@vt \gg Kdo;\ k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m \mid \texttt{Sskip} \gg k@\mathcal{P})}$$

$$\frac{}{\mathcal{S}\,(m \mid \texttt{Sseq}\ s_1\ s_2 \gg k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m \mid s_1 \gg Kseq\ s_2;\ k@\mathcal{P})}$$

$$\frac{}{\mathcal{S}\,(m \mid \texttt{Sskip} \gg Kseq\ s;\ k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m \mid s \gg k@\mathcal{P})}$$

$$\frac{}{\mathcal{S}\,(m \mid \texttt{Scontinue} \gg Kseq\ s;\ k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m \mid \texttt{Scontinue} \gg k@\mathcal{P})}$$

$$\frac{}{\mathcal{S}\,(m \mid \texttt{Sbreak} \gg Kseq\ s;\ k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m \mid \texttt{Sbreak} \gg k@\mathcal{P})}$$

$$\frac{\mathcal{P}' \leftarrow \mathbf{LabelT}(\mathcal{P}, \mathrm{L}_L)}{\mathcal{S}\,(m \mid \texttt{Slabel}\ L :\ s \gg k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m \mid s \gg k@\mathcal{P}')}$$

E.2 Conditional rules

$$\frac{s = \texttt{Sif}(e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{join}\ L}{\mathcal{S}\,(m \mid s \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\,(m \mid e \gg Kif\ s_1\ s_2\ L;\ k@\mathcal{P})}$$

$$\frac{s' = \begin{cases} s_1 & \text{if } boolof(v) = \mathbf{t} \\ s_2 & \text{if } boolof(v) = \mathbf{f} \end{cases} \qquad \mathcal{P}' \leftarrow \mathbf{SplitT}(\mathcal{P}, vt, L)}{\mathcal{E}\,(m \mid Eval\ v@vt \gg Kif\ s_1\ s_2\ L;\ k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m \mid s' \gg k@\mathcal{P}')}$$

$$\frac{}{\mathcal{S}\left(m \mid \texttt{Sswitch}\ e\ \{\ \overline{(v, s)}\ \}\ \texttt{join}\ L \gg k@\mathcal{P}\right) \longrightarrow \mathcal{E}\left(m \mid e \gg Kswitch1\ \overline{(v, s)}\ L;\ k@\mathcal{P}\right)}$$

$$\frac{\mathit{select}\ v\ \overline{(v,s)} = s \qquad \mathcal{P}' \leftarrow \mathbf{SplitT}(\mathcal{P}, vt, L)}{\mathcal{E}\left(m \mid \mathit{Eval}\ v@vt \gg \mathit{Kswitch1}\ \overline{(v,s)}\ L;\ k@\mathcal{P}\right) \longrightarrow \mathcal{S}\left(m \mid s \gg \mathit{Kswitch2};\ k@\mathcal{P}'\right)}$$

$$\frac{s = \texttt{Sbreak} \vee s = \texttt{Sskip}}{\mathcal{S}\left(m \mid s \gg \mathit{Kswitch2};\ k@\mathcal{P}\right) \longrightarrow \mathcal{S}\left(m \mid \texttt{Sskip} \gg k@\mathcal{P}\right)}$$

$$\frac{}{\mathcal{S}\left(m \mid \texttt{Scontinue} \gg \mathit{Kswitch2};\ k@\mathcal{P}\right) \longrightarrow \mathcal{S}\left(m \mid \texttt{Scontinue} \gg k@\mathcal{P}\right)}$$

E.3 Loop rules

$$\frac{s = \texttt{Swhile}(e)\ \texttt{do}\ s'\ \texttt{join}\ L}{\mathcal{S}\left(m \mid s \gg k@\mathcal{P}\right) \longrightarrow \mathcal{E}\left(m \mid e \gg \mathit{KwhileTest}\ e\ s'\ L;\ k@\mathcal{P}\right)}$$

$$\frac{\mathit{boolof}(v) = \mathbf{t} \qquad k_1 = \mathit{KwhileTest}\ e\ s\ L;\ k \qquad k_2 = \mathit{KwhileLoop}\ e\ s\ L;\ k \qquad \mathcal{P}' \leftarrow \mathbf{SplitT}(\mathcal{P}, vt, L)}{\mathcal{E}\left(m \mid \mathit{Eval}\ v@vt \gg k_1@\mathcal{P}\right) \longrightarrow \mathcal{S}\left(m \mid s \gg k_2@\mathcal{P}'\right)}$$

