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 [19]. 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. overprovisioning access rights).

[TODO: Fill out more, esp. with relevant cituations.] Although static analyses can detect and mitigate many C insecurities, the last line of defense against undetected or unfixable vulnerabilities is runtime enforcement of *security policies* using a reference monitor [1]. A large class of useful policies can be specified as *data-flow* constraints based on *metadata tags*, which attach to the underlying data to carry information like type, provenance, ownership, or security classification. The policy takes the form of a set of rules that check and update the metadata during execution; if a rule violation is encountered, the program failstops. The class of policies that can be implemented in this way includes fine-grained memory safety, secure information flow, and mandatory access control, all of which are important security requirements from the literature.

Tag-based policies can be enforced either in software or—more promisingly—with the aid of hardware extensions such as ARM MTE [2], STAR [15], and PIPE [10,4,5]. PIPE (Processor Interlocks for Policy Enforcement),[1] which has been the specific motivator for our work, is particularly flexible: it supports arbitrary software-defined flow rules over large (word-sized) tags with arbitrary structure, which enables finer-grained policies and the composition of multiple policies together. This flexibility is useful because security needs may differ radically between codebases, and even within a codebase. A conservative, one-size-fits-all policy might be too strong, causing failstops even during normal execution. Sensitive code might call for specialized protection that doesn't apply elsewhere.

But PIPE policies can be difficult for a C engineer to write: their tags and rules are defined in terms of individual machine instructions and ISA-level concepts, and in practice are based on reverse engineering the behavior of specific compilers. Moreover, some security policies can only be expressed in terms of high-level code features that are not preserved at machine level, such as function arguments, structured types, and structured control flow.

To address these problems, we introduce a *source-level* specification framework called *Tagged C*, which allows engineers to describe policies in terms of familiar C-level concepts, with tags attached to C functions, variables and data values, and rules triggered at *control points* that correspond to significant execution events, such as function calls, expression evaluation, and pointer-based memory accesses. Control points resemble "advice points" in aspect-oriented programming, but narrowly focused on the manipulation of tags. In previous work, Chhak et al. [7] outlined such a framework for a toy source language and showed how high-level policies could be compiled to ISA-level policies and enforced using PIPE-like hardware. We extend this approach to handle the full, real C language, by giving a detailed design for the necessary control points and showing how they are integrated into C's dynamic semantics. Although motivated by PIPE, Tagged C is not tied to any particular enforcement mechanism, and indeed we currently implement it using a modified C interpreter rather than via compilation. We validate the design of Tagged C by using it to specify a range of interesting security policies, including compartmentalization, memory protection, and secure information flow.

Formally, Tagged C is defined as a variant C semantics that instruments ordinary execution with control points. At each control point, a user-defined set of tag rules is consulted to propagate tags and potentially cause execution to failstop. In the limiting case where no tag rules are defined, the semantics are similar to those of ordinary C, except that the memory model is very concrete (pointers and integers are essentially the same thing).[APT: Maybe explain why here, or forward ref.] We build our semantics on top of the CompCert C semantics, which are formalized as part of the CompCert verified compiler [16]. We provide a reference interpreter also based on that of CompCert, for use in pro-

[1] Variants of PIPE have also been called PUMP [11,10], SDMP [8,20]) or CoreGuard [13].

totyping and executing policies. Tag rules are written directly in Coq as Gallina functions.

Contributions In summary we offer the following contributions:

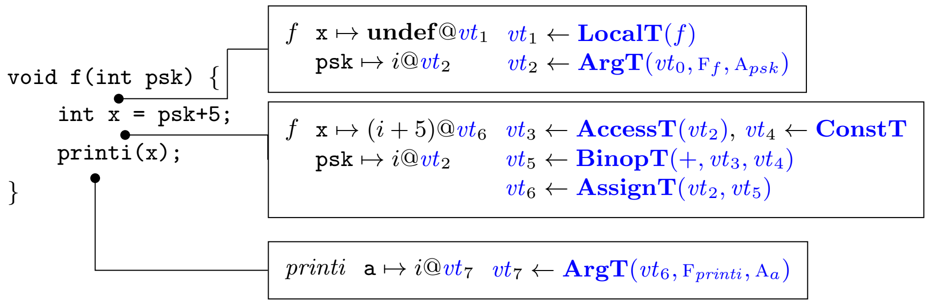
- The design of a comprehensive set of *control points* at which the C language interfaces with a tag-based policy
- Tagged C policies implementing (1) compartmentalization, (2) a realistic, permissive memory model from the literature (PVI), and (3) Secure Information Flow (SIF)
- A full formal semantic 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 Metadata Tags and Policies, by Example

Consider a straightforward security requirement: "do not leak high-security values to low-security channels." In the example in Fig. 1, assume that `psk` is a high-security value and `printi` prints to a low-security channel, so this code indirectly performs a leak via the local variable `x`. We now explain how we would use Tagged C to define a security policy that would prevent this leak. [APT: A whole explanation of the basic ideas is missing here: values carry tags, tags are checked at control points, rules are to be instantiated below, etc., etc.] [APT: And I think that you need first to explain informally how you want tag checking to work, along the lines of "We have two tags, H and L; the value of psk gets tagged H by virtue of a specific rule on f; the tag is propagated through the arithmetic by a general rule, and is caught at the printi by virtue of a specific rule on printi." The other control points are mainly distractions, especially AccessT and AssignT; perhaps it is best to leave these out of these initial examples.] Figure 1 maps each of three points in the execution of `f` to a description of the program state at that point, with the input value and all tags treated symbolically. Let i be the value passed to `psk` and vt_0 the corresponding tag. [APT: Subscripts should be blue too]. 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 flow of those tags though tag rules.

[APT: Why is `x` described before `psk`??] The initial tag on `x`, vt_1, comes from the **LocalT** tag rule; all locals for a given function name (which is a parameter to the rule) get the same tag. [APT: Not at all obvious to me (or the reader) why `x` gets an initial value and tag at all. (Indeed, should it, since it cannot be accessed before it acquires a proper initial value?) Needs an explanation at



```
void f(int psk) {
    int x = psk+5;
    printi(x);
}
```

f $\mathtt{x} \mapsto \mathbf{undef}@vt_1$ $\quad vt_1 \leftarrow \mathbf{LocalT}(f)$

$\mathtt{psk} \mapsto i@vt_2$ $\quad vt_2 \leftarrow \mathbf{ArgT}(vt_0, \mathrm{F}_f, \mathrm{A}_{psk})$

f $\mathtt{x} \mapsto (i+5)@vt_6$ $\quad vt_3 \leftarrow \mathbf{AccessT}(vt_2),\ vt_4 \leftarrow \mathbf{ConstT}$

$\mathtt{psk} \mapsto i@vt_2$ $\quad vt_5 \leftarrow \mathbf{BinopT}(+, vt_3, vt_4)$

$vt_6 \leftarrow \mathbf{AssignT}(vt_2, vt_5)$

$printi$ $\mathtt{a} \mapsto i@vt_7$ $\quad vt_7 \leftarrow \mathbf{ArgT}(vt_6, \mathrm{F}_{printi}, \mathrm{A}_a)$

Fig. 1: Example 1

least, although this is muddying the waters.] The tag on `psk` comes from **ArgT**, which is parameterized by the tag on the argument value (vt_0) and the names of both the function and the argument. Next, `x` is assigned `psk+5`. Four tag rules are 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 kind of 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 a value derived from `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 does 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. [APT: Can we consistently use primed variables for outputs (and say so here)?] "Case" statements are sometimes abbreviated by assuming that any input that doesn't match a case causes a failstop.[APT: But that doesn't happen here, so why say it? You are not writing a reference manual!]

