'

Policies

ANONYMOUS AUTHOR(S)

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*.

1 INTRODUCTION

Many essential technologies of modern life rely on new and old C code. The C language rose to prominence 30 years ago by powering the UNIX, Windows, and OSX operating systems, as well as major applications like the Oracle database [?] . and the Apache web server [Foundation 1997]. 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) [?]. [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
- Proposed *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)

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

In the next section, we give a full account of the formal semantics of Tagged C, including its control points. Then in section 4.1, 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 4.5, we give a similar description of a secure information flow policy. We round out our policies in section 4.2 with a compartmentalization policy. In **??** we discuss the degree to which the design meets our goals of flexibility and applicability to realistic security concerns.

2 RELATED WORK

CompCert C. Tagged C is built on top of CompCert C, the C semantics formalized along with the CompCert verified compiler. Our interpeter is likewise built on top of the CompCert C reference interpreter [Leroy 2009]. 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.

Reference Monitors. The concept of a reference monitor was first introduced fifty years ago in **??**: 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 **??**.)

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) [Dhawan et al. 2014] 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 [Wesley Filardo et al. 2020], stack safety [Skorstengaard et al. 2019], 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.

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

$$
\begin{aligned}
\odot &::= \; ! \mid \sim \mid - \mid \mathit{abs} \\
\oplus &::= + \mid - \mid \times \mid \div \mid \% \mid \ll \mid \gg \mid \& \mid \,|\, \mid \wedge
\end{aligned}
$$

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

$e ::= \mathit{Eval}\ v@\mathit{vt}$	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
$\mid \mathit{Eparen}\ e\ \mathit{ty}\ t$	Parenthetical with Optional Cast

Fig. 1. Tagged C Abstract Syntax

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.

3 THE LANGUAGE

Tagged C uses full C syntax with minimal modification (fig. 1), but its semantics differ in two key respects. First, 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.

Secondly, and more crucially, 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 4.

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.”

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.

Tagged C uses a small-step reduction semantics, given in full in the appendix. We will introduce a limited number of step rules as they become relevant.

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

l

<table><tr><td colspan="4">$v@vt$</td></tr><tr><td>lt_1</td><td>lt_2</td><td>lt_3</td><td>lt_4</td></tr></table>

States can be of several kinds, denoted by their script prefix: a *general state* $\mathcal{S}(\ldots)$, an *expression state* $\mathcal{E}(\ldots)$, a *call state* $C(\ldots)$, or a *return state* $\mathcal{R}(\ldots)$. Finally, the special state *failstop* ($\mathcal{F}(\ldots)$) represents a tag failure, and carries the state that produced the failure.

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

Expressions (**??**) use a contextual semantics; a call expression stores the context in the continuation all with the caller’s continuation.

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 `malloc`
FreeT	$\mathcal{P}, vt$	$\mathcal{P}', pt, \boxed{vt, \overline{lt}}$	Call to `free`
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

4 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 4.

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. [Chhak et al. 2021] 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 $\boxed{\text{boxes}}$.