$$\frac{\mathit{boolof}(v) = \mathbf{f}\ k = \mathit{KwhileTest}\ e\ s\ L;\ k'\ \mathcal{P}' \leftarrow \mathbf{SplitT}(\mathcal{P}, vt, L)}{\mathcal{E}\left(m \mid \mathit{Eval}\ v@vt \gg k@\mathcal{P}\right) \longrightarrow \mathcal{S}\left(m \mid \texttt{Sskip} \gg k'@\mathcal{P}'\right)}$$

$$\frac{s = \texttt{Sskip} \vee s = \texttt{Scontinue} \qquad k = \mathit{KwhileLoop}\ e\ s\ L;\ k'}{\mathcal{S}\left(m \mid s \gg k@\mathcal{P}\right) \longrightarrow \mathcal{S}\left(m \mid \texttt{Swhile}(e)\ \texttt{do}\ s\ \texttt{join}\ L \gg k'@\mathcal{P}\right)}$$

$$\frac{k = \mathit{KwhileLoop}\ e\ s\ L;\ k'}{\mathcal{S}\left(m \mid \texttt{Sbreak} \gg k@\mathcal{P}\right) \longrightarrow \mathcal{S}\left(m \mid \texttt{Sskip} \gg k'@\mathcal{P}\right)}$$

$$\frac{s = \texttt{Sdo}\ s'\ \texttt{while}\ (e)\ \texttt{join}\ L\ k' = \mathit{KdoWhileLoop}\ e\ s'\ L;\ k}{\mathcal{S}\left(m \mid s \gg k@\mathcal{P}\right) \longrightarrow \mathcal{S}\left(m \mid s' \gg k'@\mathcal{P}\right)}$$

$$\frac{k_1 = \mathit{KdoWhileLoop}\ e\ s\ L;\ k' \qquad k_2 = \mathit{KdoWhileTest}\ e\ s\ L;\ k}{\mathcal{S}\left(m \mid s' = \texttt{Sskip} \vee s' = \texttt{Scontinue} \gg k_1@\mathcal{P}\right) \longrightarrow \mathcal{E}\left(m \mid e \gg k_2@\mathcal{P}\right)}$$

$$\frac{\mathit{boolof}(v) = \mathbf{f}\ k = \mathit{KdoWhileTest}\ e\ s\ L;\ k'\ \mathcal{P}' \leftarrow \mathbf{SplitT}(\mathcal{P}, vt, L)}{\mathcal{S}\left(m \mid \mathit{Eval}\ v@vt \gg k@\mathcal{P}\right) \longrightarrow \mathcal{S}\left(m \mid \texttt{Sskip} \gg k'@\mathcal{P}'\right)}$$

$$\frac{\mathit{boolof}(v) = \mathbf{t}\ k = \mathit{KdoWhileTest}\ e\ s\ L;\ k' \qquad \mathcal{P}' \leftarrow \mathbf{SplitT}(\mathcal{P}, vt, L)}{\mathcal{S}\left(m \mid \mathit{Eval}\ v@vt \gg k@\mathcal{P}\right) \longrightarrow \mathcal{S}\left(m \mid \texttt{Sdo}\ s\ \texttt{while}\ (e)\ \texttt{join}\ L \gg k'@\mathcal{P}'\right)}$$

$$\frac{k = \mathit{KdoWhileLoop}\ e\ s\ L;\ k'}{\mathcal{S}\left(m \mid \texttt{Sbreak} \gg k@\mathcal{P}\right) \longrightarrow \mathcal{S}\left(m \mid \texttt{Sskip} \gg k'@\mathcal{P}\right)}$$

$$\frac{s = \texttt{Sfor}(s_1; e; s_2)\ \texttt{do}\ s_3\ \texttt{join}\ L \qquad s_1 \neq \texttt{Sskip}}{\mathcal{S}\left(m \mid s \gg k@\mathcal{P}\right) \longrightarrow \mathcal{S}\left(m \mid s_1 \gg \mathit{Kseq}\ \texttt{Sfor}(\texttt{Sskip}; e; s_2)\ \texttt{do}\ s_3\ \texttt{join}\ L;\ k@\mathcal{P}\right)}$$