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 name 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 a memory-mapped device register that is outside of the program's control, 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*.) [APT: Again, describe the tag-based solution informally first.]

ConstT
$vt := \text{L}$

LocalT$(\mathcal{P}, \text{T}_{ty})$
$vt := \text{L}$

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

AssignT$(\mathcal{P}, vt_1, vt_2)$
$vt' := vt_2$

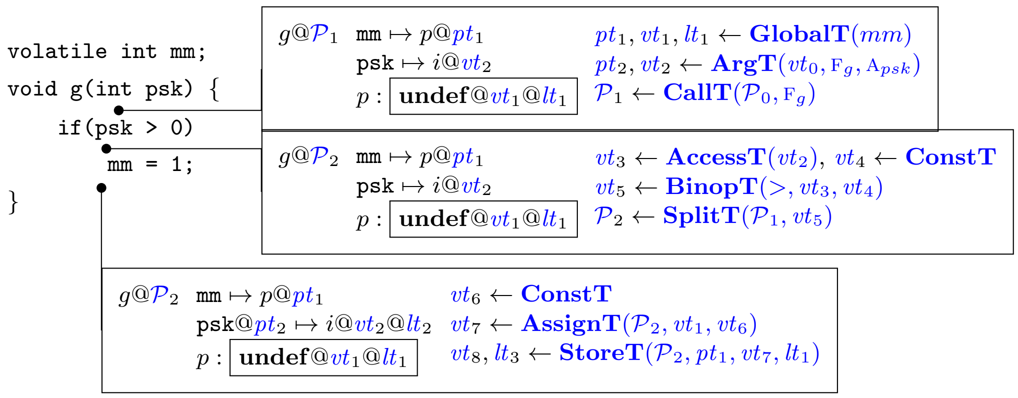
BinopT$(\oplus, \mathcal{P}, vt_1, vt_2)$
```
case  (vt₁, vt₂)  of
(L, L) ⇒ vt' := L
__     ⇒ vt' := H
```

$\tau ::= \text{H} | \text{L}$

ArgT$(\mathcal{P}, vt, \text{A}_{f,x}, \text{T}_{ty})$
```
case  (f, x, vt)  of
(printi, _, H)  ⇒ fail
(f, psk, _)     ⇒ vt' := H
__              ⇒ vt' := vt
```

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

Global variables like mm are always kept in memory, as are local arrays and structs. 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 denote by metavariable lt. The store maps their identifier to their address[APT: ???], 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 $\mathcal{P}$.



```
volatile int mm;
void g(int psk) {
    if(psk > 0)
        mm = 1;
}
```

$g@\mathcal{P}_1$ mm $\mapsto p@pt_1$; psk $\mapsto i@vt_2$; p : $\boxed{\textbf{undef}@vt_1@lt_1}$ $pt_1, vt_1, lt_1 \leftarrow$ **GlobalT**(mm); $pt_2, vt_2 \leftarrow$ **ArgT**$(vt_0, \text{F}_g, \text{A}_{psk})$; $\mathcal{P}_1 \leftarrow$ **CallT**$(\mathcal{P}_0, \text{F}_g)$

$g@\mathcal{P}_2$ mm $\mapsto p@pt_1$; psk $\mapsto i@vt_2$; p : $\boxed{\textbf{undef}@vt_1@lt_1}$ $vt_3 \leftarrow$ **AccessT**(vt_2), $vt_4 \leftarrow$ **ConstT**; $vt_5 \leftarrow$ **BinopT**$(>, vt_3, vt_4)$; $\mathcal{P}_2 \leftarrow$ **SplitT**$(\mathcal{P}_1, vt_5)$

$g@\mathcal{P}_2$ mm $\mapsto p@pt_1$; psk$@pt_2 \mapsto i@vt_2@lt_2$; p : $\boxed{\textbf{undef}@vt_1@lt_1}$ $vt_6 \leftarrow$ **ConstT**; $vt_7 \leftarrow$ **AssignT**$(\mathcal{P}_2, vt_1, vt_6)$; $vt_8, lt_3 \leftarrow$ **StoreT**$(\mathcal{P}_2, pt_1, vt_7, lt_1)$

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.

Because "don't print the password" is so similar to "don't leak the password," we can implement the latter with the same tag type, duplicating most of the rules.[APT: This gives the bogus impression that we expect to write lots of function-specific rules with their own special-purpose tags, etc.] We just need to deal with memory tags and the PC tag. 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($\mathcal{P}$, $vt$)
  case ($\mathcal{P}$, $vt$) of
    (L, L) ⇒ $\mathcal{P}'$ := L
    __     ⇒ $\mathcal{P}'$ := H
```

```
StoreT($\mathcal{P}$, $pt$, $vt$, $lt$)
  $lt'$ := $lt$
  case ($\mathcal{P}$, $vt$, $lt$) 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 Tagged C Language: Syntax and Semantics

[APT: Start boldly: "Tagged C is (almost) full ordinary C." Then list exceptions, wherever they come from (CompCert or us).]

Tagged C uses the full syntax of CompCert C [16], an elaboration[APT: that's a technical term in this context, which I don't think you mean] of the C standard into a formal operational semantics, with minimal modification[APT: from what?], with the exception of inline assembly.[APT: as I noted before, that's not part of the standard] Our semantics (given in full in the appendix) are a small-step reduction semantics which differ from CompCert C's in two key respects. 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. ([APT: This remark would be more useful in the intro.]) A control point consists of the name of a *tag rule* and the bindings of its inputs and outputs. Take, for example, the expression step reduction for binary operations, below. On the left, the "tagless" version of the rule reduces a binary operation on two input values to a single value by applying the operation. On the left, the Tagged C version adds tags to the operands and the result, with the result tag derived from the **BinopT** rule.

