

FLEXIBLE RUNTIME SECURITY ENFORCEMENT WITH TAGGED C

SEAN ANDERSON, ALLISON NAAKTGEBOREN, AND ANDREW TOLMACH

Portland State University, Department of Computer Science
e-mail address: ander28@pdx.edu, naak@pdx.edu, apt@pdx.edu

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 retroactive security enforcement, but such mechanisms vary widely in their security goals and carry nuanced trade-offs which are often not desirable to legacy code owners. We introduce Tagged C, a C variant with a built-in *tag-based reference monitor* that supports a range of user-defined security policies. Demonstrated in this paper: two varieties of *memory safety* exploring the trade-off between security and support for low-level idioms, *secure information flow* (SIF), and *compartmentalization*.

Adjectives to sell: – flexible – accessible (more than assembly) – comprehensive

1. INTRODUCTION

Many essential technologies of modern life rely on new and old C code. The C language rose to prominence 40 years ago by powering the UNIX, Windows, and OSX operating systems, as well as major applications like the Oracle database [Mun] . and the Apache web server [Fou97]. Now our cars, smartphones, home appliances (embedded systems like your garage door or tv remote), smart homes and hospitals (Internet of Things, embedded devices), and most of the internet runs the C language family (though it may not be the only language) [Mun]. [The C language family remains a force in active development, in a 2022 more than 35% of professionals report using it [so dev survey]. Legacy codebases, especially C codebases, pose a security conundrum. They are difficult or impossible to modify, the original programmers are unavailable, and no specification of behavior (however informal), is available.

This means that it may not be feasible to fix the bugs turned up by a conservative static analysis; a more permissive but unsound one, on the other hand, may miss bugs entirely. Worse, most static analyses aim to detect undefined behavior (UB), but UBs may be used intentionally in the form of low-level idioms. Where static analysis is unsatisfactory we turn to dynamic enforcement.

A tag-based reference monitor is a mechanism for dynamic security enforcement. It associates a metadata tag with the data in the underlying system, and throughout execution it updates these tags according to a set a predefined rules. If the program would violate a rule, the system halts instead, replacing a security violation with failstop behavior. By attaching such a monitor to the C language, we enable dynamic enforcement of arbitrary kinds of security, tuned so that non-standard but benign code can still run, while actually dangerous activity is failstopped.

This is the underlying concept of PIPE, an ISA extension that implements a reference monitor in hardware, as well as similar systems such as ARM MTE and [that thing from Binghamton]. Being implemented at the ISA level, these systems currently require their policies to be defined in terms of assembly code, usually with the help of a compiler. Instead, we attach tags to the C language itself, and aim to use PIPE as a compilation target, translating the high-level tags into PIPE’s ISA primitives.

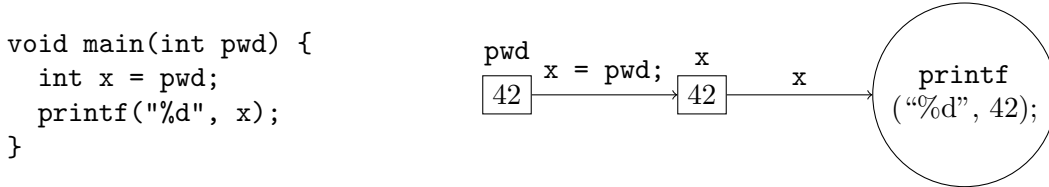
We offer the following contributions:

- A full formal semantics for Tagged C, formalized in Coq
- A complete set of *control points* at which the language interfaces with the policy
- A Tagged C interpreter, implemented in Coq and extracted to Ocaml
- Policies implementing (1) compartmentalization, (2) realistic, permissive memory models from the literature (PVI and PNVI), and (3) Secure Information Flow (SIF)

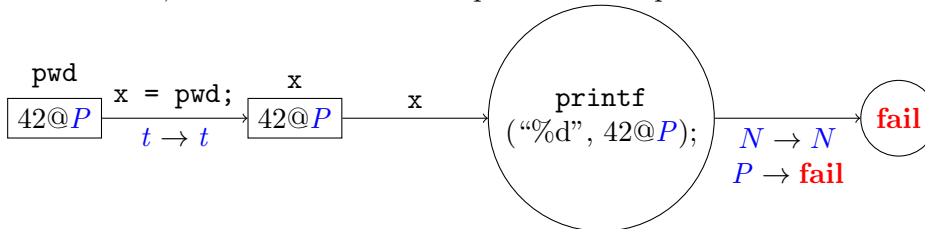
In the next section, we give a full account of the formal semantics of Tagged C, including its control points. Then in section 3.1, we describe how we attach a memory safety policy to it, in the process giving some justification of how we chose to attach the control points. In section 3.4, we give a similar description of a secure information flow policy. We round out our policies in section 3.3 with a compartmentalization policy. In ?? we discuss the degree to which the design meets our goals of flexibility and applicability to realistic security concerns.

[TODO: update outdated map]

What is Metadata Tagging? Consider a very simple security requirement: “do not `printf` the password.” For simplicity, we will suppose that `pwd` is an integer in this case. It is not possible to enforce this statically in general, though it may be over-approximated.



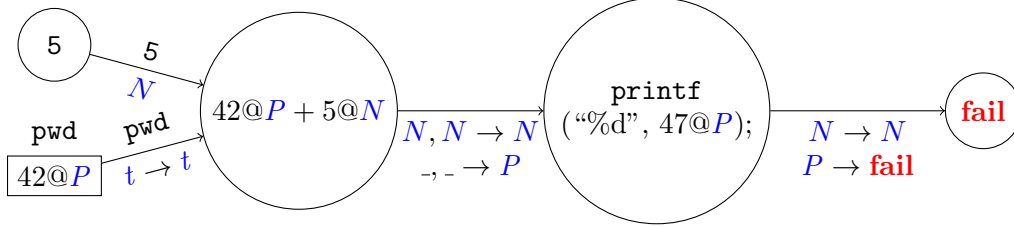
We need a way to dynamically track the value of `pwd` as it moves through the variable `x` and is then passed to `printf`. We can do so by associating metadata with the value; namely, that it originates in `pwd`. We will write this “tagged” value `42@P`, with the `@` symbol denoting a value tagged with metadata. All other values will be tagged `N`, for “not `pwd`.” Then when we copy `pwd` into `x`, it will bring its tag with it unchanged, as represented by the pattern `t → t` under the arrow. When we call `printf`, we must check that the tag is `N`, and if it is `P`, we would like to failstop rather than permit the call.



The points at which the tags are checked and either propagated or updated are termed *control points*, and the particular set of rules that are applied to tags at each control point is a *tag rule*. Collectively, the tag rules form a *policy*.

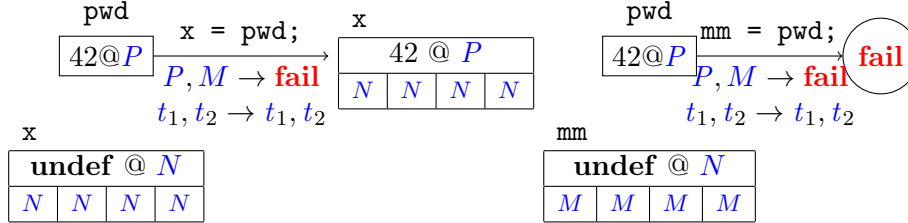
If our goal is to prevent `pwd` from being leaked, our policy should also prevent values derived from `pwd` from leaking. This means that if we add `pwd` to a constant, we still keep the result tagged `P`—otherwise, someone watching our output could deduce `P` by subtraction.

```
void main(int pwd) {
  printf(pwd+5);
}
```



Function calls aren't the only means of leaking data, however. We might additionally want to prevent `pwd` from leaking by being written mmapped memory. Supposing the `mm` is such a location, we would separately tag the location itself (not just the value in the location) to designate it as such. We'll call that tag `M`. [TODO: the diagrams for this are pretty rough.]

```
void main(int pwd) {
  int x = pwd;
  mm = pwd;
}
```



Finally, we may be concerned with less leaks of `pwd`, such as this one:

```
void main(int pwd) {
  for (int i; i<pwd; i++) {
    printf(i);
  }
}
```

This will give away the exact value of `pwd`! In order to prevent this, we must keep track of when we are inside of a loop or conditional that depends on `pwd`. To this end, we carry an additional tag associated with the global state. [TODO: diagram here, too, except I'm not sure how to effectively represent the added complexity.]