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

4.1 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* x = malloc(1);
  int* y = malloc(1);
  *x = 0;
  *(x+1) = 0;
}
```

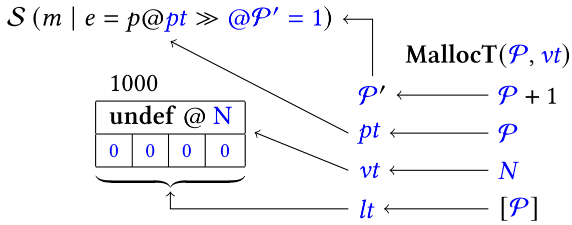
The above code is undefined behavior in C, because it writes to the address one past the end of the array pointed to by `x`. In Tagged C, it is defined in correspondence to the allocation strategy. `x` and `y` are given concrete addresses, and the program writes to the address of `x+1`. It's possible 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.

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

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.

$$\textbf{StoreT}(\mathcal{P}, pt, vt_1, vt_2, \overline{lt})$$

$$\textbf{assert}\ \forall lt \in \overline{lt}.pt = lt$$

$$\begin{aligned}
\boldsymbol{\mathcal{P}'} &\longleftarrow \mathcal{P} \\
\boldsymbol{vt'} &\longleftarrow vt_2 \\
\boldsymbol{\overline{lt'}} &\longleftarrow \overline{lt}
\end{aligned}$$

$\mathcal{P}$: 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 ⊕. 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.

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

$$\begin{aligned}
\boldsymbol{\mathcal{P}'} &\longleftarrow \mathcal{P} \\
\boldsymbol{vt'} &\longleftarrow \text{case } (\oplus,\ vt_1,\ vt_2) \text{ of} \\
&\quad +, c, N \mid +, N, c \mid -, c, N \Rightarrow c \\
&\quad -, c, c \mid _, N, N \Rightarrow N \\
&\quad _, c_1, c_2 \Rightarrow \text{fail}
\end{aligned}$$

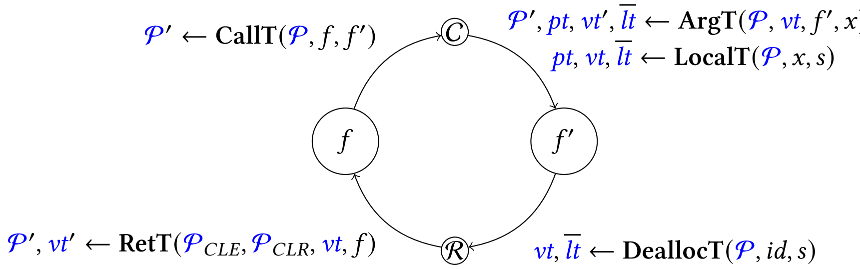


Fig. 2. Structure of a function call

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.

Realizing Memory Safety. A brief description is in order of how this policy would be implemented by a compiler to a PIPE target. This will serve to outline the basic structure of the compilation scheme described in [Chhak et al. 2021].

4.2 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 [Bessey et al. 2010].

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.

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 to example functions, f and 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 memory, tagging their values and locations with the results of **ArgT**. 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.

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

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.

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

Sharing Memory. The above policy is functional 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.

In either case, our first step is to distinguish which allocation we want to pass. We do this by labeling the statement that contains the relevant call to `malloc` The annotation could be performed manually, or perhaps automatically using some form of escape analysis. The name of the label doesn't matter, it will just be referenced via its name tag in the policies.

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

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

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

Executing a labeled statement invokes the **LabelT**$(\mathcal{P}, L)$ tag rule to update the PC Tag. Since we don't want to share `log`, we will need to label it as well.

Mandatory Access Control. [TODO]

Memory Shared by Capability. [TODO]

4.3 PVI Memory Safety

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* (provenanace 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.

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

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]

$$\textbf{GlobalT}(id, s)$$
$$\begin{aligned} \boldsymbol{pt} &\longleftarrow \mathit{glob\ id} \\ \boldsymbol{vt} &\longleftarrow N \\ \overline{\boldsymbol{lt}} &\longleftarrow [\mathit{glob\ id} \mid 0 \le i < s] \end{aligned}$$

Stack-allocated locals are allocated at the start of a function call. Like a global environment, a local environment maps indentifiers 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.

$$\textbf{LocalT}(\mathcal{P}, x, s)$$
$$\begin{aligned} \boldsymbol{pt} &\longleftarrow \mathit{dyn}\ \mathcal{P} \\ \boxed{\boldsymbol{vt}} &\longleftarrow N \\ \boxed{\overline{\boldsymbol{lt}}} &\longleftarrow [\mathit{dyn}\ \mathcal{P}] \\ \boldsymbol{\mathcal{P}'} &\longleftarrow \mathcal{P} + 1 \end{aligned}$$