Rule Name	Inputs	Outputs	Control Points
AccessT	$\mathcal{P}, vt$	vt'	Variable Accesses
LoadT	$\mathcal{P}, pt, vt, \overline{lt}$	vt'	Memory Loads
AssignT	$\mathcal{P}, vt_1, vt_2$	$\mathcal{P}', vt'$	Variable Assignments
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	$\mathcal{P}, vt$	$\mathcal{P}', vt'$	Join points in expressions
SplitT	$\mathcal{P}, vt, \mathrm{L}_L$	$\mathcal{P}'$	Control-flow split points in statements
LabelT	$\mathcal{P}, \mathrm{L}_L$	$\mathcal{P}'$	Labels/arbitrary code points
CallT	$\mathcal{P}, pt$	$\mathcal{P}'$	Call
ArgT	$\mathcal{P}, vt, \mathrm{A}_{f,x}, \mathrm{T}_{ty}$	$\mathcal{P}', pt, vt', \overline{lt}$	Call
RetT	$\mathcal{P}_{CLE}, \mathcal{P}_{CLR}, vt$	$\mathcal{P}', vt'$	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 [APT: clarify this is invoked per local that comes into scope. and wha
DeallocT	$\mathcal{P}, \mathrm{T}_{ty}$	$\mathcal{P}', vt, \overline{lt}$	Return [APT: ditto]
ExtCallT	$\mathcal{P}, pt, \overline{vt}$	$\mathcal{P}'$	Call to linked code
MallocT	$\mathcal{P}, pt, vt$	$\mathcal{P}', pt, vt, \overline{lt}$	Call to `malloc`
FreeT	$\mathcal{P}, pt, vt$	$\mathcal{P}', vt', \overline{lt}$	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 and control points[APT: Ordering still seems sub-optimal. AccessT and AssignT are seldom interesting, so could move down the list. CallT, ArgT, RetT are quite important, so could move up.]

$$\frac{\begin{array}{c} v_1 \langle\oplus\rangle v_2 = v' \\ e = \mathit{Ebinop}\ \oplus\ v_1\ v_2 \end{array}}{(m, e) \Rightarrow_{\mathrm{RH}} (m, \mathit{Eval}\ v'@vt')} \qquad \frac{\begin{array}{c} v_1 \langle\oplus\rangle v_2 = v' \quad vt' \leftarrow \mathbf{BinopT}(\oplus, \mathcal{P}, vt_1, vt_2) \\ e = \mathit{Ebinop}\ \oplus\ (\mathit{Eval}\ v_1@vt_1)\ (\mathit{Eval}\ v_2@vt_2) \end{array}}{(\mathcal{P}, m, e) \Rightarrow_{\mathrm{RH}} (\mathcal{P}, m, \mathit{Eval}\ v'@vt')}$$

The tag rule itself is instantiated as a partial function; if a policy leaves a tag rule undefined on some inputs, then those inputs are considered to violate the policy, sending execution into a special failstop state. The names and signatures of the tag rules, and their corresponding control points, are listed in Table 1.[APT: It wouldn't hurt to talk briefly through these rules, especially when the connection between rule name and control point is not obvious. And make a point of saying: "and this is all there are or ever will be (we hope)"]

[APT: This para, which I rewrote to be a bit more positive, could go much earlier in the paper.] The choice of control points and their associations with tag rules, as well as the tag rules' signatures, form the essence of Tagged C's design. We have validated our design on the three classes of policies explored in this paper, and, outside of a few known limitation discussed in **??**[APT: e.g. substructural memory safety], we believe it is sufficiently expressive to describe most other flow-based policies, although of course further experience may reveal unexpected deficiencies.

The second major semantic distinction between Tagged C and CompCert C is that Tagged C has no memory-undefined behavior. CompCert C models memory as a collection of blocks of offsets, and treats all variables as having their own block. Tagged C instead follows Concrete C [] [Need something to cite!] in separating variables into public and private data.[APT: No, you will need to explain Concrete C in more detail] Public data (all heap data, globals, arrays, structs, and address-taken locals) share a single flat address space, where its behavior is implementation defined. [SNA: Technically right?[APT: but not very informative. You should simply describe how your memory model works. then we can add some language justifying why this is an interesting model in its own right.]] Private data (non-address-taken locals and parameters) have a separate store.

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}

Combining Policies Multiple policies can be enforced in parallel. If policy A has tag type τ_A and policy B has τ_B, then policy $A \times B$ should have tag type $\tau = \tau_A \times \tau_B$. Its tag rules should apply the rules of A to the left projection of all inputs and the rules of B to the right projection to generate the components of the new tag. If either side failstops, the entire rule should failstop.

[APT: Need to describe the limits on this idea, namely when policies are not completely orthogonal. For example, a compartmentalization policy is likely to overlap a generic memory safety policy.]

This process can be applied to any number of different policies, allowing for instance a combination of a baseline memory safety policy with several more targeted information-flow policies.

4 Example Policies

The heart of Tagged C is in its security policies.[APT: Not a good lead] Using the control points shown in Section 3, we will walk through three example policies: PVI memory safety, compartmentalization, and secure information flow.

[APT: Take a deep breath and remind yourself that the point of this paper is to explain and justify the control points and tag rules, NOT to explain these policies. Every section and every paragraph should be organized with that in mind.]

4.1 PVI Memory Safety

[APT: How about a quick informal summary of what this means intuitively? Then explain why a one-size-fits-all solution is not appropriate... In general, don't expect the reader to be very familiar with the idea of UB's or with the difference between official and defacto C standards.] 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 behaviors undefined, including some that are used in practice [18]. Memarian et al. have proposed two alternatives that define useful subsets of these behaviors [17]. We choose their "provenance via integer" (PVI) memory model as our example.

Variations of memory safety have been enforced in PIPE already, but usually[APT: ??] using an ad hoc memory model.[APT: Specific citations? How are they "ad hoc?"] 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. [APT: But Concrete C also gives this a definition! Need to explain what PVI still *prevents*.] Memarian et al.'s 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.[APT: Last two sentences seem to add nothing.] For a policy to enforce PVI, it should not failstop on any program that is defined in PVI.[APT: Isn't that tautological?] However, it should failstop if and when a program reaches UB.[APT: Somewhat confusing, because PVI turns some C standard UB's into defined behaviors.] For example, on many systems, pointers are (at least) 4-byte aligned, so the low-order two bits can be "stolen" to contain other data which is manipulated using bit-level shifting and masking operations. This idiom is often used when the programmer wants to associate a flag with the pointer, e.g. in some garbage collection implementations.[**?**] [APT: I don't think Cheney is particularly relevant.] These pointer manipulations are probably UBs in standard C (the standard is not completely clear on this point), but are explicitly defined in PVI. But violations such as buffer overflows in which a load, store, or free occurs outside of the proper range remain undefined in PVI and must failstop. In Fig. 4, we see an example of such a program, and a possible memory layout in which the overflow ends up overwriting the variable `y`.[APT: It is odd that the example doesn't illustrate the sort of bit-twiddling that PVI is intended to support! Actually, the best thing to see is how the tag rule instantiations would *change* to accommodate different policies (PVI, PNVI, something stricter...)]

```
void overrun() {
    int[2] x; int y;
    *(x+2) = 42;
}
```

$\mathtt{x} \mapsto 84@pt_1$	88	$\mathtt{y} \mapsto 92@pt_2$
undef @ vt_1	**undef** @ vt_1	**undef** @ vt_2
lt_1	lt_1	lt_2

84	88	92
undef @ vt_1	**undef** @ vt_1	42 @ vt_4
lt_1	lt_1	lt_4

$\mathcal{P}_1, vt_1, pt_1, lt_1 \leftarrow \textbf{LocalT}(\mathcal{P}_0)$

$\mathcal{P}_2, vt_2, pt_2, lt_2 \leftarrow \textbf{LocalT}(\mathcal{P}_1)$

$vt_3 \leftarrow \textbf{ConstT}$

$pt_2 \leftarrow \textbf{BinopT}(+, pt_1, vt_3)$

$vt_4 \leftarrow \textbf{AssignT}(\mathcal{P}_2, vt_2)$

$\mathcal{P}_3, vt_5, lt_3 \leftarrow \textbf{StoreT}(\mathcal{P}_3, pt_2, vt_5, lt_3)$

Fig. 4: Buffer Overflow in Action

Figure 4 shows a possible initial state of memory on entry to `overflow` and the state after the assignment to `*(x+2)`, assuming that the PC tag is $\mathcal{P}_0$ at the call.[APT: Where does the 84 come from? And this layout also assumes that `y` is in memory—but why is that?] The central tag rules are **BinopT** and **StoreT**: the former computes the tag on the pointer that will be used to access `x+2` in memory, and the latter decides if the store will be allowed.[APT: But actually, the LocalT rules are also crucial here. Also: I thought you were going to refactor things so that LocalT keeps its signature from Fig. 2, and there is a separate StoreT invocation for locals that live in memory. I guess maybe not.]