General Purpose Security Enforcement. [TODO: Emphasize breadth]

CompCert C. Tagged C is built on top of CompCert C, the C semantics formalized along with the CompCert verified compiler. Our interpreter is likewise built on top of the CompCert C reference interpreter [Ler09]. We chose CompCert C as a base because it is a widely used and well-supported C semantics, with a working interpreter and a full formalization. Being written in Coq, it is ideal for future proof work.

$\odot ::= !$	$\oplus ::= +$	$ \ll$	$e ::= \text{Eval } v@vt$	Value
$ \sim$	$ -$	$ \gg$	$ \text{Evar } x$	Variable
$ -$	$ \times$	$ \&$	$ \text{Efield } e \text{ id}$	Field
$ \text{abs}$	$ \div$	$ $	$ \text{EvalOf } e$	Load from Object
	$ \%$	$ \wedge$	$ \text{Ederef } e$	Dereference Pointer
$s ::= \text{Sskip}$			$ \text{EaddrOf } e$	Address of Object
$ \text{Sdo } e$			$ \text{Eunop } \odot e$	Unary Operator
$ \text{Sseq } s_1 \ s_2$			$ \text{Ebinop } \oplus \ e_1 \ e_2$	Binary Operator
$ \text{Sif}(e) \text{ then } s_1 \text{ else } s_2 \text{ join}$			$ \text{Hcast } e \text{ ty}$	Cast
$ \text{Swhile}(e) \text{ do } s \text{ join } L$			$ \text{Econd } e_1 \ e_2 \ e_3$	Conditional
$ \text{Sdo } s \text{ while } (e) \text{ join } L$			$ \text{Esize } ty$	Size of Type
$ \text{Sfor}(s_1; e; s_2) \text{ do } s_3 \text{ join}$			$ \text{Ealign } ty$	Alignment of Type
$ \text{Sbreak}$			$ \text{Eassign } e_1 \ e_2$	Assignment
$ \text{Scontinue}$			$ \text{EassignOp } \oplus \ e_1 \ e_2$	Operator Assignment
$ \text{Sreturn}$			$ \text{EpostInc } \oplus \ e$	Post-Increment/Decrement
$ \text{SSwitch } e \{ \overline{(L, s)} \} \text{ join}$			$ \text{Ecomma } e_1 \ e_2$	Expression Sequence
$ \text{Slabel } L : s$			$ \text{Ecall } e_f(\bar{e}_{args})$	Function Call
$ \text{Sgoto } L$			$ \text{Eloc } l@lt$	Memory Location
			$ \text{Eparen } e \text{ ty } t$	Parenthetical with Optional Cast

FIGURE 1. Tagged C Abstract Syntax

PIPE Backend Implementation. In ??, Chhak et al. introduce a verified compiler from a toy high-level language with tags to a control-flow-graph-based intermediate representation with a PIPE-based ISA. This establishes a proof-of-concept for compiling a source language’s tag policy to realistic hardware. They take advantage of the fact that, like everything else in a PIPE system, instructions in memory carry tags. Instruction tags are statically determined at compile-time. They “piggyback” information about source-level control points onto the tags of the instructions that implement those source constructs.

Tagged C is designed to be implemented in the same way. But, before we can soundly transmit tag rules from the source language to the assembly level, we also need to protect the basic control-flow properties of the source language. So, a compiled Tagged C requires a backend that can at the very least protect its control flow. In the case of a PIPE-based backend, we would run a basic stack-and-function-pointer-safety policy in parallel with whatever Tagged C policy the user has provided.

2. THE LANGUAGE

Tagged C uses the full syntax of CompCert C [Ler09] with minimal modification (fig. 1). 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 Section 3.

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.

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 (Figure 1), statements by s , and continuations by k . The continuations are defined in appendix A, and step rules in appendix C.

Global environments, ranged over by ge , map identifiers to either function or global variable definitions, including the variable’s location in memory. Local environments, ranged over by le , map identifiers to atoms. Memories m map integers to triples: a value, a “value tag” vt , and a list of “location tags” \overline{lt} .

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, $m[l \dots l + s \mapsto v@vt@\overline{lt}]_s$ denotes storing that many bytes. Visually, we will represent whole atoms in memory as condensed boxes, with their location tags separate. For example, a four-byte aligned address:

$$l$$

$v@vt$			
lt_1	lt_2	lt_3	lt_4

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.]

Name	Inputs	Outputs	Control Points
GlobalT	$id \in ident, s \in \mathbb{N}$	pt, vt, \overline{lt}	Program initialization
FieldT	pt, id	pt'	Field Access
LoadT	$\mathcal{P}, pt, vt, \overline{lt}$	vt'	ValOf, AssignOp, PostIncr
StoreT	$\mathcal{P}, pt, vt_1, vt_2, \overline{lt}$	$\mathcal{P}', vt', \overline{lt}'$	Assign
ConstT		vt	Const, PostIncr
UnopT	\odot, \mathcal{P}, vt	vt	Unary Operation
BinopT	$\oplus, \mathcal{P}, vt_1, vt_2$	vt'	Binary Operation
MallocT	\mathcal{P}, vt	$\mathcal{P}', pt, \boxed{vt, \overline{lt}}$	Call to <code>malloc</code>
FreeT	\mathcal{P}, vt	$\mathcal{P}', pt, \boxed{vt, \overline{lt}}$	Call to <code>free</code>
PICastT	$\mathcal{P}, pt, \boxed{vt, \overline{lt}}$	vt	Cast from pointer to scalar
IPCastT	$\mathcal{P}, vt_1, \boxed{vt_2, \overline{lt}}$	pt	Cast from scalar to pointer
PPCastT	$\mathcal{P}, pt, \boxed{vt, \overline{lt}}$	pt'	Cast between pointers
IICastT	\mathcal{P}, vt_1	pt	Cast between scalars
ExprSplitT	\mathcal{P}, vt	\mathcal{P}'	Control-flow split points in expressions
ExprJoinT	$\mathcal{P}, \mathcal{P}', vt$	\mathcal{P}'', vt'	Parenthetical expressions
SplitT	$\mathcal{P}, vt, \boxed{L}$	\mathcal{P}'	Split points (??)
LabelT	\mathcal{P}, L	\mathcal{P}'	Label
CallT	\mathcal{P}, f, f'	\mathcal{P}'	Call
ExtCallT	$\mathcal{P}, f, f', \overline{vt}$	\mathcal{P}'	External Call
LocalT	$\mathcal{P}, x \in ident, s \in \mathbb{N}$	pt, vt, \overline{lt}	Call
ArgT	\mathcal{P}, vt, f, x, s	$\mathcal{P}', pt, vt', \overline{lt}$	Call
RetT	$\mathcal{P}_{CLE}, \mathcal{P}_{CLR}, vt, f$	\mathcal{P}', vt'	Return
DeallocT	$\mathcal{P}, id \in ident, s \in \mathbb{N}$	vt, \overline{lt}	Return

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

3. TAGS AND POLICIES

Tagged C can enforce a wide range of policies, as follows. A policy consists of a tag type τ , a default tag inhabiting that type, and an instantiation of each tag rule identified in section 3.

For each policy under discussion, we will give a code example of the sort of security situation in which it might be useful. We will introduce a formal characterization drawn from the literature of a security property that a correct policy should satisfy. [TODO: talk about properties somewhere before this?] Then we will walk through the important tag rules, and the control points that call them, introducing step rules as needed. Finally, if there are any implementation details that are necessary to realize a policy, we discuss those.

Control Points with Side-effects and Optional Arguments. Chhak et al. [CTA21] give a general strategy for mapping Tagged C’s tag rules onto instructions in a PIPE target. But as they note, translating tag rules in full generality requires adding extra instructions that may be unnecessary for some policies. The most problematic situation is when a Tagged-C control point requires 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.

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. If a policy does not make use of the options, it will be sound to compile without the extra instructions. Optional inputs and outputs are marked with `boxes`.

Name Tags. When we want to define a per-program policy, we need to be able to attach tags to the program’s functions, globals, and so on. We do this by automatically embedding their identifiers in tags, which are available to all policies. These are called *name tags* and are ranged over by *nt*. We give name tags to:

- Function identifiers
- Function arguments, written `f.x`
- Local and global variables
- Labels

3.1. Basic Memory Safety. Let’s begin by walking through a common type of policy: memory safety. Variations of memory safety have been enforced in PIPE at the assembly level already, but what does it look like to enforce it at the source level? Consider some example code:

```
void main() {
    int[1] x;
    int[1] y;
    *x = 0;
    *(x+1) = 0;
}
```

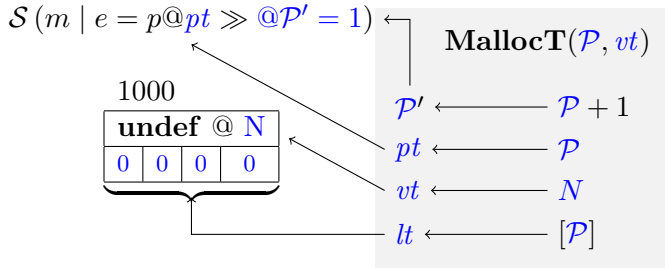
The above code is undefined behavior in C, because it writes to the address one past the end of the array pointed to by `x`. In Tagged C, it is defined in correspondence to the allocation strategy. `x` and `y` are given concrete addresses, and the program writes to the address of `x+1`. It’s possible that this address is free, in which case there is no harm; but if `y` is allocated there, then it will write to the first address of `y`.