$$\textbf{MallocT}(\mathcal{P}, vt)$$
$$\begin{aligned} \boldsymbol{pt} &\longleftarrow \mathit{dyn}\ \mathcal{P} \\ \boxed{\boldsymbol{vt}} &\longleftarrow N \\ \boxed{\overline{\boldsymbol{lt}}} &\longleftarrow [\mathit{dyn}\ \mathcal{P}] \\ \boldsymbol{\mathcal{P}'} &\longleftarrow \mathcal{P} + 1 \end{aligned}$$

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.

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

$$\textbf{StoreT}(\mathcal{P}, pt, vt_1, vt_2, \overline{lt})$$
$$\textbf{assert}\ \forall lt \in \overline{lt}.pt = lt$$
$$\begin{aligned} \boldsymbol{\mathcal{P}'} &\longleftarrow \mathcal{P} \\ \boldsymbol{vt'} &\longleftarrow vt_2 \\ \overline{\boldsymbol{lt'}} &\longleftarrow \overline{lt} \end{aligned}$$

[Not sure how important this is.] There are two other expressions that load from memory, and which therefore invoke this same rule, *assignop* and *postincr*. Note that the C spec has the order of evaluation for *assignop* "unsequenced"; we follow CompCert [Leroy 2009] in evaluating both the left and right completely before performing the load. Intuitively, assignment-with-an-operator is classed along with the standard assignment in the spec, so it is appropriate that it be ordered in the same way.

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.

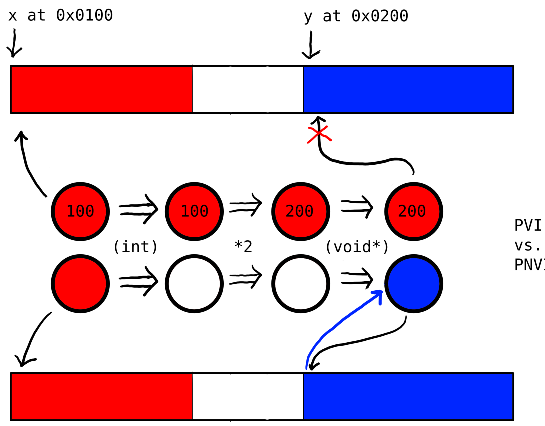


Fig. 3. Integer-pointer casts in PVI and PNVI

$$\begin{array}{l} \mathbf{UnopT}(\odot, \mathcal{P}, vt) \\ \mathcal{P}' \longleftarrow \mathcal{P} \\ vt' \longleftarrow vt \end{array} \qquad \begin{array}{l} \mathbf{BinopT}(\oplus, \mathcal{P}, vt_1, vt_2) \\ \mathcal{P}' \longleftarrow \mathcal{P} \\ vt' \longleftarrow \text{case } (vt_1,\ vt_2) \text{ of} \\ \qquad dyn\ n, N \Rightarrow dyn\ n \\ \qquad glob\ id, N \Rightarrow glob\ id \\ \qquad N, t \Rightarrow t \end{array}$$

4.4 PNVI Memory Safety

[TODO: PNVI needs a lot more motivation, especially given that the security benefits are likely marginal].

In PNVI, by contrast, an integer cast to a pointer gains the provenance of the object it points to when the cast occurs. While PNVI supports a wider range of programs, it is inconsistent with important assumptions of the C memory model, in ways that may have serious security consequences. The difference between PVI and PNVI is illustrated in Figure 3.

In PNVI, the basic provenance model remains the same as PVI, so we can reuse most of the same rules. The primary difference is what happens when we cast a pointer to an integer. In PVI, tags are propagated as normal. To support PNVI, we need the *cast* expression to update the tags of a pointer being cast to an integer and vice versa. We add two special-case steps to reflect this.

$$\frac{\boxed{m[p]_{|ty|} = \_@vt_2@\overline{lt}} \qquad vt \leftarrow \mathbf{PICastT}(\mathcal{P}, pt, \boxed{vt, \overline{lt}})}{\mathcal{E}\,(m \mid Ecast\ int\ Eval\ p@pt \gg k@\mathcal{P})\ ptr(ty) \longrightarrow \mathcal{E}\,(m \mid Eval\ p@vt \gg k@\mathcal{P})\ int}$$

$$\frac{\boxed{m[p]_{|ty|} = \_@vt_2@\overline{lt}} \qquad pt \leftarrow \mathbf{IPCastT}(\mathcal{P}, vt_1, \boxed{vt_2, \overline{lt}})}{\mathcal{E}\,(m \mid Ecast\ int\ Eval\ p@pt \gg k@\mathcal{P})\ ptr(ty) \longrightarrow \mathcal{E}\,(m \mid Eval\ p@vt \gg k@\mathcal{P})\ int}$$

For casting an integer to a pointer, we don't need the optional "peek" at the memory that it points to. We simply clear the tag on the resulting integer. On the other hand, when casting back to a pointer, we need to check the color of the object that it points to.

$$\mathbf{PICastT}(\mathcal{P}, pt, \boxed{vt, \overline{lt}})$$
$$\begin{aligned} \mathcal{P}' &\longleftarrow \mathcal{P} \\ vt &\longleftarrow N \end{aligned}$$

$$\mathbf{IPCastT}(\mathcal{P}, vt_1, \boxed{vt_2, \overline{lt}})$$
$$\textbf{assert } \exists t. \forall lt \in \overline{lt}. lt = t \land t \neq N$$
$$\begin{aligned} \mathcal{P}' &\longleftarrow \mathcal{P} \\ pt &\longleftarrow t \end{aligned}$$

Realizing the Integer-Pointer Cast. The pointer cast rules take as input the tags on the location pointed to by the argument being cast. This requires the compiler to add extra instructions to retrieve that tag. On RISCV, the sequence would be as follows, assuming that `a0` contains the value being cast. The meaning of instruction tags will be explained below.