We can prevent overwrites 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 tag matches the target memory location tag. [APT: Need citations for this idea. At least our S&P paper. R&D credit `https://dl.acm.org/doi/10.1145/1321631.1321673`.] The default tag N indicates that there is currently no color. The rules for the PVI memory safety policy are given in Fig. 5. In this case, we will have $pt_1 = lt_1 = 0$ and $pt_2 = lt_2 = 1$. When the program tries to write to `x+2`, the **StoreT** rule compares $pt' = 0$ with lt_2, and failstops because they differ. [APT: Inconsistency in these rules: do LoadT and StoreT take a single lt or a vector of them?]

The cast rules[APT: what 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

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

$$\begin{array}{l}
\textbf{BinopT}(\oplus, \mathcal{P}, vt_1, vt_2) \\
\ \text{case} \ \ (vt_1,\ vt_2) \ \ \text{of} \\
\ (clr, N) \qquad\ \Rightarrow vt' := clr \\
\ (N, t) \qquad\quad\ \Rightarrow vt' := t \\
\ (clr_1, clr_2) \Rightarrow vt' := N
\end{array}$$

$$\begin{array}{l}
\textbf{UnopT}(\odot, \mathcal{P}, vt) \\
vt' := vt \\
\{\ vt\ \}
\end{array}$$

$$\begin{array}{l}
\textbf{LocalT}(\mathcal{P}, \tau_{ty}) \\
\mathcal{P}' := \mathcal{P} + 1;\ pt := \mathcal{P} \\
vt := N;\ lt := [\mathcal{P}]
\end{array}$$

$$\begin{array}{l}
\textbf{MallocT}(\mathcal{P}, pt, vt) \\
\mathcal{P}' := \mathcal{P} + 1;\ pt := \mathcal{P} \\
vt := N;\ lt := [\mathcal{P}]
\end{array}$$

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

$$\begin{array}{l}
\textbf{StoreT}(\mathcal{P}, pt, vt, lt) \\
\textbf{assert}\ \forall lt \in lt.pt = lt \\
\mathcal{P}' := \mathcal{P};\ vt' := vt_2;\ lt' := lt
\end{array}$$

Fig. 5: PVI Memory Safety Policy

and stores will be successful. Function pointers are an exception: Tagged C's underlying control-flow protections prevent them from being tampered with.

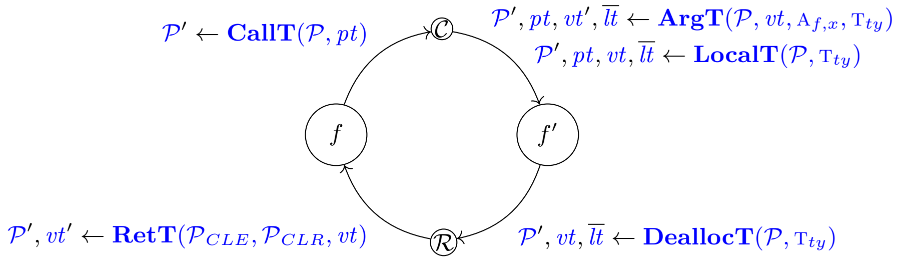
4.2 Compartmentalization

In a perfect world, all C programs would be memory safe. In reality, it is common for a codebase to contain undefined behavior that will not be fixed. Developers intentionally use low-level idioms that are UB [18], or the cost and risk of regressions may make it undesirable to fix bugs in older code [6].[APT: Do you want to say that we may not want to turn on the failstop protections of the previous section because that would cause too much code to break?]

A compartmentalization policy can isolate potentially risky code, such as code with intentional or unfixed UB, from safety-critical code, and enforce the principle of least privilege. Compartmentalization limits the possible damage from exploitation[APT: ??] to the containing compartment. And even in the absence of language-level errors, compartmentalization can usefully restrict how code in one compartment may interact with another. Ideally, each component should have only the *least privilege* necessary to complete its task. This popular defense concept can be implemented at many levels. It is often built into a system's fundamental design, like a web browser sandbox for untrusted javascript. For our use case, we consider a compartmentalization scheme being added to the system after development. A set of compartment identifiers are ranged over by C, and function and global identifiers are mapped to compartments by $comp(id)$.

Coarse-grained Protection The core of a compartmentalization policy is once again based on memory protection. In the simplest version, a function's stack frame and any heap-allocated regions are only accessible by functions within its compartment. The system keeps track of the active compartment using the PC tag.

In the Tagged C semantics, 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. 6. In the initial call step, **CallT** uses the tag on the function pointer (typically derived from F_f) to update the PC tag. Then, in the step from the call state, the function arguments are placed in the local store, tagged with the results of **ArgT**, and stack locals are allocated. Non-address-taken locals go in the store as well, tagged with **ConstT**, while locals that must be stack-allocated have memory allocated and tagged with the results of **LocalT**. [APT: Is that really how you want to do that? (See previous note re Fig 4.)] On return, locals are deallocated, their location tags are updated by **DeallocT**, and **RetT** updates both the PC tag and the tag on the returned value, with access to the original caller's PC tag.



$\mathcal{P}' \leftarrow \textbf{CallT}(\mathcal{P}, pt)$
$\mathbb{C}$
$\mathcal{P}', pt, vt', \overline{lt} \leftarrow \textbf{ArgT}(\mathcal{P}, vt, \text{A}_{f,x}, \text{T}_{ty})$
$\mathcal{P}', pt, vt, \overline{lt} \leftarrow \textbf{LocalT}(\mathcal{P}, \text{T}_{ty})$
f
f'
$\mathcal{P}', vt' \leftarrow \textbf{RetT}(\mathcal{P}_{CLE}, \mathcal{P}_{CLR}, vt)$
$\mathbb{R}$
$\mathcal{P}', vt, \overline{lt} \leftarrow \textbf{DeallocT}(\mathcal{P}, \text{T}_{ty})$

Fig. 6: Structure of a function call and return, where f calls f'.

In our compartmentalization policy (Fig. 7), we define a tag to be a compartment identifier or the default N tag. The PC tag always carries the compartment of the active function, kept up to date by the **CallT** and **RetT** rules.

$\tau ::= C | N$

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

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

MallocT$(\mathcal{P}, pt, vt)$
$\mathcal{P}' := \mathcal{P}; pt := N;$
$vt := N; lt := \mathcal{P}$

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

LocalT$(\mathcal{P}, \text{T}_{ty})$
$\mathcal{P}' := \mathcal{P}; pt := N;$
$vt := N; lt := \mathcal{P}$

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

Fig. 7: Compartmentalization Policy

[APT: Misc. complaints: Vectors on $\overline{lt}$ not explained. CLE and CLR not explained. Bracketed outputs in CallT and RetT left over from a previous version? Explain MallocT rule (or omit here). Everything in Fig. 7 should be in blue.]