For our example, we’ll assume a straightforward first-fit allocator, with the heap growing upward from address 1000, and the stack growing downward from address 2000. Our set of tags consists of N , for non-pointers, and pointer “colors” $c \in \mathbb{N}$. The PC Tag (\mathcal{P}) tracks the next color to allocate, so it’s initialized to 0, and everything else is N . N is the default for constants. [TODO: needs another pass]

$$\mathcal{S}(m \mid e = \text{malloc}(1@t) \gg @P = 0)$$

1000	1004	1008	...	1092	1096
undef @ N	undef @ N	undef @ N		undef @ N	undef @ N
N N N N	N N N N	N N N N		N N N N	N N N N

The call to `malloc` allocates the region from 1000 to 1039, and returns a pointer to the base address, 1000. We consult the policy to determine (1) the tag on the resulting value, (2) the updated tags on the allocated memory region, and (3) the updated PC Tag. Specifically, we invoke the **MallocT** tag rule, which takes the PC Tag and the tag on the size argument and returns these three updated tags.

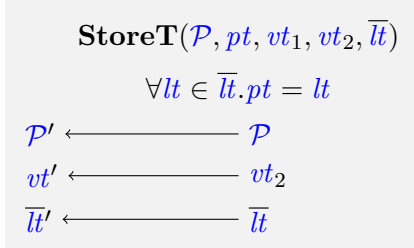


In this case, we tag the pointer and the memory region with the current count, and then increment the count. Once the pointer is stored in `x`, our memory is:

We do the same for allocating `y`, to get the memory:

1000	1004	1008	...	1092	1096
undef @ N	undef @ N	undef @ N		1004 @ 1	1000 @ 0
0 0 0 0	1 1 1 1	N N N N		N N N N	N N N N

Next, the program stores a 0 to address 1000. The constant 0 takes on the default tag, *I*. The policy needs to check that this store is valid, in addition to determining the tags on the value that is stored. This check is performed by comparing the tag on the pointer to the tags on memory—each byte being written, in case the pointer is misaligned. Then the tag on the value being stored is propagated with it into memory, unchanged.



$\mathcal{P} : 2$

1000	1004	1008	...	1092 (y)	1096 (x)
undef@N	undef@N	undef@N		1004@1	1000@0
0 0 0 0	1 1 1 1	N N N N		N N N N	N N N N

Finally, on the last line, we add 2 to `x`, which invokes the **BinopT** tag rule to combine the tags on the arguments. **BinopT** takes as argument the operation \oplus . In memory safety terms, we can add a pointer to a non-pointer in either order, and we can subtract a

non-pointer from a pointer (but not the reverse), to yield a pointer to the same object. We can subtract two pointers to the same object from one another to yield a non-pointer, the offset between them. All other binary operations are only permitted between non-pointers.

$$\text{BinopT}(\oplus, \mathcal{P}, vt_1, vt_2)$$

$$\begin{array}{l} \mathcal{P}' \longleftarrow \mathcal{P} \\ vt' \longleftarrow \begin{cases} c & \text{when } (\oplus, vt_1, vt_2) \text{ is } +, c, N \mid +, N, c \mid -, c, N \\ N & \text{is } -, c, c \mid -, N, N \\ \text{fail} & \text{is otherwise} \end{cases} \end{array}$$

So, when we try to write through the pointer 1004@0, the bytes at addresses 1004-1007 are tagged 1, and the policy issues a failstop.

3.2. PVI Memory Safety. [NOTE: Andrew suggested putting this first, seeing if it works.]

The simple memory safety policy described above is too restrictive to run many real-world C programs, because they contain undefined behavior that is nevertheless part of the “de facto standard” [?]. These low-level idioms are one reason that we might settle for isolating risky code in a compartment instead of enforcing full memory safety.

Memarian et al. [?] propose two memory models that aim to capture this de facto standard, support the common low-level idioms, yet still place sufficient restrictions on programs that it remains sound to use alias analysis in optimizations. The first of these is *PVI* (provenance via integer), in which pointers remain valid when they are cast to integers, subjected to the full range of arithmetic operations, and cast back.

Memarian et al. do not propose to enforce PVI, merely to use it as an alternative to the C standard. But its relative permissiveness makes it a great target for enforcement in Tagged C!

[TODO: example of what we do want (I-P cast, maybe using low bits as flags?), and what we don’t (memory violations that use similar idioms)]

PVI Definitions. Since PVI is a more realistic policy than the basic memory safety described above, we will go into some details elided there. First of all, the distinction between heap-allocated memory, stack objects, and global variables. The latter are tagged based on their identifiers, while heap- and stack-objects are tagged dynamically using unique colors.

$$\begin{array}{ll} \tau ::= glob\ id & id \in ident \\ & dyn\ C \\ & C \in \mathbb{N} \end{array}$$

When initializing program memory, before any execution, each global *id* has its memory locations and its pointer in the global environment tagged with *glob id*, using the **GlobalT** tag rule.

[TODO: diagram]

$$\begin{aligned}
 & \text{GlobalT}(id, s) \\
 & \textcolor{blue}{pt} \leftarrow \text{glob } id \\
 & \textcolor{blue}{vt} \leftarrow N \\
 & \overline{\textcolor{blue}{lt}} \leftarrow [\text{glob } id \mid 0 \leq i < s]
 \end{aligned}$$

Stack-allocated locals are allocated at the start of a function call. Like a global environment, a local environment maps identifiers to base, bound, type, and tag. The rule is almost identical to allocation of globals, except that the stack allocator, *stack_alloc* will be more complex in order to support deallocation (in practice, it uses a normal stack structure and allocates and deallocates by increasing and decreasing a “stack pointer”).

The tag rules for allocating memory in the heap and in the stack should look familiar.

$$\begin{array}{ll}
 \text{LocalT}(\mathcal{P}, x, s) & \text{MallocT}(\mathcal{P}, vt) \\
 \textcolor{blue}{pt} \leftarrow \text{dyn } \mathcal{P} & \textcolor{blue}{pt} \leftarrow \text{dyn } \mathcal{P} \\
 \boxed{\textcolor{blue}{vt}} \leftarrow N & \boxed{\textcolor{blue}{vt}} \leftarrow N \\
 \boxed{\overline{\textcolor{blue}{lt}}} \leftarrow [\text{dyn } \mathcal{P}] & \boxed{\overline{\textcolor{blue}{lt}}} \leftarrow [\text{dyn } \mathcal{P}] \\
 \textcolor{blue}{\mathcal{P}'} \leftarrow \mathcal{P} + 1 & \textcolor{blue}{\mathcal{P}'} \leftarrow \mathcal{P} + 1
 \end{array}$$

Color Checking. As in the basic policy, when we perform a memory load or store, we check that the pointer tag on the left hand of the assignment matches the location tag on all of the bytes being loaded or stored.

$$\begin{array}{ll}
 \text{LoadT}(\mathcal{P}, pt, vt, \overline{\textcolor{blue}{lt}}) & \text{StoreT}(\mathcal{P}, pt, vt_1, vt_2, \overline{\textcolor{blue}{lt}}) \\
 \text{assert } \forall \textcolor{blue}{lt} \in \overline{\textcolor{blue}{lt}}. pt = \textcolor{blue}{lt} & \text{assert } \forall \textcolor{blue}{lt} \in \overline{\textcolor{blue}{lt}}. pt = \textcolor{blue}{lt} \\
 \textcolor{blue}{vt}' \leftarrow vt & \textcolor{blue}{\mathcal{P}'} \leftarrow \mathcal{P} \\
 & \textcolor{blue}{vt}' \leftarrow vt_2 \\
 & \overline{\textcolor{blue}{lt}'} \leftarrow \overline{\textcolor{blue}{lt}}
 \end{array}$$