```
lw a1 a0 0 @ RETRIEVE
sub a1 a1 a1 @ L
add a0 a1 a0 @ IPCAST
```

In the underlying assembly, we use instruction tags to inform the low-level monitor of the purpose of each instruction. `RETRIEVE` indicates a special load whose job is retrieve value and location tags from a location in memory. When it sees a `RETRIEVE` tag, the monitor allows the load even if it should failstop under the Concrete C backstop policy. If the load should failstop, however, it is given a default tag rather than the tags on the memory. A legal load recieves both the value and the location tags.

The `L` instruction tag simply denotes taking the left-operand's tag on the result of a binary operation. In this case both operations are identical, but we still need to pick one. Finally, the `IPCAST` tag declares that this instruction should mimic the Tagged-C-level rule.

4.5 Secure Information Flow

Memory safety and compartmentalization are both aimed at preventing or mitigating 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 [Denning and Denning 1977]. 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 out 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 $pc\ \overline{L}$ 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

Initially, the PC Tag is $pc\ \emptyset$, 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!

$$\begin{aligned} \tau ::=\ & N \\ & vtaint \\ & pc\ \overline{L} \\ & vol \end{aligned}$$

$$\begin{aligned} & \textbf{ExtCallT}(\mathcal{P}, f, f', \overline{vt}) \\ & \quad \textbf{assert}\ \forall vt \in \overline{vt}. vt = N \land \mathcal{P} = N \\ \boldsymbol{\mathcal{P}'} \longleftarrow\ & \mathcal{P} \\ \boldsymbol{vt'} \longleftarrow\ & \text{case } f \text{ of} \\ & read\_and\_parse \Rightarrow vtaint \\ & \_ \Rightarrow vt \end{aligned}$$

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 between.

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

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.

$$\begin{array}{rl} & \textbf{BinopT}(\oplus, \mathcal{P}, vt_1, vt_2) \\ \boldsymbol{vt'} \longleftarrow & vt_1 \sqcup vt_2 \end{array}$$

$$\begin{array}{rl} & \textbf{StoreT}(\mathcal{P}, pt, vt_1, vt_2, \overline{lt}) \\ & \quad \textbf{assert } \mathcal{P} \sqcup pt \sqcup vt_2 \sqsubseteq \\ \boldsymbol{\mathcal{P}'} \longleftarrow & \mathcal{P} \\ \boldsymbol{vt'} \longleftarrow & \mathcal{P} \sqcup pt \sqcup vt_2 \\ \overline{\boldsymbol{lt'}} \longleftarrow & \overline{lt} \end{array}$$

Now things become trickier, because 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 tained value. At that point, any change to the machine state constitutes an information flow. This is termed an *implicit flow*.

Take, for example, the expression `x = y ? (z=0) : (z=1);` where `y` is tainted and `z` is volatile. If `y` is non-zero, then `z` will be assigned 0, thus leaking one bit of information about the contents of `y`. So, we need to remember while inside of whichever branch we take that the state depends on `y` and restrict external writes accordingly.

?? discusses the building of expression contexts. For our purposes here, it is crucial to understand that when we construct a parenthetical expression, we can set the PC Tag locally within that context, so that our rules have different behavior. In this case, when we step from the condition expression to a parenthetical, we will mark it with a PC Tag that reflects the heightened security of its contents.

$$\begin{array}{l} \textit{Econd}\ (\textit{Eval}\ 1@\textit{vtaint}\ )\ (\textit{Eassign}\ (\textit{Evar}\ x)\ (\textit{Eval}\ 0@N))\ (s_2) \\ \longrightarrow \textit{Eparen}\ (\textit{Eassign}\ (\textit{Evar}\ x)\ \textit{Eval}\ 0@N)\ \textit{ty}\ t \\ \longrightarrow \textit{Eparen}\ (\textit{Eassign}\ (\textit{Eloc}\ 4000@N)\ (\textit{Eval}\ 0@N))\ \textit{ty}\ t \\ \longrightarrow \textit{Eparen}\ (\textit{Eval}\ 0@N)\ \textit{ty}\ t \end{array}$$

$$\begin{array}{rl} & \textbf{ExprSplitT}(\mathcal{P}, vt) \\ \boldsymbol{t} \longleftarrow & \mathcal{P} \sqcup vt \end{array}$$

4000

$$\boxed{0\ @\ \textbf{fail} \leftarrow \textbf{StoreT}(t, N, N, [vol])}$$

vol	*vol*	*vol*	*vol*

On the other hand, if `z` is not volatile, then we end up with a value in our parentheses, and we will remove the parentheses and carry on with execution. We need to transfer the taint information from the context onto the value, and in some cases we might want to merge it with the global PC Tag. We do this with the **ExprJoinT** tag rule.

$$\begin{array}{rl} & \textbf{ExprJoinT}(\mathcal{P}, \mathcal{P}', vt) \\ \boldsymbol{\mathcal{P}'} \longleftarrow & \mathcal{P} \\ \boldsymbol{vt'} \longleftarrow & t \end{array}$$

This principle applies to other expressions with control flow, namely *and* and *or* expressions that shortcut.

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.

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. 4; `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. 5, if we are in a *termination-insensitive* setting [Askarov et al. 2008]. In termination-insensitive noninterference, we allow for the possibility that

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

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

J:  public2 = 42;

    return public2;
}
```