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 [APT: It is not clear what new thing this section teaches us about control points and rules. If there is anything, be explicit about stating it; if not, perhaps the section does not pull its weight.]

The above policy works if our compartments only ever communicate by passing non-pointer values. In practice, this is far too restrictive: many library functions take pointers and operate on memory shared with the caller. [APT: rest of this paragarph has nothing to do with sharing memory per se, and could go up in the intro to the whole compartmentalization section.] 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. For example, the hostname needs to conform to an expected pattern[APT: ??? Need to introduce the example first!], such as in an enterprise network, to differentiate between different classes of computers (employee, server, contractor, etc). The standard library, in its own compartment, has helpful functions, provided the caller provides[APT: awk] the buffers from which to set or get the hostname.

[APT: The role of this example is not clear. Does it, for exmaple, contain both shared and unshared allocations in the sense of the scheme below? If so, tie that into the explanation. At the very least, talk through where you would expect to see failstopping occur.]

```
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 shared memory: *mandatory access control* (MAC) and *capabilities*.[APT: If you are going to say “literature” you need to provide citations!] MAC explicitly enumerates the access rights of each compartment, which can include giving two compartments access to the same memory. Capability systems treat pointers as 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 distinguish between memory objects that may be passed and those that must not be. 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 be used to specialize 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.[APT: This is not a pretty story, and should probably be told more apologetically (or at least with a pointer to the limitations section).]

The policy (Fig. 8) works by gluing together compartmentalization and memory safety. The PC tag carries both the current compartment color, for tagging unshared allocations, and the next free color, for tagging shared allocations. **MallocT** uses the function tag to determine which color to attach to the pointer and allocated region. During loads and stores, the location tag of the target address determines whether access is checked via the identity of the active compartment (for unshared allocations) or the validity of the pointer (for shared ones).

$\tau ::= N \mid C \mid clr \mid (C, clr)$

```
MallocT($\mathcal{P}, pt, vt$)
let $(C, clr) := \mathcal{P}$ in
case $f'$ of
malloc        $\Rightarrow \mathcal{P}' := \mathcal{P}; pt := C;$
                $vt := N; lt := C$
malloc_share  $\Rightarrow \mathcal{P}' := (C, clr + 1);$
                $pt := clr; vt := N';$
                $lt := clr$
```

```
StoreT($\mathcal{P}, pt, vt, lt$)
let $(C, clr) := \mathcal{P}$ in
assert $\exists lt'. \forall lt \in lt. lt = lt'$
case $lt'$ of
$C'$ $\Rightarrow$ assert $C = C'$
        $\mathcal{P}' := \mathcal{P}; vt' := vt_2; lt := C$
$clr$ $\Rightarrow$ assert $pt = clr$
        $\mathcal{P}' := \mathcal{P}; vt' := vt_2; lt := clr$
```

```
LoadT($\mathcal{P}, pt, vt, \overline{lt}$)
let $(C, clr) := \mathcal{P}$ in
assert $\exists lt'. \forall lt \in lt. lt = lt'$
case $lt'$ of
$C'$ $\Rightarrow$ assert $C = C'$
        $vt' := vt$
$clr$ $\Rightarrow$ assert $pt = clr$
        $vt' := vt$
```

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

Fig. 8: Compartmentalization with Shared Capabilities

[APT: read to here.]

4.3 Secure Information Flow

Our final example policy will be a more realistic version of our first: *secure information flow (SIF)* [9], which is part of a larger family of policies known as *information flow control* (IFC). [APT: In fact, one could argue that everything we do with metadata tags is really 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 first being sanitized:

```
void sanitize(char* in, char* out);
void sql_query(char* query, char* res);

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

  scanf("%19", name);
  sanitize(name, name_san);
  strncat(query, name, strlen(name));
  strncat(query, "\"",1);

  sql_query(query, res);
  printf(res);
}
```

This function sanitizes its input `name`, then appends the result to an appropriate SQL query, storing the query's result in `res`. But the programmer has accidentally used the unsanitized string `name` reather than `name_san`! This creates the opportunity for an SQL injection attack[?]: a user of this function could (presumably at the behest of an outside user) pass it a `name` containing "`Bobby"; drop table;`".

The fact that the input can be sanitized also makes this[APT: what? haven't described the policy yet] an *intransitive* policy: information may flow from `scanf` to `sanitize`, and from `sanitize` to `sql_query`, but not directly from `scanf` to `sql_query`.

Since we care about a single source[APT: ??], we can once again use a pair of H and L tags, although in this case we aim to prevent L-integrity data from flowing to H-integrity locations. We additionally need to carry significant information on the PC tag, so we define a third type of tag, written using the constructor PC, which carries (1) the current function identifier, (2) a natural number to record a count of tainted expression scopes, and (3) a stack of label identifiers to record the join points of tainted statement scopes. We will discuss (2) and (3) in detail below. Initially, the PC tag is $\text{PC}\ f\ 0\ \varepsilon$.

Selected tag rules for expressions are given in Fig. 9. We define two operators on tags: the "join" operator $\sqcap$ takes the higher of two security levels, and the "reduce" operator $|\cdot|$ converts a PC tag into a security level, H or L. The

$$\tau ::= \text{H} \mid \text{L} \mid \text{PC}\ f\ d\ Ls \quad d \in \mathbb{N},\ Ls \in \mathit{list}\ \mathit{id}$$

$$|t| \triangleq \begin{cases} \text{H} & \text{if } t = \text{H} \text{ or } t = \text{PC}\ f\ \mathit{0}\ \varepsilon \\ \text{L} & \text{otherwise} \end{cases}$$

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

```
**StoreT**$(\mathcal{P}, pt, vt, lt)$
let $\text{PC}\ f\ d\ Ls := \mathcal{P}$ in
$\mathcal{P}' := \mathcal{P}$
$lt' := lt$
case $f$ of
 scanf     $\Rightarrow vt' := \text{L}$
 sanitize  $\Rightarrow vt' := \text{H}$
 ______    $\Rightarrow vt' := \mathcal{P} \sqcap vt \sqcap pt$
```