Color Propagation. In our example memory safety policy, we placed significant restrictions on the ways that pointer-tagged values could be subject to integer operations. In PVI, this is not the case: all unary operations maintain the tag on their input, and all binary operations where exactly one argument is tagged as a pointer propagate that tag to their result. Performing an operation with two pointer-tagged values sets the tag on the result to *N*. It can still be used as an integer, but if cast back to a pointer it will be invalid.

$$\begin{array}{ll}
 \text{UnopT}(\odot, \mathcal{P}, vt) & \text{BinopT}(\oplus, \mathcal{P}, vt_1, vt_2) \\
 \textcolor{blue}{\mathcal{P}'} \leftarrow \mathcal{P} & \textcolor{blue}{\mathcal{P}'} \leftarrow \mathcal{P} \\
 \textcolor{blue}{vt}' \leftarrow vt & \textcolor{blue}{vt}' \leftarrow \text{case } (vt_1, vt_2) \text{ of} \\
 & \quad \text{dyn } n, N \Rightarrow \text{dyn } n \\
 & \quad \text{glob } id, N \Rightarrow \text{glob } id \\
 & \quad N, t \Rightarrow t
 \end{array}$$

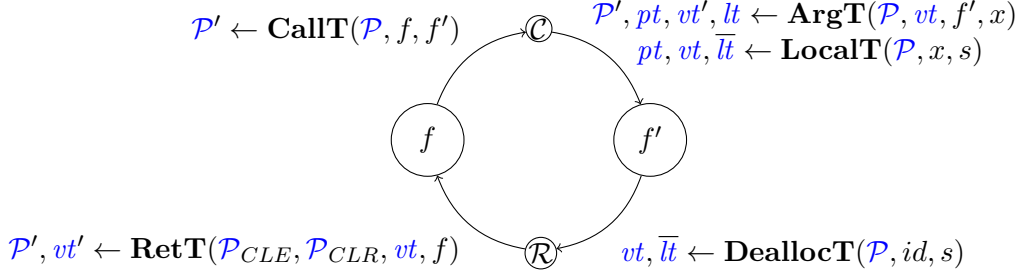


FIGURE 2. Structure of a function call

3.3. Compartmentalization. In a perfect world, all C programs would be memory safe. But it is unfortunately common for a codebase to contain undefined behavior that will not be fixed, including memory undefined behavior. This may occur because developers intentionally use low-level idioms that are UB [?]. Or the cost and potential risk of regressions may make it undesirable to fix bugs in older code, as opposed to code under active development that is held to a higher standard [BBC⁺10].

A compartmentalization policy isolates potentially risky code, such as code with known UB, from safety-critical code, minimizing the damage that can be done if a vulnerability is exploited. This is a very common form of protection that can be implemented at many levels. It is often built into a system's fundamental design, like a web browser sandbox untrusted javascript. But for our use-case, we consider a compartmentalization scheme being added to the system after the fact.

[TODO: a few words about least privilege, the other main compartmentalization use.]

Let's assume that we have a set of compartment identifiers, ranged over by C , and a mapping from function identifiers to compartments, $comp(f)$. This mapping must be provided by a security engineer.

Coarse-grained Protection. The core of a compartmentalization scheme is once again memory protection. For the simplest version, we will enforce that memory allocated by a function is only accessible by functions that share its compartment. To do that, we need to keep track of which compartment we're in, 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. 2 with some function f calling f' . Three of these steps feature control points. 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, **RetT** updates both the PC Tag and the tag on the returned value.

In our compartmentalization policy, we define a tag to be a compartment identifier or the default N tag.

$$\tau ::= C|N$$

At any given time, the PC Tag carries the compartment of the active function. This is kept up to date by the **CallT** and **RetT** rules. Note that Tagged C automatically keeps track of the PC Tag at the time of a call, so that it can be used as a parameter in the return.

$$\begin{array}{ll}
\text{CallT}(\mathcal{P}, f, f') & \text{RetT}(\mathcal{P}_{CLE}, \mathcal{P}_{CLR}, vt, f) \\
\text{let } C := \text{comp}(f') \text{ in} & \mathcal{P}' \leftarrow \mathcal{P}_{CLR} \\
\mathcal{P}' \leftarrow \mathcal{P} & vt' \leftarrow vt
\end{array}$$

Now that we know which compartment we're in, we can make sure that its memory is protected. This will essentially work just like the basic memory safety policy, 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 memory tags to the PC Tag, not the pointer.

$$\begin{array}{lll}
\text{MallocT}(\mathcal{P}, vt) & \text{LoadT}(\mathcal{P}, pt, vt, \overline{lt}) & \text{StoreT}(\mathcal{P}, pt, vt_1, vt_2, \overline{lt}) \\
pt \leftarrow N & \text{assert } \forall lt \in \overline{lt}. \mathcal{P} = lt & \text{assert } \forall lt \in \overline{lt}. lt = \mathcal{P} \\
\boxed{vt} \leftarrow N & vt' \leftarrow vt & \mathcal{P}' \leftarrow \mathcal{P} \\
\boxed{\overline{lt}} \leftarrow [\mathcal{P}] & & vt' \leftarrow N \\
\mathcal{P}' \leftarrow \mathcal{P} & & \overline{lt}' \leftarrow \overline{lt}
\end{array}$$

Sharing Memory. The above policy works if our compartments only ever communicate by passing non-pointer values. In practice, this is far too restrictive! Many libraries take pointers and operate on the associated memory, starting with the most basic ones, like the standard library's string functions. And yet, if we are forced to include large external libraries in the same compartment as critical code, we have lost much of the value of compartmentalization.

So, we need to modify our policy to account for intentional sharing of memory. In our example, the function `setup` in compartment *A* will allocate several buffers, call a function that fills the buffer `msg`, and then pass `msg` to the `strlen` function in compartment *B*.

```
// In compartment A
void setup() {
    int* key = malloc(100);
    char* msg = malloc(100);
    char* log = malloc(100);

    fetch_msg(&msg); // Function also in A

    ...
    int msg_size = strlen(msg); // StdLib function in B
    ...
}
```

Naturally, we want `strlen` to be able to read `msg`, but we would rather that it not read `key`, if it should happen to be malicious. Even if `log` isn't as sensitive, we have no reason to share it, so it should be protected.

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.

We can handle a mix of allocations that will be passed and those that will not 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.

```
// In compartment A
void setup() {
    int* key = malloc(100);
    char* msg = malloc_share(100);
    char* log = malloc(100);

    fetch_msg(&msg); // Function also in A

    ...
    int msg_size = strlen(msg); // StdLib function in B
    ...
}
```

Mandatory Access Control. Mandatory access control works by associating objects in memory with the compartments that are allowed to access them.

Memory Shared by Capability. Mandatory access control requires the policy designer to identify every pair of object and compartment that it will be shared with. This may require too much analysis if objects are shared widely throughout the system. Conversely, it does not distinguish between accesses via a valid pointer and those that are the result of UB. [TODO: example]

3.4. Secure Information Flow. Memory safety and compartmentalization both either prevent or mitigate memory errors. But programs can be memory safe and still do insecure things! Consider the following code, in which we have some error-handling code that writes to a log.

```
int checked_div(int a, int b) {
    if (a % b == 0) {
        return a / b;
    } else {
        fprintf(log, "%d should divide %d but doesn't\n", b, a);
        return 0;
    }
}

void main(int factor) {
    ...
    int key = read_and_parse(keyfile);

    int dividend = checked_div(key, factor);
    if (!dividend) {
        ...
    } else {
```

```

    ...
  }
}

```

The `checked_div` function sometimes writes its arguments to a log, which is reasonable enough, except when it's called with a key as an argument! Suddenly we have keys being written to an unexpected and probably unprotected file.

This is an instance of problematic information-flow. The solution is to implement a *secure information flow* (SIF) policy in Tagged C. SIF is a variant of *information flow control* (IFC) described in the venerable Denning and Denning [DD77]. At its simplest, if we classify inputs and outputs to the program into secure (“high”) and public (“low”) classifications, then the high inputs do not influence the low outputs. This generalizes to an arbitrary set of security classes, but our first example is concerned with just two: the value returned from `read_and_parse` and the output to the log. In our treatment of this example, we will describe a policy tailored to this particular set of security classes.

SIF Example Policy 1. Let's assume that `read_and_parse` is an *external* function—that is, we will not model its internal behavior, so we know nothing about the value it returns. We can therefore treat that value as an input, and track its influence through the system.