$$\frac{s = \texttt{Sfor}(\texttt{Sskip}; e; s_2)\ \texttt{do}\ s_3\ \texttt{join}\ L}{\mathcal{S}\,(m \mid s \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\,(m \mid e \gg \mathit{Kfor}\ (e, s_2)\ s_3\ L;\ k@\mathcal{P})}$$

$$\frac{\mathit{boolof}\,(v) = \mathbf{f} \qquad \mathcal{P}' \leftarrow \mathbf{SplitT}(\mathcal{P}, vt, L)}{\mathcal{E}\,(m \mid \mathit{Eval}\ v@vt \gg \mathit{Kfor}\ (e, s_2)\ s_3\ L;\ k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m \mid \texttt{Sskip} \gg k@\mathcal{P})}$$

$$\frac{k = \mathit{Kfor}\ (e, s_2)\ s_3\ L;\ k'\ \mathit{boolof}\,(v) = \mathbf{t}\ \mathcal{P}' \leftarrow \mathbf{SplitT}(\mathcal{P}, vt, L)}{\mathcal{E}\,(m \mid \mathit{Eval}\ v@vt \gg k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m \mid s_3 \gg k@\mathcal{P})}$$

$$\frac{k = \mathit{Kfor}\ (e, s_2)\ s_3\ L; \qquad s = \texttt{Sskip} \vee s = \texttt{Scontinue}}{\mathcal{S}\,(m \mid s \gg k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m \mid \texttt{Sfor}(\texttt{Sskip}; e; s_2)\ \texttt{do}\ s_3\ \texttt{join}\ L \gg \mathit{KforPost}\ (e, s_2)\ s_3\ L;\ k@\mathcal{P})}$$

$$\frac{k = \mathit{Kfor}\ (e, s_1)\ s_2\ L;\ k'}{\mathcal{S}\,(m \mid \texttt{Sbreak} \gg k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m \mid \texttt{Sskip} \gg k'@\mathcal{P})}$$

$$\frac{k = \mathit{KforPost}\ (e, s_2)\ s_3\ L;\ k'}{\mathcal{S}\,(m \mid \texttt{Sskip} \gg k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m \mid \texttt{Sfor}(\texttt{Sskip}; e; s_2)\ \texttt{do}\ s_3\ \texttt{join}\ L \gg k@\mathcal{P})}$$

E.4 Contexts

Our expression semantics are contextual. A context *ctx* is a function from an expression to an expression and a tag. We identify a valid context using the *context* relation over a "kind" (left-hand or right-hand, LH or RH), and an expression.