```
**LoadT**$(\mathcal{P}, pt, vt, \overline{lt})$
let $\text{PC}\ f\ d\ Ls := \mathcal{P}$ in
 case $f$ of
 sql_query $\Rightarrow$ assert $vt \sqcap pt = \text{H}$
                  $vt' := vt$
 ______        $\Rightarrow vt' := vt$
```

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

Fig. 9: SIF Policy: Tags and Selected Rules

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

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

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

$$\begin{tikzcd}
 & & b_1 \arrow[dr] & & \\
{} \arrow[r] & S \arrow[ur] \arrow[dr] & & J \arrow[r] & {} \\
 & & b_2 \arrow[ur] & &
\end{tikzcd}$$

Fig. 10: Leaking via if statements

function `scanf` taints all of its writes by marking them L. [APT: I thought we'd agreed that this was an overly heavy-handed way to make `scanf` taint the input. However...] This extends to its return value in **RetT** as well (not shown).[APT: But it doesn't have a return value! And how would we express that it should taint its (second) argument?] Similarly, all outputs of `sanitize` are tagged L, so that when it copies H-tagged data into its output buffer, we consider those data safe.[APT: But again, how is this expressed in a policy? For examples, baybe better to avoid passing data back via args (although of course we should have a story for that).]

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 L value.[APT: Ditto on this being a heavy-handed and indirect scheme. Just want to say that the `query` is L.]

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. 10, 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 L, while `public2` is tagged H.[APT: This is confusing integrity and confidentiality.]

This is where the other components of the PC tag come into play. We perform a program transformation introducing the labels of all join points explicitly, so that in Fig. 10, `J` is an explicit label in the code, if it wasn't already.[APT: Clarify that this "transformation" is strictly internal/conceptual; maybe put forward pointer to para at end of this section (or more that here)?] We introduce an internal form of each conditional statement (if, switch, while, do-while, and for) that takes as an extra parameter the label of that conditional's join point. As seen in Fig. 11, the **SplitT** tag rule takes this label as an optional parameter and, if the conditional branches on a low value, pushes the join point to the stack. Then the **LabelT** rule checks if execution has reached a join point and if it has, removes it from the stack. A PC tag can only reduce to H if its stack is empty.

Branching expressions work similarly, except that the join point occurs at a different internal expression form: the parenthetical expression. For example, the expression `1 ? a : b` reduces to the parenthetical `(a)`.[APT: This requires a much deeper understanding of reduction semantics than anything else in the paper – can that be avoided?] The **ExprJoinT** rule applies when `a` is fully reduced and the semantics throw away the parentheses. Because expressions are nested, we only need to keep track of how deep execution has gone since the first time it branched on a L value, by incrementing and decrementing the depth d. [APT: This is obscure.] **ExprJoinT** also tags the result of the expression.[APT: This seems too trivial to note in the text. Except: what *is* the reason for joining with P in the paritcular rule shown?]

We assume a *termination-insensitive* setting [3], in which we allow an observer to glean information by the termination or non-termination of the program. 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.

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

```
**SplitT**$(\mathcal{P}, vt, \text{L}_L)$
case $\mathcal{P}, vt$ of
$\text{PC}\ f\ \ d\ \ Ls, \text{L} \Rightarrow \mathcal{P}' := \text{PC}\ f\ \ d\ \ (L :: Ls)$
$\_\_, \text{H} \qquad\qquad \Rightarrow \mathcal{P}' := \mathcal{P}$
```

```
**ExprSplitT**$(\mathcal{P}, vt)$
case $\mathcal{P}, vt$ of
$\text{PC}\ f\ \ d\ \ Ls, \text{L} \Rightarrow \mathcal{P}' := \text{PC}\ f\ \ (d + 1)\ \ Ls$
$\_\_, \text{H} \qquad\qquad \Rightarrow \mathcal{P}' := \mathcal{P}$
```

```
**LabelT**$(\mathcal{P}, \text{L}_L)$
case $\mathcal{P}$ of
$\text{PC}\ f\ \ d\ \ Ls \Rightarrow$ let $Ls' = pop\_all\ \text{L}_L\ \ Ls$ in
                 $\mathcal{P}' := \text{PC}\ f\ \ d\ \ Ls'$
$\_\_ \qquad\qquad \Rightarrow \mathcal{P}' := \mathcal{P}$
```