For this initial, simplified policy, we will assume that it is the only input that we care about, so we have four classes of tags. The default tag N represents values that are not tainted by the sensitive input, the tag $vtaint$ represents values that have been influenced by `read_and_parse`, and the tag $pc\ f\ \bar{L}$ carries a set of labels representing that the current control-flow of the program is tainted (we will discuss this in detail below.) Lastly, the tag vol marks the memory locations of *volatile* global variables. Volatile variables represent mmapped regions or memory that for other reasons is accessible outside the process.

Initially, the PC Tag is $pc\ f\ \emptyset$, and all values and memory locations are tagged N . The taint tags are introduced at the external call to `read_and_parse`. At the same time, all external calls must check that they aren't leaking a tainted value!

$$\begin{array}{ll}
 \tau ::= N & \text{ExtCallT}(\mathcal{P}, f, f', \overline{vt}) \\
 vtaint & \text{assert } \forall vt \in \overline{vt}. vt = N \wedge \mathcal{P} = N \\
 pc\ f\ \bar{L} & \mathcal{P}' \leftarrow \mathcal{P} \\
 vol & vt' \leftarrow \text{case } f \text{ of} \\
 & \quad read_and_parse \Rightarrow vtaint \\
 & \quad _ \Rightarrow vt
 \end{array}$$

When two values are combined with a binary operation, the resulting value is tainted if either of them was. We define this as the *join* or *least-upper-bound* operator, \sqcup . We will then compare tags according to a partial order, the *no-higher-than* operator, \sqsubseteq . In this case, $a \sqsubseteq b$ means that a does not have higher privilege than b , and so information is allowed to flow from a to b .

$$t_1 \sqcup t_2 \triangleq \begin{cases} vtaint & \text{if } t_1 = vtaint \\ vtaint & \text{if } t_2 = vtaint \\ N & \text{otherwise} \end{cases} \quad t_1 \sqsubseteq t_2 \triangleq \begin{cases} \text{false} & \text{if } t_1 = vtaint \text{ and } t_2 = vol \\ \text{true} & \text{otherwise} \end{cases}$$

The policy needs to failstop if a tainted value becomes visible to the outside world. That can happen when the value is passed as an argument to an external function, as we

saw above, or when it is stored to volatile memory (typically representing a file or external device that might be read or might transfer.

$$\begin{array}{ll}
 \text{BinopT}(\oplus, \mathcal{P}, vt_1, vt_2) & \text{StoreT}(\mathcal{P}, pt, vt_1, vt_2, \overline{lt}) \\
 vt' \leftarrow vt_1 \sqcup vt_2 & \text{assert } \forall lt \in \overline{lt}. \mathcal{P} \sqcup pt \sqcup vt_2 \sqsubseteq lt \\
 & \mathcal{P}' \leftarrow \mathcal{P} \\
 & vt' \leftarrow \mathcal{P} \sqcup pt \sqcup vt_2 \\
 & \overline{lt}' \leftarrow \overline{lt}
 \end{array}$$

Things become trickier when we consider that the program’s control-flow itself can be tainted. This can occur in any of our semantics’ steps that can produce different statements and continuations depending on the tainted value. At that point, any change to the machine state constitutes an information flow. This is termed an *implicit flow*.

Implicit flows become much more complex outside of expressions, when we have more complex control flow. This time the taint is carried on the PC Tag itself. When the PC Tag is tainted, all stores to memory and all updates to environments must also be tainted until all branches eventually rejoin, which might be at any point. We term the point at which it is safe to remove taint a *join point*. In terms of the program’s control-flow graph, the join point of a branch is its immediate post-dominator []. [TODO: this is the Denning cited in Bay and Askarov]

In many simple programs, the join point of a conditional or loop is obvious: the point at which the chosen branch is complete, or the loop has ended. Such a simple example can be seen in fig. 3; `public1` must be tagged with the taint tag of `secret`, while it is safe to tag `public2` `N`, because that is after the join point, J. The same goes for fig. 4, if we are in a *termination-insensitive* setting [AHSS08]. In termination-insensitive noninterference, we allow an observer to glean information by the termination or non-termination of the program. So, it is safe to assume that the post-dominator *J* of the while loop is reached.

[TODO: implicit flow rules for statements]

But in the presence of unrestricted go-to statements, a join point may not be local (and sometimes may not exist within the function, assuming that we have not consolidated return points.) Consider fig. 5, which uses go-to statements to create an approximation of an if-statement whose join-point is far removed from the for-loop. The label J now has nothing to do with the semantics of any particular statement.

Luckily this can be determined statically from a function’s full control-flow graph, so we can implement it as long as we’re willing to deviate from our purely syntax-based tag rules by performing a code transformation. This can be done completely automatically; for each split point in the code, the control-flow graph identifies its join point statement, and the transformation must wrap that statement in a fresh label.

to implement the policy, we must first transform our program by adding labels at the join point of each conditional. Every statement that branches carries an optional label indicating its corresponding join point, if it has one—a function with multiple returns might not, in which case once the PC Tag is tainted, it must remain so until a return.

Intransitive SIF. Our second example involves information from outside of the system ending up somewhere it isn’t supposed to.

```
void sanitize(buf);
char* sql_query(char* query);
```

```

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

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

J:  public2 = 42;

    return public2;
}

```

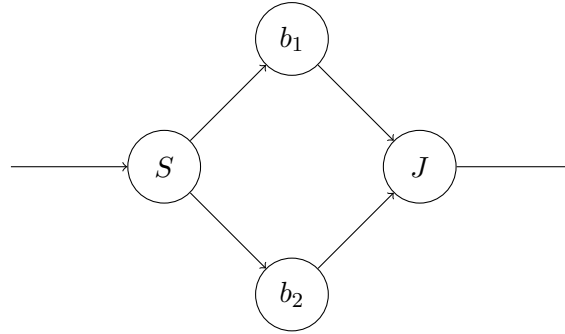


FIGURE 3. Leaking via if statements

```

int f(bool secret) {
    int public1=1;
    int public2;

S:  while (secret) {
b1:    public1 = 1;
        secret = false;
    }

J:  public2 = 42;

    return public2;
}

```

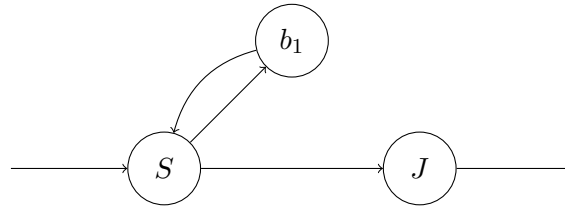


FIGURE 4. Leaking via while statements


```

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

    while (secret) {
        goto b1;
    }

b2: public1 = 1;
    goto J;

b1: public1 = 1;

J:  public2 = 42;
    return public2;
}