$$
\begin{array}{lll}
\textit{context}\ k\ C[e] ::= & & \\
\quad | \textit{context}\ k\ \lambda e.e & & \\
\quad | \textit{context}\ \textsc{lh}\ \lambda e.\textit{Ederef}\ C[e] & & \text{where } \textit{context}\ \textsc{rh}\ C[e] \\
\quad | \textit{context}\ \textsc{lh}\ \lambda e.\textit{Efield}\ C[e]\ \textit{id} & & \text{where } \textit{context}\ \textsc{rh}\ C[e] \\
\quad | \textit{context}\ \textsc{rh}\ \lambda e.\textit{EvalOf}\ C[e] & & \text{where } \textit{context}\ \textsc{lh}\ C[e] \\
\quad | \textit{context}\ \textsc{rh}\ \lambda e.\textit{EaddrOf}\ C[e] & & \text{where } \textit{context}\ \textsc{lh}\ C[e] \\
\quad | \textit{context}\ \textsc{rh}\ \lambda e.\textit{Eunop}\ \odot\ C[e] & & \text{where } \textit{context}\ \textsc{rh}\ C[e] \\
\quad | \textit{context}\ \textsc{rh}\ \lambda e.\textit{Ebinop}\ \oplus\ C[e_1]\ e_2 & & \text{where } \textit{context}\ \textsc{rh}\ C[e_1] \\
\quad | \textit{context}\ \textsc{rh}\ \lambda e.\textit{Ebinop}\ \oplus\ e_1\ C[e_2] & & \text{where } \textit{context}\ \textsc{rh}\ C[e_2] \\
\quad | \textit{context}\ \textsc{rh}\ \lambda e.\textit{Ecast}\ C[e]\ \textit{ty} & & \text{where } \textit{context}\ \textsc{rh}\ C[e] \\
\quad | \textit{context}\ \textsc{rh}\ \lambda e.\textit{EseqAnd}\ C[e_1]\ e_2 & & \text{where } \textit{context}\ \textsc{rh}\ C[e_1] \\
\quad | \textit{context}\ \textsc{rh}\ \lambda e.\textit{EseqOr}\ C[e_1]\ e_2 & & \text{where } \textit{context}\ \textsc{rh}\ C[e_1] \\
\quad | \textit{context}\ \textsc{rh}\ \lambda e.\textit{Econd}\ C[e_1]\ e_2\ e_3 & & \text{where } \textit{context}\ \textsc{rh}\ C[e_1] \\
\quad | \textit{context}\ \textsc{rh}\ \lambda e.\textit{Eassign}\ C[e_1]\ e_2 & & \text{where } \textit{context}\ \textsc{lh}\ C[e_1] \\
\quad | \textit{context}\ \textsc{rh}\ \lambda e.\textit{Eassign}\ e_1\ C[e_2] & & \text{where } \textit{context}\ \textsc{rh}\ C[e_2] \\
\quad | \textit{context}\ \textsc{rh}\ \lambda e.\textit{EassignOp}\ \oplus\ C[e_1]\ e_2 & & \text{where } \textit{context}\ \textsc{lh}\ C[e_1] \\
\quad | \textit{context}\ \textsc{rh}\ \lambda e.\textit{EassignOp}\ \oplus\ e_1\ C[e_2] & & \text{where } \textit{context}\ \textsc{rh}\ C[e_2] \\
\quad | \textit{context}\ \textsc{rh}\ \lambda e.\textit{EpostInc}\ \oplus\ C[e] & & \text{where } \textit{context}\ \textsc{lh}\ C[e] \\
\quad | \textit{context}\ \textsc{rh}\ \lambda e.\textit{Ecall}\ C[e_1]\ (\overline{e_2}) & & \text{where } \textit{context}\ \textsc{rh}\ C[e_1] \\
\quad | \textit{context}\ \textsc{rh}\ \lambda e.\textit{Ecall}\ e_1(C[\overline{e_2}]) & & \text{where } \textit{context}\ \textsc{rh}\ C[e]\ \text{ for } e \in \overline{e_2} \\
\quad | \textit{context}\ \textsc{rh}\ \lambda e.\textit{Ecomma}\ C[e_1]\ e_2 & & \text{where } \textit{context}\ \textsc{rh}\ C[e_1] \\
\quad | \textit{context}\ \textsc{rh}\ \lambda e.\textit{Eparen}\ C[e]\ \textit{ty} & & \text{where } \textit{context}\ \textsc{rh}\ C[e] \\
\quad | \textit{context}\ \textsc{rh}\ \lambda e.\textit{Eparen}\ C[e]\ \textit{ty}\ t & & \text{where } \textit{context}\ \textsc{rh}\ C[e]
\end{array}
$$

Next, we define a notion of expression reduction. A left-hand reduction relates an expression to an expression. A right-hand reduction relates a triple of PC Tag, memory, and expression to another such triple.

$$\frac{\textit{context}\ \textsc{lh}\ C[e] \qquad e \Rightarrow_{\textsc{lh}} e'}{\mathcal{E}\,(m \mid C[e] \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\,(m \mid C[e] \gg k@\mathcal{P})}$$

$$\frac{\textit{context}\ \textsc{rh}\ C[e] \qquad (\mathcal{P}, m, e) \Rightarrow_{\textsc{rh}} (\mathcal{P}', m', e')}{\mathcal{E}\,(m \mid C[e] \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\,(m' \mid C[e] \gg k@\mathcal{P}')}$$

E.5 Expression Rules

$$\frac{le[id] = (l, \_, pt, ty)}{\textit{Evar}\ id \Rightarrow_{\textsc{lh}} \textit{Eloc}\ l@pt}$$

$$\frac{le[id] = \bot \quad ge[id] = \textsc{var}(l, \_, pt, ty)}{\textit{Evar}\ id \Rightarrow_{\textsc{lh}} \textit{Eloc}\ l@pt}$$

$$\frac{le[id] = \bot \;\; ge[id] = \textsc{Var}(f, pt)}{Evar\ id \Rightarrow_{\textsc{LH}} Efloc\ l@pt}$$