```
**ExprJoinT**$(\mathcal{P}, vt)$
$vt' := vt \sqcap \mathcal{P}$
case $\mathcal{P}$ of
$\text{PC}\ f\ \ (d + 1)\ \ Ls \Rightarrow \mathcal{P}' := \text{PC}\ f\ \ d\ \ Ls$
$\_\_ \qquad\qquad\qquad \Rightarrow \mathcal{P}' := \mathcal{P}$
```

Fig. 11: SIF Conditionals

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. [7] 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. 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 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. [APT: But perhaps note that tag-based policies can express some trace properties more easily than value-based policies could, essentially because tags can summarize the relevant parts of the trace history.]

[APT: Something about malloc tagging, and type-based tagging in general.]

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.

Beyond these, there is always the possibility that we have missed some control point that would enable interesting security policies outside of the areas we have considered. How hard would it be to add control points, or extend the ones that exist? That depends whether the extension in question would impact compilation strategies. For instance, we have avoided control points that allow updates to their operand tags, although this could be useful in enforcing policies that attempt to maintain the uniqueness of tags, because tag-based hardware is unlikely to support linear tags, so any such updates would need the compiler to generate extra instructions.

7 Related Work

Reference Monitors The concept of a reference monitor was first introduced fifty years ago [1] as 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 [14]).

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) [12] 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, STAR, 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 [23], stack safety [21], 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.[APT: A grot-esque coinage.]

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 [16], and three example policies. There are several significant next-steps.

Compilation An interpreter is all well and good[APT: too informal], 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[APT: too informal].

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. Technical Report ESD-TR-73-51, U.S. Air Force Electronic Systems Division (Oct 1972), `http://csrc.nist.gov/publications/history/ande72.pdf`
2. Armv8.5-a memory tagging extension white paper, `https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf`
3. 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)
4. Azevedo de Amorim, A., Collins, N., DeHon, A., Demange, D., Hritcu, C., Pichardie, D., Pierce, B.C., Pollack, R., Tolmach, A.: A verified information-flow architecture. Journal of Computer Security **24**(6), 689–734 (2016). `https://doi.org/10.3233/JCS-15784`, `http://dx.doi.org/10.3233/JCS-15784`
5. Azevedo de Amorim, A., Dénès, M., Giannarakis, N., Hritcu, C., Pierce, B.C., Spector-Zabusky, A., Tolmach, A.P.: Micro-policies: Formally verified, tag-based security monitors. In: 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015. pp. 813–830. IEEE Computer Society (2015). `https://doi.org/10.1109/SP.2015.55`, `http://dx.doi.org/10.1109/SP.2015.55`
6. 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`

7. 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
8. DeHon, A., Boling, E., Nikhil, R., Rad, D., Schwarz, J., Sharma, N., Stoy, J., Sullivan, G., Sutherland, A.: DOVER: A Metadata-Extended RISC-V. In: RISC-V Workshop (Jan 2016), http://riscv.org/wp-content/uploads/2016/01/Wed1430-dover_riscv_jan2016_v3.pdf, accompanying talk at http://youtu.be/r5dIS1kDars
9. 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
10. Dhawan, U., Hritcu, C., Rubin, R., Vasilakis, N., Chiricescu, S., Smith, J.M., Knight, Jr., T.F., Pierce, B.C., DeHon, A.: Architectural support for software-defined metadata processing. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 487–502. ASPLOS '15, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2694344.2694383, http://doi.acm.org/10.1145/2694344.2694383
11. 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
12. 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
13. Dover Microsystems: Coreguard overview, https://www.dovermicrosystems.com/solutions/coreguard/
14. 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
15. Gollapudi, R., Yuksek, G., Demicco, D., Cole, M., Kothari, G.N., Kulkarni, R.H., Zhang, X., Ghose, K., Prakash, A., Umrigar, Z.: Control flow and pointer integrity enforcement in a secure tagged architecture. In: 2023 IEEE Symposium on Security and Privacy (SP). pp. 2974–2989. IEEE Computer Society, Los Alamitos, CA, USA (may 2023). https://doi.org/10.1109/SP46215.2023.00102, https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00102
16. 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
17. 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

18. 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`
19. 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`
20. Roessler, N., DeHon, A.: Protecting the stack with metadata policies and tagged hardware. In: 2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA. pp. 478–495 (2018). `https://doi.org/10.1109/SP.2018.00066`, `https://doi.org/10.1109/SP.2018.00066`
21. 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)
22. Stack Overflow: 2022 stack overflow annual developer survey (2022), `https://survey.stackoverflow.co/2022/`
23. 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`

A Syntax

Tagged C has the full complement of typical C expressions (Fig. 12). $\mathit{Eval}\ v@vt$, $\mathit{Eloc}\ p@pt$, and $\mathit{Eparen}\ e\ ty$ are internal forms. A constant c in the concrete syntax is transformed into $\mathit{Eval}\ c@\mathbf{ConstT}$, and in general $\mathit{Eval}\ v@vt$ is a fully-reduced right-hand value. $\mathit{Eloc}\ p@pt$ is a fully-reduced left-hand value that represents the address of a variable. $\mathit{Eparen}\ e\ ty$ is the result of a conditional or shortcutting expression, with being a type annotation in case the result needs to be cast.

Some common C expressions are derived forms. An array index expression, $e_1[e_2]$ expands to $\mathit{Ederef}\ \mathit{Ebinop}\ +\ e_1\ e_2.Thepre-incrementexpression++e$ expands to $\mathit{Eassign}\ e\ \mathit{Ebinop}\ +\ e\ 1@\mathbf{ConstT}$, and likewise for pre-decrement.

Similarly, statements cover the full C standard. Conditional statements carry optional labels as internal forms, so that an if statement in the concrete syntax becomes `Sif`(e) `then` s_1 `else` s_2 `join` $\bot$.

B 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.]

$e ::= \mathit{Eval}\ v@\mathit{vt}$	Value
$\mid \mathit{Evar}\ x$	Variable
$\mid \mathit{Efield}\ e\ \mathit{id}$	Field
$\mid \mathit{EvalOf}\ e$	Load from Object
$\mid \mathit{Ederef}\ e$	Dereference Pointer
$\mid \mathit{EaddrOf}\ e$	Address of Object
$\mid \mathit{Eunop}\ \odot\ e$	Unary Operator
$\mid \mathit{Ebinop}\ \oplus\ e_1\ e_2$	Binary Operator
$\mid \mathit{Ecast}\ e\ \mathit{ty}$	Cast
$\mid \mathit{Econd}\ e_1\ e_2\ e_3$	Conditional
$\mid \mathit{Esize}\ \mathit{ty}$	Size of Type
$\mid \mathit{Ealign}\ \mathit{ty}$	Alignment of Type
$\mid \mathit{Eassign}\ e_1\ e_2$	Assignment
$\mid \mathit{EassignOp}\ \oplus\ e_1\ e_2$	Operator Assignment
$\mid \mathit{EpostInc}\ \oplus\ e$	Post-Increment/Decrement
$\mid \mathit{Ecomma}\ e_1\ e_2$	Expression Sequence
$\mid \mathit{Ecall}\ e_f(\overline{e}_{args})$	Function Call
$\mid \mathit{Eloc}\ l@\mathit{lt}$	Memory Location
$\mid \mathit{Eparen}\ e\ \mathit{tyt}$	Parenthetical with Optional Cast

Fig. 12: Expression Syntax

$$
\begin{aligned}
s ::= {} & \texttt{Sskip} \\
\mid & \texttt{Sdo}\ e \\
\mid & \texttt{Sseq}\ s_1\ s_2 \\
\mid & \texttt{Sif}(e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{join}\ L \\
\mid & \texttt{Swhile}(e)\ \texttt{do}\ s\ \texttt{join}\ L \\
\mid & \texttt{Sdo}\ s\ \texttt{while}\ (e)\ \texttt{join}\ L \\
\mid & \texttt{Sfor}(s_1; e; s_2)\ \texttt{do}\ s_3\ \texttt{join} \\
\mid & \texttt{Sbreak} \\
\mid & \texttt{Scontinue} \\
\mid & \texttt{Sreturn} \\
\mid & \texttt{Sswitch}\ e\ \{\ \overline{(L, s)}\ \}\ \texttt{join} \\
\mid & \texttt{Slabel}\ L :\ s \\
\mid & \texttt{Sgoto}\ L
\end{aligned}
$$

Fig. 13: Tagged C Abstract Syntax

In the below definition, memories are ranged over by m, local environments by le, and continuations by k.

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