```

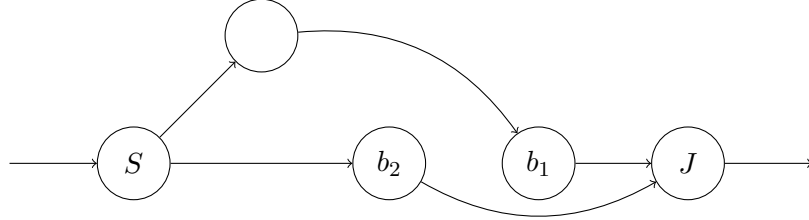


FIGURE 5. Cheating with go-tos

```

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

    scanf("%19f", name);
    sanitize(name);
    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;”.

In this example, we want to implement an *intransitive integrity* SIF policy: we wish to allow `name` to influence the result of `sanitize`, naturally, and the result of `sanitize` to influence the value passed to `sql_query`, but we do not wish for `name` to influence `sql_query` directly.

We are now forced to consider multiple sources of information, `name` and the output of `sanitize`, as well as multiple information “sinks,” the input to `sanitize` and `sql_query`. Let Σ be the set of these three identifiers, ranged over by σ . A value may be tainted any subset of Σ , written *vtaint* $\bar{\sigma}$. The PC Tag tracks the current function identifier and an association list of labels and sources. Each pair (L, σ) in the association list indicates that until reaching label L , the state itself has been influenced by σ .

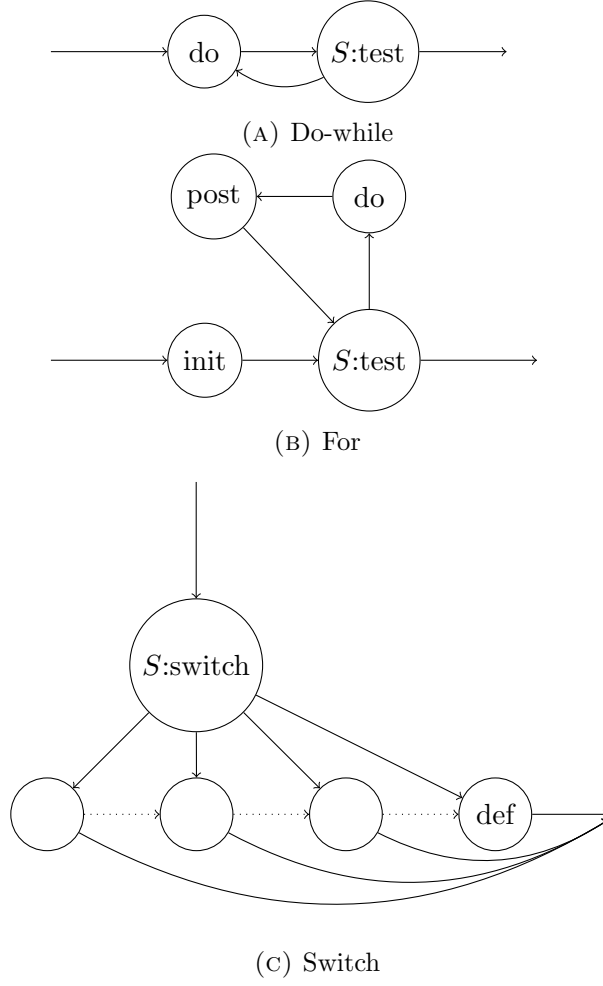


FIGURE 6. Remaining Branch Statements

$$\tau ::= \text{vtaint } \overline{id}$$

$$pc\ f\ \overline{(L, id)}$$

$$N$$

We define the join operation in this new setting, as well as the *minus* operation ($t_1 - t_2$).

$$t_1 \sqcup t_2 \triangleq \begin{cases} \text{vtaint } (\overline{\sigma}_1 \cup \overline{\sigma}_2) & \text{if } t_1 = \text{vtaint } \overline{\sigma}_1 \text{ and } t_2 = \text{vtaint } \overline{\sigma}_2 \\ \text{vtaint } (\overline{\sigma}_2 \cup \{\sigma \mid (L, \sigma) \in \overline{(L, \sigma)}_1\}) & \text{if } t_1 = pc\ f\ \overline{(L, \sigma)}_1 \text{ and } t_2 = \text{vtaint } \overline{\sigma}_2 \\ \text{vtaint } (\overline{\sigma}_1 \cup \{\sigma \mid (L, \sigma) \in \overline{(L, \sigma)}_2\}) & \text{if } t_2 = pc\ f\ \overline{(L, \sigma)}_2 \text{ and } t_1 = \text{vtaint } \overline{\sigma}_1 \\ \perp & \text{otherwise} \end{cases}$$

$$t - \sigma \triangleq \begin{cases} \text{vtaint } \overline{\sigma} - \sigma & \text{if } t = \text{vtaint } \overline{\sigma} \\ \perp & \text{otherwise} \end{cases}$$

And once again we wish to define the “no-higher-than” relation. In this case, recall that we want to avoid the `name` argument flowing to `sql_query`. So we will define that `sql_query(query)`, as a sink, is strictly higher security than `sql_query(name)`, and every other combination is fine.

$$t_1 \sqsubseteq (t_2, t_3) \triangleq \begin{cases} \mathbf{f} & \text{if } t_1 = \text{vtaint } \bar{\sigma}, t_2 = \text{sql_query}, t_3 = \text{query}, \text{ and } \text{name} \in \bar{\sigma} \\ \mathbf{t} & \text{otherwise} \end{cases}$$

Tainting and Checking Arguments and Returns. A function argument or return value can be either a source or a sink. So, when they are processed by the argument and return rules, we must both check that the value being passed or returned is not tainted by a forbidden source, and then add the new source to its taint. Recall that we want the first argument to the `sanitize` function to “forget” the influence of `name`.

<p>ArgT(\mathcal{P}, vt, f, x, s)</p> <p>assert $\mathcal{P} \sqcup vt \sqsubseteq (f, x)$</p> <p>let $vt_1 := vt - \{\text{sanitize}(\text{src})\}$ in</p> <p>let $vt_2 := vt_1 \sqcup \text{vtaint } \{f(x)\}$ in</p> <p>$vt' \leftarrow vt_2$</p> <p>$\mathcal{P}' \leftarrow \mathcal{P}$</p> <p>$\bar{lt} \leftarrow [N]$</p>	<p>RetT($\mathcal{P}_{CLE}, \mathcal{P}_{CLR}, vt, f$)</p> <p>assert $\mathcal{P} \sqcup vt \subseteq \text{sink } (f.\text{ret})$</p> <p>let $vt_1 := vt - \{\sigma \mid \sigma/f.\text{ret} \in I\}$ in</p> <p>let $vt_2 := vt_1 \sqcup \text{tainted } \{f.\text{ret}\}$ in</p> <p>$vt' \leftarrow vt_2$</p> <p>$\mathcal{P}' \leftarrow \mathcal{P}$</p>
---	--

Dynamic Sinks and Globals. One scenario that does not really match the others is when the sink is dynamically allocated memory. In this case, we need to tag the memory at allocation-time with the forbidden sources. Global variables are also possible sources or sinks, so we initialize their tags to carry this information.

<p>MallocT(\mathcal{P}, vt)</p> <p>$pt \leftarrow \mathcal{P} \sqcup \text{vtaint } \emptyset$</p> <p>$\boxed{vt} \leftarrow \text{vtaint } \emptyset$</p> <p>$\boxed{\bar{lt}} \leftarrow [\text{sink } f.m]$</p> <p>$\mathcal{P}' \leftarrow \mathcal{P}$</p>	<p>GlobalT(id, s)</p> <p>$pt \leftarrow \text{vtaint } \emptyset$</p> <p>$vt \leftarrow \text{vtaint } \{id\}$</p> <p>$\bar{lt} \leftarrow [\text{sink } id]$</p>
---	--

PC Tag Taint. It now becomes slightly more complicated to keep track of the join-point labels associated with various sources. [TODO: fix the alignment here.]

<p>SplitT($\mathcal{P}, vt, \boxed{L}$)</p> <p>let $pc\ f\ (\overline{L, \sigma}) := \mathcal{P}$ in</p> <p>let $\text{vtaint } \sigma := vt$ in</p> <p>$\mathcal{P}' \leftarrow pc\ f\ ((\overline{L, \sigma}) \cup (L, \sigma))$</p>	<p>LabelT(\mathcal{P}, L)</p> <p>let $pc\ f\ (\overline{L, \sigma}) := \mathcal{P}$ in</p> <p>$\mathcal{P}' \leftarrow pc\ f\ \{(L', \sigma) \mid (L', \sigma) \in \overline{L, \sigma} \wedge L \neq L'\}$</p>
---	---

The branching constructs are rather complicated, involving multiple steps and manipulations of the continuation that are not that relevant to their control points. Rather than

give their semantics in full, it suffices to identify which transitions contain **SplitT** control points. In fig. 6, these are the transitions from the state marked S . Their semantics are given in full in the appendix.

Realizing IFC. In order to implement an IFC policy, we need to specify the rules that it needs to enforce. The positive here is that the rules are not dependent on one another (with the exception of declassification rules), and default to permissiveness when no rule is given. We assume that the user would supply a separate file consisting of a list of triples: the source, the sink, and the type of rule. This is then translated into the policy.

The other implementation detail to consider are the label tags. These resemble instruction tags, and that is exactly how they would be implemented: as a special instruction tag on the appropriate instruction, which might be an existing instruction or a specially added no-op, that the processor handles by introducing a tag corresponding to that label.

It remains to generate those labels. For purposes of an IFC policy, we first generate the program’s control flow graph. Then, for each if, while, do-while, for, and switch statement, we identify the immediate post-dominator in the graph, and wrap it in a label statement with a fresh identifier. That identifier is also added as a field in the original conditional statement. The tags associated with the labels are initialized at program state—in the case of IFC, these defaults declare that there are no secrets to lower when it is reached.