$$\frac{}{(\mathcal{P}, m, Ederef\ (Eval\ v@vt)) \Rightarrow_{\textsc{RH}} (\mathcal{P}, m, Eloc\ (to\_ptr\ v)@vt)}$$

$$\frac{ty = TStruct\ id \vee ty = TUnion\ id\ offset\ id\ fld = \delta \;\; pt' \leftarrow \mathbf{FieldT}(pt, \mathrm{T}_{ty}, \mathrm{G}_x)}{Efield\ (Eval\ p@pt : ty)\ fld) \Rightarrow_{\textsc{LH}} Eloc\ (p + \delta)@pt'}$$

$$\frac{m[l]_{|ty|} = v@vt@\overline{lt} \qquad vt' \leftarrow \mathbf{LoadT}(\mathcal{P}, pt, vt, \overline{lt})}{(\mathcal{P}, m, EvalOf\ (Eloc\ l@pt) : ty) \Rightarrow_{\textsc{RH}} (\mathcal{P}, m, Eval\ v@vt')}$$

$$\frac{}{(\mathcal{P}, m, EaddrOf\ (Eloc\ p@pt)) \Rightarrow_{\textsc{RH}} (\mathcal{P}, m, Eval\ p@pt)}$$

$$\frac{\langle\odot\rangle\ v = v' \qquad vt \leftarrow \mathbf{UnopT}(\odot, \mathcal{P}, vt)}{(\mathcal{P}, m, Eunop\ \odot\ (Eval\ v@vt)) \Rightarrow_{\textsc{RH}} (\mathcal{P}, m, Eval\ v'@vt')}$$

$$\frac{v_1\ \langle\oplus\rangle\ v_2 = v' \;\; vt' \leftarrow \mathbf{BinopT}(\oplus, \mathcal{P}, vt_1, vt_2) \quad e = Ebinop\ \oplus\ (Eval\ v_1@vt_1)\ (Eval\ v_2@vt_2)}{(\mathcal{P}, m, e) \Rightarrow_{\textsc{RH}} (\mathcal{P}, m, Eval\ v'@vt')}$$

$$\frac{\neg isptr(ty_1)\ \neg isptr(ty_2) \qquad pt \leftarrow \mathbf{IICastT}(\mathcal{P}, vt_1)}{(\mathcal{P}, m, Ecast\ (Eval\ v@vt : ty_1)\ ty_2) \Rightarrow_{\textsc{RH}} (\mathcal{P}, m, Eval\ v@vt' : ty_2)}$$

$$\frac{ty_1 = ptr\ ty_1' \qquad \neg isptr(ty_2) \quad m[v]_{|ty_1'|} = \_@vt@\overline{lt} \qquad vt \leftarrow \mathbf{PICastT}(\mathcal{P}, pt, vt, \overline{lt})}{(\mathcal{P}, m, Ecast\ (Eval\ v@pt : ty_1)\ ty_2) \Rightarrow_{\textsc{RH}} (\mathcal{P}, m, Eval\ v@vt' : ty_2)}$$

$$\frac{\neg isptr(ty_1) \qquad ty_2 = ptr\ ty_2' \quad m[v]_{|ty_2'|} = \_@vt_2@\overline{lt} \qquad pt \leftarrow \mathbf{IPCastT}(\mathcal{P}, vt_1, vt_2, \overline{lt})}{(\mathcal{P}, m, Ecast\ (Eval\ v@vt_1 : ty_1)\ ty_2) \Rightarrow_{\textsc{RH}} (\mathcal{P}, m, Eval\ v@pt : ty_2)}$$

$$\frac{ty_1 = ptr\ ty_1' \qquad ty_2 = ptr\ ty_2' \quad m[v]_{|ty_1'|} = m[v]_{|ty_2'|} = \_@vt@\overline{lt} \quad pt' \leftarrow \mathbf{PPCastT}(\mathcal{P}, pt, vt, \overline{lt})}{(\mathcal{P}, m, Ecast\ (Eval\ v@pt : ty_1)\ ty_2) \Rightarrow_{\textsc{RH}} (\mathcal{P}, m, Eval\ v@pt' : ty_2)}$$

$$\frac{boolof(v) = \mathbf{t} \qquad \mathcal{P}' \leftarrow \mathbf{ExprSplitT}(\mathcal{P}, vt)}{(\mathcal{P}, m, EseqAnd\ (Eval\ v@vt)\ e) \Rightarrow_{\textsc{RH}} (\mathcal{P}', m, Eparen\ e\ Tbool\ \mathcal{P})}$$