States in general contain a memory, a local environment, and a continuation.

B.1 Memories

B.2 Environments

B.3 Continuations

A continuation acts like a stack of pending operations. The base of the stack is *Kemp*. *Kdo* indicates that a do statement is evaluating an expression. *Kseq* with parameter s indicates that, after the current statement is done executing, s is next. *Kif* means that execution is evaluating the test expression of an if statement, and its parameters are the branches of the if. Similarly, the test continuations for while, do-while, and for loops indicate that the test expression is being evaluated. The associated loop continuations indicate that execution is in the loop body. They continuations carry all of the information of the original loop.

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

C 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 **undef**@vt@lt for some vt and 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.

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

D Step Rules

D.1 Sequencing rules

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

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

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

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

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

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

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

D.2 Conditional rules

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

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

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

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

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

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

D.3 Loop rules

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

D.4 Contexts

Our expression semantics are contextual. A context $C[[]\,e]_k$ is a function from an expression to an expression, with a "kind" flag k (left-hand or right-hand, LH or RH).

$$\begin{aligned}
C[e]_{\text{LH}} ::= &\\
&|e\\
&|\mathit{Ederef}\ C[e]_{\text{RH}}\\
&|\mathit{Efield}\ C[e]_{\text{RH}}\ \mathit{id}
\end{aligned}$$

$$
\begin{aligned}
C[e]_{\text{RH}} ::= \quad & \\
& | e \\
& | \mathit{EvalOf}\ C[e]_{\text{LH}} \\
& | \mathit{EaddrOf}\ C[e]_{\text{LH}} \\
& | \mathit{Eunop}\ \odot\ C[e]_{\text{RH}} \\
& | \mathit{Ebinop}\ \oplus\ C[e_1]_{\text{RH}}\ e_2 \\
& | \mathit{Ebinop}\ \oplus\ e_1\ C[e_2]_{\text{RH}} \\
& | \mathit{Ecast}\ C[e]_{\text{RH}}\ \mathit{ty} \\
& | \mathit{EseqAnd}\ C[e_1]_{\text{RH}}\ e_2 \\
& | \mathit{EseqOr}\ C[e_1]_{\text{RH}}\ e_2 \\
& | \mathit{Econd}\ C[e_1]_{\text{RH}}\ e_2\ e_3 \\
& | \mathit{Eassign}\ C[e_1]_{\text{LH}}\ e_2 \\
& | \mathit{Eassign}\ e_1\ C[e_2]_{\text{RH}} \\
& | \mathit{EassignOp}\ \oplus\ C[e_1]_{\text{LH}}\ e_2 \\
& | \mathit{EassignOp}\ \oplus\ e_1\ C[e_2]_{\text{RH}} \\
& | \mathit{EpostInc}\ \oplus\ C[e]_{\text{LH}} \\
& | \mathit{Ecall}\ C[e_1]_{\text{RH}}(\overline{e_2}) \\
& | \mathit{Ecall}\ e_1(C[\overline{e_2}]_{\text{RH}}) \\
& | \mathit{Ecomma}\ C[e_1]_{\text{RH}}\ e_2 \\
& | \mathit{Eparen}\ C[e]_{\text{RH}}\ \mathit{ty}
\end{aligned}
$$

A left-hand reduction $e \Rightarrow_{\text{LH}} e'$ relates an expression to an expression. A right-hand reduction $(\mathcal{P}, m, e) \Rightarrow_{\text{RH}} (\mathcal{P}', m', e')$ relates a triple of PC Tag, memory, and expression to another such triple. Given these reduction relations, we construct step rules for contexts in expressions.

$$\frac{C[e]_{\text{LH}} \qquad e \Rightarrow_{\text{LH}} e'}{\mathcal{E}\,(m \mid C[e] \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\,(m \mid C[e'] \gg k@\mathcal{P})}$$

$$\frac{C[e]_{\text{RH}} \qquad (\mathcal{P}, m, e) \Rightarrow_{\text{RH}} (\mathcal{P}', m', e')}{\mathcal{E}\,(m \mid C[e] \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\,(m' \mid C[e'] \gg k@\mathcal{P}')}$$

All that remains is to give the expression reductions themselves.

$$\frac{le[id] = (l, \_, pt, ty)}{\mathit{Evar}\ id \Rightarrow_{\text{LH}} \mathit{Eloc}\ l@pt}$$

$$\frac{le[id] = \bot \quad ge[id] = \text{VAR}(l, \_, pt, ty)}{\mathit{Evar}\ id \Rightarrow_{\text{LH}} \mathit{Eloc}\ l@pt}$$

$$\frac{le[id] = \bot \quad ge[id] = \text{VAR}(f, pt)}{\mathit{Evar}\ id \Rightarrow_{\text{LH}} \mathit{Efloc}\ l@pt}$$

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

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

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

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

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

$$\frac{\begin{array}{c} v_1\ \langle\oplus\rangle\ v_2 = v' \quad \mathit{vt}' \leftarrow \mathbf{BinopT}(\oplus, \mathcal{P}, \mathit{vt}_1, \mathit{vt}_2) \\ e = \mathit{Ebinop}\ \ \oplus\ \ (\mathit{Eval}\ v_1@\mathit{vt}_1)\ (\mathit{Eval}\ v_2@\mathit{vt}_2) \end{array}}{(\mathcal{P}, m, e) \Rightarrow_{\mathrm{RH}} (\mathcal{P}, m, \mathit{Eval}\ v'@\mathit{vt}')}$$

$$\frac{\neg \mathit{isptr}(\mathit{ty}_1)\ \neg \mathit{isptr}(\mathit{ty}_2) \qquad \mathit{pt} \leftarrow \mathbf{IICastT}(\mathcal{P}, \mathit{vt}_1)}{(\mathcal{P}, m, \mathit{Ecast}\ (\mathit{Eval}\ v@\mathit{vt} : \mathit{ty}_1)\ \mathit{ty}_2) \Rightarrow_{\mathrm{RH}} (\mathcal{P}, m, \mathit{Eval}\ v@\mathit{vt}' : \mathit{ty}_2)}$$

$$\frac{\begin{array}{cc} \mathit{ty}_1 = \mathit{ptr}\ \mathit{ty}_1' & \neg \mathit{isptr}(\mathit{ty}_2) \\ m[v]_{|\mathit{ty}_1'|} = \_@\mathit{vt}@\overline{\mathit{lt}} & \mathit{vt} \leftarrow \mathbf{PICastT}(\mathcal{P}, \mathit{pt}, \mathit{vt}, \overline{\mathit{lt}}) \end{array}}{(\mathcal{P}, m, \mathit{Ecast}\ (\mathit{Eval}\ v@\mathit{pt} : \mathit{ty}_1)\ \mathit{ty}_2) \Rightarrow_{\mathrm{RH}} (\mathcal{P}, m, \mathit{Eval}\ v@\mathit{vt}' : \mathit{ty}_2)}$$

$$\frac{\begin{array}{cc} \neg \mathit{isptr}(\mathit{ty}_1) & \mathit{ty}_2 = \mathit{ptr}\ \mathit{ty}_2' \\ m[v]_{|\mathit{ty}_2'|} = \_@\mathit{vt}_2@\overline{\mathit{lt}} & \mathit{pt} \leftarrow \mathbf{IPCastT}(\mathcal{P}, \mathit{vt}_1, \mathit{vt}_2, \overline{\mathit{lt}}) \end{array}}{(\mathcal{P}, m, \mathit{Ecast}\ (\mathit{Eval}\ v@\mathit{vt}_1 : \mathit{ty}_1)\ \mathit{ty}_2) \Rightarrow_{\mathrm{RH}} (\mathcal{P}, m, \mathit{Eval}\ v@\mathit{pt} : \mathit{ty}_2)}$$

$$\frac{\begin{array}{cc} \mathit{ty}_1 = \mathit{ptr}\ \mathit{ty}_1' & \mathit{ty}_2 = \mathit{ptr}\ \mathit{ty}_2' \\ m[v]_{|\mathit{ty}_1'|} = m[v]_{|\mathit{ty}_2'|} = \_@\mathit{vt}@\overline{\mathit{lt}} & \mathit{pt}' \leftarrow \mathbf{PPCastT}(\mathcal{P}, \mathit{pt}, \mathit{vt}, \overline{\mathit{lt}}) \end{array}}{(\mathcal{P}, m, \mathit{Ecast}\ (\mathit{Eval}\ v@\mathit{pt} : \mathit{ty}_1)\ \mathit{ty}_2) \Rightarrow_{\mathrm{RH}} (\mathcal{P}, m, \mathit{Eval}\ v@\mathit{pt}' : \mathit{ty}_2)}$$

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

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

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

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

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

$$\frac{\begin{array}{c} m[l]_{|\mathit{ty}|} = v_1@\mathit{vt}_1@\overline{\mathit{lt}} \qquad m' = m[l \mapsto v_2@\mathit{vt}'@\overline{\mathit{lt}}'] \\ \mathcal{P}', \mathit{vt}', \mathit{lt}' \leftarrow \mathbf{StoreT}(\mathcal{P}, \mathit{pt}, \mathit{vt}, \mathit{lt}) \end{array}}{(\mathcal{P}, m, \mathit{Eassign}\ \ (\mathit{Eloc}\ l@\mathit{pt})\ (\mathit{Eval}\ v_2@\mathit{vt}_2)) \Rightarrow_{\mathrm{RH}} (\mathcal{P}', m', \mathit{Eval}\ v_2@\mathit{vt}_2)}$$

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

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

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

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

D.5 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.

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

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

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

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

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.]

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

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

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

And finally, we have the return rules.

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

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

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