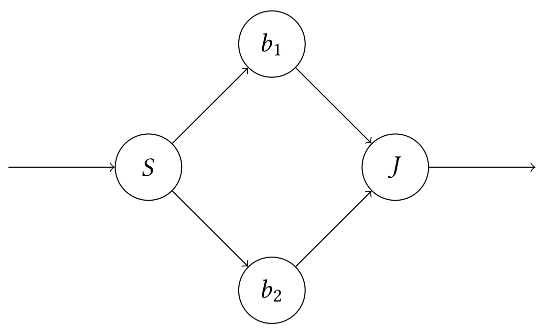


Fig. 4. Leaking via if statements

an observer could 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.

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. 6, 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 still be determined statically from a function's full control-flow graph. So, 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(src, dst);
char* sql_query(char* query);

void get_data(char* name, char* buf, int field) {
  // field: 1=address, 2=phone, 3(default)=astrological sign
  char[10] name_san;
  char[100] query;
```

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

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

J:  public2 = 42;

    return public2;
}
```

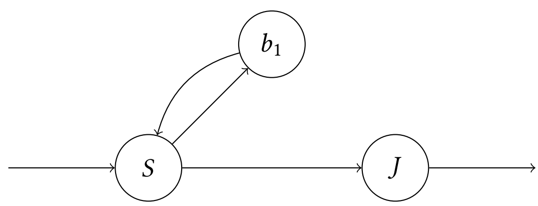


Fig. 5. Leaking via while statements

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

    while (secret) {
        goto b1;
    }

b2: public1 = 1;
    goto J;

b1: public1 = 1;

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

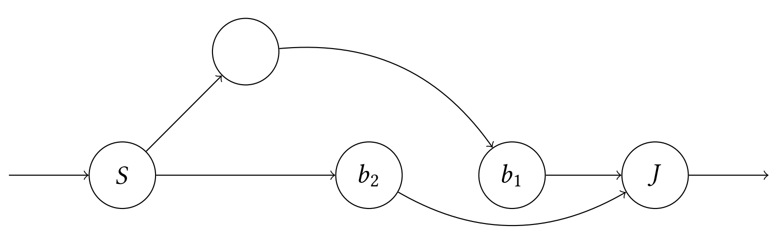
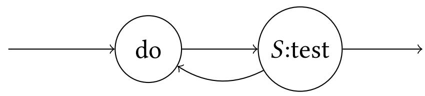
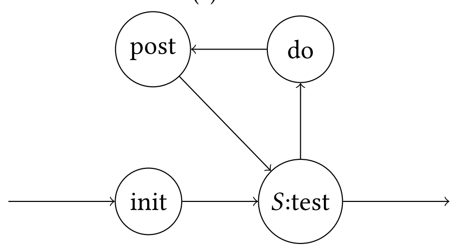


Fig. 6. Cheating with go-tos

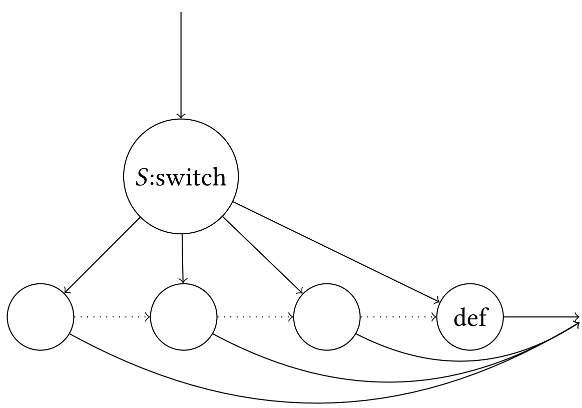
```
  sanitize(name, name_san);

  switch(field) {
    case 1:
      sprintf(query, "select address where name =");
      strncat(query, name_san, strlen(name_san));
      break;
    case 2:
```