$$\frac{boolof(v) = \mathbf{f} \qquad \mathcal{P}' \leftarrow \mathbf{ExprSplitT}(\mathcal{P}, vt)}{(\mathcal{P}, m, EseqAnd\ (Eval\ v@vt)\ e) \Rightarrow_{\textsc{RH}} (\mathcal{P}', m, Eparen\ (Eval\ 0@vt')\ Tbool\ \mathcal{P})}$$

$$\frac{boolof(v) = \mathbf{t} \qquad \mathcal{P}' \leftarrow \mathbf{ExprSplitT}(\mathcal{P}, vt)}{(\mathcal{P}, m, EseqOr\ (Eval\ v@vt)\ e) \Rightarrow_{\textsc{RH}} (\mathcal{P}', m, Eparen\ (Eval\ 1@vt')\ Tbool\ \mathcal{P})}$$

$$\frac{boolof(v) = \mathbf{f} \qquad \mathcal{P}' \leftarrow \mathbf{ExprSplitT}(\mathcal{P}, vt)}{(\mathcal{P}, m, EseqOr\ (Eval\ v@vt)\ e) \Rightarrow_{\textsc{RH}} (\mathcal{P}', m, Eparen\ e\ Tbool\ \mathcal{P})}$$

$$\frac{e' = \begin{cases} e_1 & \text{if } \mathit{boolof}(v) = \mathbf{t} \\ e_2 & \text{if } \mathit{boolof}(v) = \mathbf{f} \end{cases} \qquad \mathcal{P}' \leftarrow \mathbf{ExprSplitT}(\mathcal{P}, vt)}{(\mathcal{P}, m, \mathit{Econd}\ (\mathit{Eval}\ v@vt)\ e_1\ e_2) \Rightarrow_{\mathrm{RH}} (\mathcal{P}', m, \mathit{Eparen}\ e'\ \ \mathcal{P}}$$

$$\frac{m[l]_{|ty|} = v_1@vt_1@\overline{lt} \qquad m' = m[l \mapsto v_2@vt'@\overline{lt}'] \qquad \mathcal{P}', vt', lt' \leftarrow \mathbf{StoreT}(\mathcal{P}, pt, vt, lt)}{(\mathcal{P}, m, \mathit{Eassign}\ (\mathit{Eloc}\ l@pt)\ (\mathit{Eval}\ v_2@vt_2)) \Rightarrow_{\mathrm{RH}} (\mathcal{P}', m', \mathit{Eval}\ v_2@vt_2)}$$

$$\frac{m[l]_{|ty|} = v_1@vt@\overline{lt} \oplus \in \{+, -, *, /, \%, <<, >>, \&, \hat{}, |\} \quad vt' \leftarrow \mathbf{LoadT}(\mathcal{P}, pt, vt, \overline{lt}) \quad e = \mathit{Eassign}\ (\mathit{Eloc}\ l@pt)\ (\mathit{Ebinop}\ \oplus\ (\mathit{Eval}\ v_1@vt')\ (\mathit{Eval}\ v_2@vt_2))}{(\mathcal{P}, m, \mathit{EassignOp}\ \oplus\ (\mathit{Eloc}\ l@pt)\ (\mathit{Eval}\ v_2@vt_2)) \Rightarrow_{\mathrm{RH}} (\mathcal{P}, m, e)}$$

$$\frac{m[l] = v@vt@\overline{lt} \quad \oplus \in \{+, -\} \qquad vt' \leftarrow \mathbf{LoadT}(\mathcal{P}, pt, vt, \overline{lt}) \quad e = \mathit{Ecomma}\ (\mathit{Eassign}\ (\mathit{Eloc}\ l@pt)\ (\mathit{Ebinop}\ \oplus\ \mathit{Eval}\ v@vt'\ 1@def))\ (\mathit{Eval}\ v@vt')}{(\mathcal{P}, m, \mathit{EpostInc}\ \oplus\ \mathit{Eloc}\ l@pt) \Rightarrow_{\mathrm{RH}} (\mathcal{P}, m, e)}$$

$$\frac{}{(\mathcal{P}, m, \mathit{Ecomma}\ (\mathit{Eval}\ v@vt)\ e) \Rightarrow_{\mathrm{RH}} (\mathcal{P}, m, e)}$$

$$\frac{\mathcal{P}'', vt' \leftarrow \mathbf{ExprJoinT}(\mathcal{P}, \mathcal{P}', vt)}{(\mathcal{P}, m, \mathit{Eparen}\ e\ ty\ \mathcal{P}') \Rightarrow_{\mathrm{RH}} (\mathcal{P}'', m, \mathit{Eval}\ v@vt')}$$