4. IMPLEMENTING TAGGED C WITH PIPE

[TODO: more detailed description of Chris’ work and why the optional arguments are optional]

5. 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

5.1. Limitations of the Tag Mechanism. 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.

6. RELATED WORK

Reference Monitors. The concept of a reference monitor was first introduced fifty years ago in [And72]: 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 [GM82].)

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) [DVR⁺14] 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 [WFGW⁺20], stack safety [SDB19], 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]

7. FUTURE WORK

We have presented the language and a reference interpreter, built on top of the CompCert interpreter [Ler09], 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

- [AHSS08] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In Sushil Jajodia and Javier Lopez, editors, *Computer Security - ESORICS 2008*, pages 333–348, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [And72] James P. Anderson. Computer Security Technology Planning Study. Technical report, U.S. Air Force Electronic Systems Division, 10 1972.
- [BBC⁺10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 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. doi:10.1145/1646353.1646374.
- [CTA21] CHR Chhak, Andrew Tolmach, and Sean Anderson. Towards formally verified compilation of tag-based policy enforcement. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, page 137–151, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3437992.3439929.
- [DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, jul 1977. doi:10.1145/359636.359712.
- [DVR⁺14] Udit Dhawan, Nikos Vasilakis, Raphael Rubin, Silviu Chiricescu, Jonathan M Smith, Thomas F Knight Jr., Benjamin C Pierce, and André DeHon. PUMP: A Programmable Unit for Metadata Processing. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, HASP ’14, page 8:1–8:8, New York, NY, USA, 2014. ACM. URL: <http://doi.acm.org/10.1145/2611765.2611773>, doi:10.1145/2611765.2611773.
- [Fou97] The Apache Software Foundation. Apache http server project, 1997. URL: <https://httpd.apache.org/>.
- [GM82] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, 1982. URL: <http://dblp.uni-trier.de/db/conf/sp/sp1982.html#GoguenM82a>.
- [Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, jul 2009. doi:10.1145/1538788.1538814.
- [Mun] Daniel Muñoz. After all these years, the world is still powered by c programming. URL: <https://www.toptal.com/c/after-all-these-years-the-world-is-still-powered-by-c-programming>.
- [SDB19] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. StkTokens: Enforcing Well-bracketed Control Flow and Stack Encapsulation using Linear Capabilities. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28, 2019.
- [WFGW⁺20] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. Cornucopia:

Temporal safety for cheri heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 608–625, 2020. doi:10.1109/SP40000.2020.00098.

APPENDIX A. CONTINUATIONS

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

APPENDIX B. 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@ $vt@l\bar{t}$** for some vt and $\bar{l}\bar{t}$, 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.

$$globals\ xs\ b = \begin{cases} (m_0, ge_0) & \text{if } xs = \varepsilon \\ (m[p \dots p + |ty| \mapsto \mathbf{undef@}vt@l\bar{t}]_{|ty|}, & \text{if } xs = (id, ty) :: xs' \\ ge[id \mapsto (p, p + |ty|, ty, pt)]) & \text{and } pt, vt, \bar{l}\bar{t} \leftarrow \mathbf{GlobalT}(id, s) \\ & \text{where } (m, ge) = globals\ xs' (b + |ty|) \end{cases}$$

APPENDIX C. STEP RULES

C.1. **Sequencing rules.** $\frac{}{\mathcal{S}(m \mid Sdo\ e \gg k@P) \longrightarrow \mathcal{E}(m \mid e \gg Kdo; k@P)}$

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

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

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

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

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

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

C.2. **Conditional rules.** $\frac{s = \text{Sif}(e) \text{ then } s_1 \text{ else } s_2 \text{ join } L}{\mathcal{S}(m \mid s \gg k@P) \longrightarrow \mathcal{E}(m \mid e \gg Kif\ s_1\ s_2\ L; k@P)}$

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

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

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

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

$$\frac{}{\mathcal{S}(m \mid \text{Scontinue} \gg Kswitch2; k@P) \longrightarrow \mathcal{S}(m \mid \text{Scontinue} \gg k@P)}$$

C.3. **Loop rules.** $\frac{s = \text{Swhile}(e) \text{ do } s' \text{ join } L}{\mathcal{S}(m \mid s \gg k@P) \longrightarrow \mathcal{E}(m \mid e \gg KwhileTest\ e\ s'\ L; k@P)}$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

C.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.

$context\ k\ C[e] ::=$	
$ context\ k\ \lambda e.e$	
$ context\ LH\ \lambda e.Ederef\ C[e]$	where $context\ RH\ C[e]$
$ context\ LH\ \lambda e.Efield\ C[e]\ id$	where $context\ RH\ C[e]$
$ context\ RH\ \lambda e.EvalOf\ C[e]$	where $context\ LH\ C[e]$
$ context\ RH\ \lambda e.EaddrOf\ C[e]$	where $context\ LH\ C[e]$
$ context\ RH\ \lambda e.Eunop\ \odot\ C[e]$	where $context\ RH\ C[e]$
$ context\ RH\ \lambda e.Ebinop\ \oplus\ C[e_1]\ e_2$	where $context\ RH\ C[e_1]$
$ context\ RH\ \lambda e.Ebinop\ \oplus\ e_1\ C[e_2]$	where $context\ RH\ C[e_2]$
$ context\ RH\ \lambda e.Ecast\ C[e]\ ty$	where $context\ RH\ C[e]$
$ context\ RH\ \lambda e.EseqAnd\ C[e_1]\ e_2$	where $context\ RH\ C[e_1]$
$ context\ RH\ \lambda e.EseqOr\ C[e_1]\ e_2$	where $context\ RH\ C[e_1]$
$ context\ RH\ \lambda e.Econd\ C[e_1]\ e_2\ e_3$	where $context\ RH\ C[e_1]$
$ context\ RH\ \lambda e.Eassign\ C[e_1]\ e_2$	where $context\ LH\ C[e_1]$
$ context\ RH\ \lambda e.Eassign\ e_1\ C[e_2]$	where $context\ RH\ C[e_2]$
$ context\ RH\ \lambda e.EassignOp\ \oplus\ C[e_1]\ e_2$	where $context\ LH\ C[e_1]$
$ context\ RH\ \lambda e.EassignOp\ \oplus\ e_1\ C[e_2]$	where $context\ RH\ C[e_2]$
$ context\ RH\ \lambda e.EpostInc\ \oplus\ C[e]$	where $context\ LH\ C[e]$
$ context\ RH\ \lambda e.Ecall\ C[e_1]\ (\overline{e_2})$	where $context\ RH\ C[e_1]$
$ context\ RH\ \lambda e.Ecall\ e_1(C[\overline{e_2}])$	where $context\ RH\ C[e]$ for $e \in \overline{e_2}$
$ context\ RH\ \lambda e.Ecomma\ C[e_1]\ e_2$	where $context\ RH\ C[e_1]$
$ context\ RH\ \lambda e.Eparen\ C[e]\ ty$	where $context\ RH\ C[e]$
$ context\ RH\ \lambda e.Eparen\ C[e]\ ty\ t$	where $context\ RH\ C[e]$

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.

$$\frac{context\ LH\ C[e] \quad e \Rightarrow_{LH} e'}{\mathcal{E}(m \mid C[e] \gg k@P) \longrightarrow \mathcal{E}(m \mid C[e] \gg k@P)}$$

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