(a) Do-while



(b) For



(c) Switch

Fig. 7. Remaining Branch Statements

```
      sprintf(query, "select phone where name =");
      strncat(query, name_san, strlen(name_san));
      break;
    default:
      sprintf(query, "select sign where name =");
      strncat(query, name, strlen(name)); // Oops!
      break;
  }

  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 particularly want to implement an *intransitive* 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.

Data Sources and Sinks. We generalize our SIF policy by identifying parts of the state as *sources* and as *sinks*. A source σ can be an argument of a function, its return value (as before), or a global (whether or not that global is volatile.) A sink ψ can be any of these, plus the set of heap objects allocated by a given function. We write them as follows:

$$\begin{array}{lll} \sigma ::= x & & \text{Global} \\ & f(x) & \text{Argument x of f} \\ & f.ret & \text{Return value of f} \end{array} \qquad \begin{array}{lll} \psi ::= x & & \text{Global} \\ & f(x) & \text{Argument x of f} \\ & f.ret & \text{Return value of f} \\ & f.m & \text{Memory owned by f} \end{array}$$

We track the influence of a particular source through the system in the form of tags on values. Sinks that are in memory have their memory locations tagged accordingly. And the PC Tag at all times tracks a set of sources that are implicitly influencing the state, described further below.

$$\begin{array}{ll} \tau ::= & vtaint\ \overline{\sigma} \\ & sink\ \psi \\ & pc\ \overline{(L, \sigma)} \\ & N \end{array}$$

A value that is tagged $vtaint\ \overline{\sigma}$ has been influnced by all of the sources in $\overline{\sigma}$. A location tagged $sink\ \psi$ belongs to ψ. The PC Tag carries a set of taints and the labels of their associated join points. We define the join and operation in this new setting, and the *minus* operation ($t_1 - t_2$).

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

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

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 `get_data(name)`, and every other combination is fine.

$$t_2 \sqsubseteq t_1 \triangleq \begin{cases} \mathbf{f} & \text{if } t_1 = vtaint\ \overline{\sigma}, t_2 = sink\ (\texttt{sql\_query(query)}), \text{ and } (\texttt{get\_data(name)}) \in \overline{\sigma} \\ \mathbf{t} & \text{otherwise} \end{cases}$$

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`.

$$\begin{array}{ll}
\mathbf{ArgT}(\mathcal{P}, vt, f, x, s) & \\
& \textbf{assert } \mathcal{P} \sqcup vt \sqsubseteq sink\ (f(x)) \\
& let\ vt_1 := vt - \{\texttt{santize(src)}\}\ in \\
& let\ vt_2 := vt_1 \sqcup vtaint\ \{f(x)\}\ in \\
\boldsymbol{vt'} \longleftarrow & vt_2 \\
\boldsymbol{\mathcal{P}'} \longleftarrow & \mathcal{P} \\
\overline{\boldsymbol{lt}} \longleftarrow & [N]
\end{array}$$

$$\begin{array}{ll}
\mathbf{RetT}(\mathcal{P}_{CLE}, \mathcal{P}_{CLR}, vt, f) & \\
& \textbf{assert } \mathcal{P} \sqcup vt \subseteq sink\ (f.ret) \\
& let\ vt_1 := vt - \{\sigma \mid \sigma / f.ret \in I\}\ in \\
& let\ vt_2 := vt_1 \sqcup tainted\ \{f.ret\}\ in \\
\boldsymbol{vt'} \longleftarrow & vt_2 \\
\boldsymbol{\mathcal{P}'} \longleftarrow & \mathcal{P}
\end{array}$$

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.

$$\begin{array}{ll}
\mathbf{MallocT}(\mathcal{P}, vt) & \\
\boldsymbol{pt} \longleftarrow & \mathcal{P} \sqcup vtaint\ \emptyset \\
\boxed{\boldsymbol{vt}} \longleftarrow & vtaint\ \emptyset \\
\boxed{\overline{\boldsymbol{lt}}} \longleftarrow & [sink\ f.m] \\
\boldsymbol{\mathcal{P}'} \longleftarrow & \mathcal{P}
\end{array}$$

$$\begin{array}{ll}
\mathbf{GlobalT}(id, s) & \\
\boldsymbol{pt} \longleftarrow & vtaint\ \emptyset \\
\boldsymbol{vt} \longleftarrow & vtaint\ \{id\} \\
\overline{\boldsymbol{lt}} \longleftarrow & [sink\ id]
\end{array}$$

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.]

$$\begin{array}{ll}
\mathbf{SplitT}(\mathcal{P}, vt, \boxed{L}) & \\
& let\ pc\ \overline{(L, \sigma)} := \mathcal{P}\ in \\
& let\ vtaint\ \sigma := vt\ in \\
\boldsymbol{\mathcal{P}'} \longleftarrow & pc\ (\overline{(L, \sigma)} \cup (L, \sigma))
\end{array}$$

$$\begin{array}{ll}
\mathbf{LabelT}(\mathcal{P}, L) & \\
& let\ pc\ \overline{(L, \sigma)} := \mathcal{P}\ in \\
\boldsymbol{\mathcal{P}'} \longleftarrow & pc\ \{(L', \sigma) \mid (L', \sigma) \in \overline{(L, \sigma)} \land L \neq L'\}
\end{array}$$

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. 7, 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. But importantly, in this case, these tags are mutable; in a policy that can be expected to take advantage of their mutability, we will need an extra store to set the tag for later.

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 lowre when it is reached.

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 FUTURE WORK