E.6 Call and Return Rules

In order to make a call, we need to reduce the function expression to an *Efloc* _@ value, an abstract location corresponding to a particular function. Then we can make the call.

$$\frac{\mathcal{P}' \leftarrow \mathbf{CallT}(\mathcal{P}, \mathrm{F}_f, \mathrm{F}_{f'})}{\mathcal{E}\ (m \mid C[\mathit{Ecall}\ \mathit{Efloc}\ f'@(\overline{v@vt})]\ ty \gg k@\mathcal{P}) \longrightarrow \mathcal{C}\ (m \mid f'(v@vt) \gg \mathit{Kcall}\ f\ C\ \mathcal{P};\ k@\mathcal{P}')}$$

When we make an internal call, we need to allocated space for locals and arguments using the helper function *frame*.

$$\mathit{frame}\ xs\ as\ m = \begin{cases} (m''[p \mapsto \mathbf{undef}@vt@\overline{lt}]_{|ty|}, & \text{if } xs = (id, ty) :: xs' \\ le'[id \mapsto (p, p + |ty|, ty, pt)]) & \text{where } (m', p) \leftarrow \mathit{stack\_alloc}\ |ty|\ m, \\ & \mathcal{P}', pt, vt, \overline{lt} \leftarrow \mathbf{LocalT}(\mathcal{P}, \mathrm{T}_{ty}), \\ & \text{and } (m'', le') = \mathit{frame}\ xs'\ as\ m' \\ (m''[p \mapsto v@vt'@\overline{lt}]_{|ty|}, & \text{if } as = (id, ty, v@vt) :: as' \text{ and } xs = \varepsilon \\ le'[id \mapsto (p, p + |ty|, ty, pt)]) & \text{where } (m', p) \leftarrow \mathit{stack\_alloc}\ |ty|\ m, \\ & \mathcal{P}', pt, vt', \overline{lt} \leftarrow \mathbf{ArgT}(\mathcal{P}, vt, \mathrm{A}_{f,x}, \mathrm{T}_{ty}), \\ & \text{and } (m'', le') = \mathit{frame}\ xs'\ as\ m' \\ (m, \lambda x.\bot) & \text{if } xs = \varepsilon \text{ and } as = \varepsilon \end{cases}$$

$$\frac{def(f) = INT(xs, as, s)\ m', le' = frame\ xs\ (zip\ as\ args)\ m\ le}{\mathcal{C}\,(m \mid f(args) \gg k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m' \mid s \gg k@\mathcal{P})\,/le'}$$

On the other hand, when we make an external call, we step directly to a return state with some value being returned and an updated memory. [TODO: talk more about how the tag policy applies in external functions, what they can and can't do with tags.]

$$\frac{def(f) = EXT(spec)\ \mathcal{P}' \leftarrow \mathbf{ExtCallT}(\mathcal{P}, \mathrm{F}_f, \mathrm{F}_{f'}, \overline{vt})\ \mathcal{P}'', m', (v@vt) = spec\ \mathcal{P}'\ args\ m}{\mathcal{C}\,(m \mid f(args) \gg k@\mathcal{P}) \longrightarrow \mathcal{R}\,(m' \mid v@vt \gg k@\mathcal{P}'')}$$

Special external functions, such as malloc, just get their own rules.

$$\frac{\begin{array}{c}\mathcal{P}', pt, vt, \overline{lt} \leftarrow \mathbf{MallocT}(\mathcal{P}, \mathrm{F}_f, \mathrm{F}_{f'}, vt)\ \ m', p \leftarrow heap\_alloc\ size\ m \\ m'' = m'\,[p + i \mapsto (\mathbf{undef}, vt, lt)]_{size}\end{array}}{\mathcal{C}\,(m \mid malloc((size@t)) \gg k@\mathcal{P}) \longrightarrow \mathcal{R}\,(m'' \mid Eval\ p@pt \gg k@\mathcal{P}')}$$