C.5. Expression Rules. $\frac{le[id] = (l, -, pt, ty)}{Evar\ id \Rightarrow_{LH} Eloc\ l@pt}$

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

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

$$\begin{array}{c}
\frac{}{(\mathcal{P}, m, \text{Ederef } (\text{Eval } v@vt)) \Rightarrow_{\text{RH}} (\mathcal{P}, m, \text{Eloc } (to_ptr \ v)@vt)} \\
\frac{ty = TStruct \ id \vee ty = TUnion \ id \quad offset \ id \ fld = \delta \quad pt' \leftarrow \mathbf{FieldT}(pt, id)}{Efield \ (\text{Eval } p@pt : ty) \ fld \Rightarrow_{\text{LH}} \text{Eloc } (p + \delta)@pt'} \\
\frac{m[l]_{|ty|} = v@vt@l \quad vt' \leftarrow \mathbf{LoadT}(\mathcal{P}, pt, vt, l)}{(\mathcal{P}, m, \text{EvalOf } (\text{Eloc } l@pt) : ty) \Rightarrow_{\text{RH}} (\mathcal{P}, m, \text{Eval } v@vt')} \\
\\
\frac{}{(\mathcal{P}, m, \text{EaddrOf } (\text{Eloc } p@pt)) \Rightarrow_{\text{RH}} (\mathcal{P}, m, \text{Eval } p@pt)} \\
\frac{\langle \odot \rangle v = v' \quad vt \leftarrow \mathbf{UnopT}(\odot, \mathcal{P}, vt)}{(\mathcal{P}, m, \text{Eunop } \odot \ (\text{Eval } v@vt)) \Rightarrow_{\text{RH}} (\mathcal{P}, m, \text{Eval } v'@vt')} \\
\frac{v_1 \langle \oplus \rangle v_2 = v' \quad vt' \leftarrow \mathbf{BinopT}(\oplus, \mathcal{P}, vt_1, vt_2) \quad e = \text{Ebinop } \oplus \ (\text{Eval } v_1@vt_1) \ (\text{Eval } v_2@vt_2)}{(\mathcal{P}, m, e) \Rightarrow_{\text{RH}} (\mathcal{P}, m, \text{Eval } v'@vt')} \\
\\
\frac{\neg isptr(ty_1) \quad \neg isptr(ty_2) \quad pt \leftarrow \mathbf{ICastT}(\mathcal{P}, vt_1)}{(\mathcal{P}, m, \text{Ecast } (\text{Eval } v@vt : ty_1) \ ty_2) \Rightarrow_{\text{RH}} (\mathcal{P}, m, \text{Eval } v@vt' : ty_2)} \\
\frac{ty_1 = ptr \ ty'_1 \quad \neg isptr(ty_2) \quad m[v]_{|ty'_1|} = _@vt@l \quad vt \leftarrow \mathbf{PCastT}(\mathcal{P}, pt, \boxed{vt, l})}{(\mathcal{P}, m, \text{Ecast } (\text{Eval } v@pt : ty_1) \ ty_2) \Rightarrow_{\text{RH}} (\mathcal{P}, m, \text{Eval } v@vt' : ty_2)} \\
\frac{\neg isptr(ty_1) \quad ty_2 = ptr \ ty'_2 \quad m[v]_{|ty'_2|} = _@vt_2@l \quad pt \leftarrow \mathbf{IPCastT}(\mathcal{P}, vt_1, \boxed{vt_2, l})}{(\mathcal{P}, m, \text{Ecast } (\text{Eval } v@vt_1 : ty_1) \ ty_2) \Rightarrow_{\text{RH}} (\mathcal{P}, m, \text{Eval } v@pt : ty_2)} \\
\frac{ty_1 = ptr \ ty'_1 \quad ty_2 = ptr \ ty'_2 \quad m[v]_{|ty'_1|} = m[v]_{|ty'_2|} = _@vt@l \quad pt' \leftarrow \mathbf{PPCastT}(\mathcal{P}, pt, \boxed{vt, l})}{(\mathcal{P}, m, \text{Ecast } (\text{Eval } v@pt : ty_1) \ ty_2) \Rightarrow_{\text{RH}} (\mathcal{P}, m, \text{Eval } v@pt' : ty_2)} \\
\\
\frac{boolof(v) = \mathbf{t} \quad \mathcal{P}' \leftarrow \mathbf{ExprSplitT}(\mathcal{P}, vt)}{(\mathcal{P}, m, \text{EseqAnd } (\text{Eval } v@vt) \ e) \Rightarrow_{\text{RH}} (\mathcal{P}', m, \text{Epren } e \ Tbool \ \mathcal{P})} \\
\frac{boolof(v) = \mathbf{f} \quad \mathcal{P}' \leftarrow \mathbf{ExprSplitT}(\mathcal{P}, vt)}{(\mathcal{P}, m, \text{EseqAnd } (\text{Eval } v@vt) \ e) \Rightarrow_{\text{RH}} (\mathcal{P}', m, \text{Epren } (\text{Eval } 0@vt') \ Tbool \ \mathcal{P})} \\
\frac{boolof(v) = \mathbf{t} \quad \mathcal{P}' \leftarrow \mathbf{ExprSplitT}(\mathcal{P}, vt)}{(\mathcal{P}, m, \text{EseqOr } (\text{Eval } v@vt) \ e) \Rightarrow_{\text{RH}} (\mathcal{P}', m, \text{Epren } (\text{Eval } 1@vt') \ Tbool \ \mathcal{P})} \\
\frac{boolof(v) = \mathbf{f} \quad \mathcal{P}' \leftarrow \mathbf{ExprSplitT}(\mathcal{P}, vt)}{(\mathcal{P}, m, \text{EseqOr } (\text{Eval } v@vt) \ e) \Rightarrow_{\text{RH}} (\mathcal{P}', m, \text{Epren } e \ Tbool \ \mathcal{P})} \\
\frac{e' = \begin{cases} e_1 & \text{if } boolof(v) = \mathbf{t} \\ e_2 & \text{if } boolof(v) = \mathbf{f} \end{cases} \quad \mathcal{P}' \leftarrow \mathbf{ExprSplitT}(\mathcal{P}, vt)}{(\mathcal{P}, m, \text{Econd } (\text{Eval } v@vt) \ e_1 \ e_2) \Rightarrow_{\text{RH}} (\mathcal{P}', m, \text{Epren } e' \ \mathcal{P})}
\end{array}$$

$$\begin{array}{c}
\frac{m[l]_{|ty|} = v_1 @ vt_1 @ \bar{lt} \quad m' = m[l \mapsto v_2 @ vt' @ \bar{lt}']}{\frac{\mathcal{P}', vt', \bar{lt}' \leftarrow \text{StoreT}(\mathcal{P}, pt, vt_1, vt_2, \bar{lt})}{(\mathcal{P}, m, Eassign (Eloc l @ pt) (Eval v_2 @ vt_2)) \Rightarrow_{RH} (\mathcal{P}', m', Eval v_2 @ vt_2)}} \\
\\
\frac{m[l]_{|ty|} = v_1 @ vt @ \bar{lt} \quad \oplus \in \{+, -, *, /, \%, <, >, \&, ^, |\} \quad vt' \leftarrow \text{LoadT}(\mathcal{P}, pt, vt, \bar{lt})}{\frac{e = Eassign (Eloc l @ pt) (Ebinop \oplus (Eval v_1 @ vt')) (Eval v_2 @ vt_2))}{(\mathcal{P}, m, EassignOp \oplus (Eloc l @ pt) (Eval v_2 @ vt_2)) \Rightarrow_{RH} (\mathcal{P}, m, e)}} \\
\\
\frac{m[l] = v @ vt @ \bar{lt} \quad \oplus \in \{+, -\} \quad vt' \leftarrow \text{LoadT}(\mathcal{P}, pt, vt, \bar{lt})}{\frac{e = Ecomma (Eassign (Eloc l @ pt) (Ebinop \oplus Eval v @ vt' 1 @ def)) (Eval v @ vt')}{(\mathcal{P}, m, EpostInc \oplus Eloc l @ pt) \Rightarrow_{RH} (\mathcal{P}, m, e)}} \\
\\
\frac{}{(\mathcal{P}, m, Ecomma (Eval v @ vt) e) \Rightarrow_{RH} (\mathcal{P}, m, e)} \\
\\
\frac{\mathcal{P}', vt' \leftarrow \text{ExprJointT}(\mathcal{P}, \mathcal{P}', vt)}{(\mathcal{P}, m, Eparen e ty \mathcal{P}') \Rightarrow_{RH} (\mathcal{P}'', m, Eval v @ vt')}
\end{array}$$

C.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.

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

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

$$\text{frame } xs \text{ as } m = \begin{cases} (m''[p \mapsto \text{undef} @ vt @ \bar{lt}]_{|ty|}, & \text{if } xs = (id, ty) :: xs' \\ le'[id \mapsto (p, p + |ty|, ty, pt)]) & \text{where } (m', p) \leftarrow \text{stack_alloc } |ty| \ m, \\ & pt, vt, \bar{lt} \leftarrow \text{LocalT}(\mathcal{P}, x, s), \\ & \text{and } (m'', le') = \text{frame } xs' \text{ as } m' \\ \\ (m''[p \mapsto v @ vt' @ \bar{lt}]_{|ty|}, & \text{if } as = (id, ty, v @ vt) :: as' \text{ and } xs = \varepsilon \\ le'[id \mapsto (p, p + |ty|, ty, pt)]) & \text{where } (m', p) \leftarrow \text{stack_alloc } |ty| \ m, \\ & \mathcal{P}', pt, vt', \bar{lt} \leftarrow \text{ArgT}(\mathcal{P}, vt, f, x, s), \\ & \text{and } (m'', le') = \text{frame } xs' \text{ as } m' \\ \\ (m, \lambda x. \perp) & \text{if } xs = \varepsilon \text{ and } as = \varepsilon \end{cases}$$

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

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.]

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

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

$$\frac{\mathcal{P}', pt, \boxed{vt, \overline{lt}} \leftarrow \mathbf{MallocT}(\mathcal{P}, vt) \quad m', p \leftarrow \text{heap_alloc } size \ m \quad m'' = m' [p + i \mapsto (\mathbf{undef}, vt, lt)]_{size}}{\mathcal{C}(m \mid \text{malloc}((size@t)) \gg k@\mathcal{P}) \longrightarrow \mathcal{R}(m'' \mid \text{Eval } p@pt \gg k@\mathcal{P}')} \quad \text{MallocT}$$

And finally, we have the return rules.

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

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

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