We have presented the language and a reference interpreter, built on top of the CompCert interpreter [Leroy 2009], 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

Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. 2008. Termination-Insensitive Noninterference Leaks More Than Just a Bit. In *Computer Security - ESORICS 2008*, Sushil Jajodia and Javier Lopez (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 333–348.

Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (feb 2010), 66–75. https://doi.org/10.1145/1646353.1646374

CHR Chhak, Andrew Tolmach, and Sean Anderson. 2021. Towards Formally Verified Compilation of Tag-Based Policy Enforcement. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Virtual, Denmark) *(CPP 2021)*. Association for Computing Machinery, New York, NY, USA, 137–151. https://doi.org/10.1145/3437992.3439929

Dorothy E. Denning and Peter J. Denning. 1977. Certification of Programs for Secure Information Flow. *Commun. ACM* 20, 7 (jul 1977), 504–513. https://doi.org/10.1145/359636.359712

Udit Dhawan, Nikos Vasilakis, Raphael Rubin, Silviu Chiricescu, Jonathan M Smith, Thomas F Knight Jr., Benjamin C Pierce, and André DeHon. 2014. PUMP: A Programmable Unit for Metadata Processing. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy (HASP '14)*. ACM, New York, NY, USA, 8:1–8:8. https://doi.org/10.1145/2611765.2611773

The Apache Software Foundation. 1997. *Apache HTTP Server Project.* Retrieved April 11, 2023 from https://httpd.apache.org/

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (jul 2009), 107–115. https://doi.org/10.1145/1538788.1538814

]Munoz:PoweredbyC Daniel Munoz. [n. d.]. *After All These Years, the World is Still Powered by C Programming.* Retrieved April 11, 2023 from https://www.toptal.com/c/after-all-these-years-the-world-is-still-powered-by-c-programming

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

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. 2020. Cornucopia: Temporal Safety for CHERI Heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*. 608–625. https://doi.org/10.1109/SP40000.2020.00098

A CONTINUATIONS

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

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 $\mathbf{undef}@vt@\overline{lt}$ for some vt and $\overline{lt}$, unless given an initializer. Let m_0 and ge_0 be the initial (empty) memory and environment. The parameter b marks the start of the global region.

$$globals\ xs\ b = \begin{cases} (m_0, ge_0) & \text{if } xs = \varepsilon \\ (m[p \ldots p + |ty| \mapsto \mathbf{undef}@vt@\overline{lt}]_{|ty|}, & \text{if } xs = (id, ty) :: xs' \\ \ ge[id \mapsto (p, p + |ty|, ty, pt)]) & \text{and } pt, vt, \overline{lt} \leftarrow \mathbf{GlobalT}(id, s) \\ & \text{where } (m, ge) = globals\ xs'\ (b + |ty|) \end{cases}$$

C STEP RULES

C.1 Sequencing rules

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

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

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

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

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

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

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

C.2 Conditional rules

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

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

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

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

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

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

C.3 Loop rules

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{context}\ k\ C[e] ::=} \\
\quad |\mathit{context}\ k\ \lambda e.e & \\
\quad |\mathit{context}\ \textsc{lh}\ \lambda e.\mathit{Ederef}\ C[e] & \text{where } \mathit{context}\ \textsc{rh}\ C[e] \\
\quad |\mathit{context}\ \textsc{lh}\ \lambda e.\mathit{Efield}\ C[e]\ \ \mathit{id} & \text{where } \mathit{context}\ \textsc{rh}\ C[e] \\
\quad |\mathit{context}\ \textsc{rh}\ \lambda e.\mathit{EvalOf}\ C[e] & \text{where } \mathit{context}\ \textsc{lh}\ C[e] \\
\quad |\mathit{context}\ \textsc{rh}\ \lambda e.\mathit{EaddrOf}\ C[e] & \text{where } \mathit{context}\ \textsc{lh}\ C[e] \\
\quad |\mathit{context}\ \textsc{rh}\ \lambda e.\mathit{Eunop}\ \odot\ C[e] & \text{where } \mathit{context}\ \textsc{rh}\ C[e] \\
\quad |\mathit{context}\ \textsc{rh}\ \lambda e.\mathit{Ebinop}\ \oplus\ C[e_1]\ \ e_2 & \text{where } \mathit{context}\ \textsc{rh}\ C[e_1] \\
\quad |\mathit{context}\ \textsc{rh}\ \lambda e.\mathit{Ebinop}\ \oplus\ e_1\ C[e_2] & \text{where } \mathit{context}\ \textsc{rh}\ C[e_2] \\
\quad |\mathit{context}\ \textsc{rh}\ \lambda e.\mathit{Ecast}\ C[e]\ \ \mathit{ty} & \text{where } \mathit{context}\ \textsc{rh}\ C[e] \\
\quad |\mathit{context}\ \textsc{rh}\ \lambda e.\mathit{EseqAnd}\ C[e_1]\ \ e_2 & \text{where } \mathit{context}\ \textsc{rh}\ C[e_1] \\