And finally, we have the return rules.

$$\frac{k = Kcall\ le'\ ctx\ \mathcal{P}_{CLR}\ k' \qquad \mathcal{P}', vt' \leftarrow \mathbf{RetT}(\mathcal{P}_{CLE}, \mathcal{P}_{CLR}, vt, \mathrm{F}_f)}{\mathcal{R}\,(m \mid Eval\ v@vt \gg k@\mathcal{P}_{CLE}) \longrightarrow \mathcal{E}\,(m \mid ctx[Eval\ v@vt'] \gg k'@\mathcal{P}')\,/le'}$$

$$\frac{dealloc\ m\ \mathcal{P} = (\mathcal{P}', m')}{\mathcal{E}\,(m \mid Eval\ v@vt \gg Kreturn;\ k@\mathcal{P}) \longrightarrow \mathcal{R}\,(m \mid Eval\ v@vt \gg k@\mathcal{P}')}$$

$$\frac{dealloc\ m\ \mathcal{P} = (\mathcal{P}', m')}{\mathcal{S}\,(m \mid \texttt{Sreturn}\ \gg k@\mathcal{P}) \longrightarrow \mathcal{R}\,(m' \mid Eval\ \mathbf{undef}@def \gg k@\mathcal{P}')}$$

F Moved from Intro

[SNA: I'm organizing our diss tracks into paragraphs that we can cut or move as needed]

Why Dynamic? Unfortunately, it is not always possible to fully secure C code before run-time. Ideally, bugs would be quickly identified and then fixed promptly. That is not always possible for a variety of reasons: bugs may escape detection, require significant effort to diagnose, or be impractical to fix. There are many techniques for finding bugs, but there is a shared stumbling block: C is not well defined. We cannot always agree on when something is a bug in C, especially code using Undefined Behaviors (UB) [**?**]. Confusion around expected behavior is no small problem. There are 191 undefined behaviors and 52 unspecified behaviors in the C99 specification [**?**]. Sometimes these behaviors are benign and skillfully used by the developer, other times they are unintended and highly dangerous. Unfortunately the distinction between the two is easily lost. Discerning expert code review is considered best practice, although it is rarely perfectly successful [] even if an expert is available at all. Even when there is both consensus and detection of a bug[APT: ??] AN: we can find it at and we can agree its a problem that should be fixed, changing the code may not be possible because it is in proprietary 3rd party libraries and drivers, or because regulations prohibit changes [3].

[APT: last clause is mysterious] AN: for example FDA approval used to forbid patching because you'd have to go through recertification. So healthcare wouldn't patch. SNA pointed out the coverity paper comments on this as a reason for bugs not getting fixed

Why C-Level? Tag-based enforcement in general has a significant body of work at the assembly level, especially PIPE (Programmable Interlocks for Policy Enforcement) []. However, even at the assembly-level these systems need the compiler to be in the trusted computing base (TCB), as many policies require knowledge of source-level constructs, even ones that do not depend on detailed knowledge of the program's behavior [cite Nick and Andre; anyone else?]. Moving policy-definition to the source level therefore does not expand the TCB and allows C developers to reason about policies in terms of the language that they program in regularly.

Notations Values are ranged over by v, variable identifiers by x, and function identifiers by f. Tags use a number of metavariables: t ranges over all tags, while we will use vt to refer to the tags associated with values, pt for tags on pointer values and memory-location expressions, lt for tags associated with memory locations themselves, nt for "name tags" automatically derived from identifiers, $\mathcal{P}$ for the global "program counter tag" or PC Tag. An *atom* is a pair of a value and a tag, *Eval* $v@vt$; the @ symbol should be read as a pair in general, and is used when the second object in the pair is a tag. Expressions are ranged over by e, statements by s, and continuations by k. The continuations are defined in appendix B, and step rules in appendix E.

A memory is an array of bytes, where each byte is part of an atom. Each byte is also associated with a "location tag" lt. When a contiguous region of s bytes starting at location l comprise an atom $v@vt$, and their locations tags comprise the list $\overline{lt}$, we write $m[l]_s = v@vt@\overline{lt}$. Likewise, $m[l \ldots l+s \mapsto v@vt@\overline{lt}]_s$ denotes storing that many bytes. Visually, we will represent whole atoms in memory as condensed boxes, with their location tags separate.