\quad |\mathit{context}\ \textsc{rh}\ \lambda e.\mathit{EseqOr}\ C[e_1]\ \ e_2 & \text{where } \mathit{context}\ \textsc{rh}\ C[e_1] \\
\quad |\mathit{context}\ \textsc{rh}\ \lambda e.\mathit{Econd}\ C[e_1]\ \ e_2\ e_3 & \text{where } \mathit{context}\ \textsc{rh}\ C[e_1] \\
\quad |\mathit{context}\ \textsc{rh}\ \lambda e.\mathit{Eassign}\ C[e_1]\ \ e_2 & \text{where } \mathit{context}\ \textsc{lh}\ C[e_1] \\
\quad |\mathit{context}\ \textsc{rh}\ \lambda e.\mathit{Eassign}\ e_1\ C[e_2] & \text{where } \mathit{context}\ \textsc{rh}\ C[e_2] \\
\quad |\mathit{context}\ \textsc{rh}\ \lambda e.\mathit{EassignOp}\ \oplus\ C[e_1]\ \ e_2 & \text{where } \mathit{context}\ \textsc{lh}\ C[e_1] \\
\quad |\mathit{context}\ \textsc{rh}\ \lambda e.\mathit{EassignOp}\ \oplus\ e_1\ C[e_2] & \text{where } \mathit{context}\ \textsc{rh}\ C[e_2] \\
\quad |\mathit{context}\ \textsc{rh}\ \lambda e.\mathit{EpostInc}\ \oplus\ C[e] & \text{where } \mathit{context}\ \textsc{lh}\ C[e] \\
\quad |\mathit{context}\ \textsc{rh}\ \lambda e.\mathit{Ecall}\ C[e_1]\ (\overline{e_2}) & \text{where } \mathit{context}\ \textsc{rh}\ C[e_1] \\
\quad |\mathit{context}\ \textsc{rh}\ \lambda e.\mathit{Ecall}\ e_1(C[\overline{e_2}]) & \text{where } \mathit{context}\ \textsc{rh}\ C[e]\ \text{ for } e \in \overline{e_2} \\
\quad |\mathit{context}\ \textsc{rh}\ \lambda e.\mathit{Ecomma}\ C[e_1]\ \ e_2 & \text{where } \mathit{context}\ \textsc{rh}\ C[e_1] \\
\quad |\mathit{context}\ \textsc{rh}\ \lambda e.\mathit{Eparen}\ C[e]\ \ \mathit{ty} & \text{where } \mathit{context}\ \textsc{rh}\ C[e] \\
\quad |\mathit{context}\ \textsc{rh}\ \lambda e.\mathit{Eparen}\ C[e]\ \ \mathit{ty}\ \mathit{t} & \text{where } \mathit{context}\ \textsc{rh}\ C[e]
\end{array}
$$

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

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

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

C.5 Expression Rules

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

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

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

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

$$\frac{ty = TStruct\ id \vee ty = TUnion\ id \quad offset\ id\ fld = \delta \quad pt' \leftarrow \textbf{FieldT}(pt, id)}{Efield\ (Eval\ p@pt : ty)\ fld) \Rightarrow_{\text{LH}} Eloc\ (p + \delta)@pt'}$$

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

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

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

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

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

$$\frac{\begin{array}{cc} ty_1 = ptr\ ty_1' & \neg isptr(ty_2) \\ m[v]_{|ty_1'|} = \_@vt@\overline{lt} & vt \leftarrow \textbf{PICastT}(\mathcal{P}, pt, \boxed{vt, \overline{lt}}) \end{array}}{(\mathcal{P}, m, Ecast\ (Eval\ v@pt : ty_1)\ ty_2) \Rightarrow_{\text{RH}} (\mathcal{P}, m, Eval\ v@vt' : ty_2)}$$

$$\frac{\begin{array}{cc} \neg isptr(ty_1) & ty_2 = ptr\ ty_2' \\ m[v]_{|ty_2'|} = \_@vt_2@\overline{lt} & pt \leftarrow \textbf{IPCastT}(\mathcal{P}, vt_1, \boxed{vt_2, \overline{lt}}) \end{array}}{(\mathcal{P}, m, Ecast\ (Eval\ v@vt_1 : ty_1)\ ty_2) \Rightarrow_{\text{RH}} (\mathcal{P}, m, Eval\ v@pt : ty_2)}$$

$$\frac{\begin{array}{cc} ty_1 = ptr\ ty_1' & ty_2 = ptr\ ty_2' \\ m[v]_{|ty_1'|} = m[v]_{|ty_2'|} = \_@vt@\overline{lt} & pt' \leftarrow \textbf{PPCastT}(\mathcal{P}, pt, \boxed{vt, \overline{lt}}) \end{array}}{(\mathcal{P}, m, Ecast\ (Eval\ v@pt : ty_1)\ ty_2) \Rightarrow_{\text{RH}} (\mathcal{P}, m, Eval\ v@pt' : ty_2)}$$

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

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

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

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

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

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

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

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

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

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

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.

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

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

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

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

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

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

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

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

And finally, we have the return rules.

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

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

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