Policies

ANONYMOUS AUTHOR(S)

1 INTRODUCTION

Today's computing infrastructure is built atop layers of legacy C code, much of it distressingly insecure and difficult to maintain. These foundations need be shored up with additional security enforcement, but such mechanisms vary widely in their security goals and carry nuanced trade-offs between permissiveness and power. Enter 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.*

Legacy codebases, especially C codebases, pose a security conundrum. They are difficult to modify, the original programmers having moved on, so 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. They may also contain undefined behavior (UB), possibly being used intentionally as a low-level idiom, while most static analyses treat UB as a failure. 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) realistic, permissive memory models from the literature (PVI and PNVI), (2) Secure Information Flow (SIF), and (3) compartmentalization

In the next section, we give a full account of the formal semantics of Tagged C, including its control points. Then in section 4, 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 5, we give a similar description of a secure information flow policy. We round out our policies in section 6 with a compartmentalization policy. In **??** we discuss the degree to which the design meets our goals of flexibility and applicability to realistic security concerns.

1.1 Background

Exploits. [Talk about how there are a lot of exploits, they're problems, etc.]

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

Hardware Reference Monitors. The concept of a reference monitor has been around for a long time, initially as a theoretical construct [TODO: cite the security automaton paper]. Recently, several hardware implementations have cropped up, all tag-based. Arm MTE is specifically designed to enforce a narrow form of memory safety. It gives each address and register a 4-bit tag, which by default is used to distinguish adjacent objects in memory from one another, preventing buffer overflows, but not necessarily other kinds of memory safety violations. [TODO: read the Binghamton paper, figure out where they sit here.]

Programmable Interlocks for Policy Enforcement (PIPE) [TODO:cite] is a hardware extension that uses large (word-sized in the underlying ISA) tags. The size of its tags gives it the flexibility to implement complex policies with structured tags, and even run multiple policies at once. PIPE is therefore our primary target for Tagged C.

While it is not a reference monitor, no discussion of hardware enforcement mechanisms would be complete without a discussion of CHERI, a dedicated capability machine. 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 [TODO: cite Cornucopia], stack safety [TODO: StkTokens], 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.

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 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 2.

$$\odot ::= \; ! \mid \sim \mid \text{-} \mid abs \qquad \oplus ::= + \mid - \mid \times \mid \div \mid \% \mid \ll \mid \gg \mid \& \mid | \mid \wedge$$

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

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

Fig. 1. Tagged C Abstract Syntax

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.

3 FORMAL SEMANTICS

Tagged C uses a small-step reduction semantics, given in full in the appendix.

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, and $\mathcal{P}$ 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 B.

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, and a load or store will access some number of bytes. We write $m[l]_s = v@vt@\overline{lt}$ to denote loading s bytes, starting at location l, and interpreting them as a value

Name	Inputs	Outputs	Control Points
GlobalT	$id \in ident, s \in \mathbb{N}$	$pt, vt, \overline{lt}$	Program initialization
LocalT	$\mathcal{P}, id \in ident, s \in \mathbb{N}$	$pt, vt, \overline{lt}$	Call
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	$\mathcal{P}, vt$	vt	UnOp
BinopT	$\mathcal{P}, vt_1, vt_2$	vt'	BinOp
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	$\mathcal{P}, pt, \boxed{vt, \overline{lt}}$	$\mathcal{P}', vt$	Cast from pointer to scalar
IPCastT	$\mathcal{P}, vt_1, \boxed{vt_2, \overline{lt}}$	$\mathcal{P}', pt$	Cast from scalar to pointer
PPCastT	$\mathcal{P}, pt, \boxed{vt, \overline{lt}}$	$\mathcal{P}', pt'$	Cast between pointers
IICastT	$\mathcal{P}, vt_1$	$\mathcal{P}', pt$	Cast between scalars
SplitT	$\mathcal{P}, vt, \boxed{L}$	$\mathcal{P}'$	Split points (**??**)
JointT	$\mathcal{P}, \boxed{L}$	$\mathcal{P}'$	Label
ArgT	$\mathcal{P}, vt, f, x$	vt'	Call
CallerRetT	$\mathcal{P}, \mathcal{P}', vt$	vt'	Return
CalleeRetT	$\mathcal{P}, \mathcal{P}', vt$	vt'	Return

v, a value tag vt, and a list of s location tags. Likewise, $m[l \mapsto v@vt@\overline{lt}]_s$ denotes storing that many bytes. vt is tied to a full value, which may consist of multiple bytes, while each tag in $\overline{lt}$ is tied to an individual byte. When writing multiple contiguous values, we will write a range of locations. So in the case of an array of 10 integers, s would be 4, and $m[l \ldots l + 10 \mapsto v@vt@\overline{lt}]_4$ would write $v@vt$ to ten words starting at l, with the four bytes of each word tagged with $\overline{lt}$'s four tags. This guarantees that even misaligned loads and stores always have a valid location tag to check (possibly multiple, mismatched location tags, in which case the policy can failstop if needed.)

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.

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

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

Control Points with Side-effects and Optional Arguments. Most control points can be mapped cleanly onto one or more instructions in a compiled program. For example, the **BinopT** control point takes as input the tags on the parameters of an operation (as well as the PC tag) and yields a tag for the result, so the target-level rule, which does the same, can be identical. Other control points may correspond to multiple target-level instructions, requiring a more complicated mapping. We will not call these out unless they are particularly noteworthy. From a performance standpoint, 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, which must update tags in locations that are not written, or in which the source construct does not have corresponding instructions in the target.

These situations require the compiler to add instructions to manipulate tags. If the tag rules that instantiate those control points do not make use of them, these instructions are needless overhead. In these cases, the control points will take optional parameters or return optional results, and we will explain how the rule should be implemented in the target if the options are used. If compiling with a known policy that does not make use of the options, it will be sound to eliminate the extra instructions. If *all* of the control point's outputs are optional and unused, the control point need not be compiled at all. In this document, optional inputs and outputs will be 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

3.1 Writing Policies

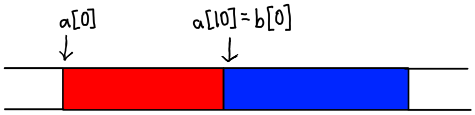
We currently write policies by embedding them directly in Coq. That is not going to change in the near future, but we should tell a story about how we envision it being done in general, in a production version.

4 ENFORCING MEMORY SAFETY

Memory safety policies operate on under a "lock and key" model, in which objects in memory are tagged with a unique identifier (the "lock") and may only be accessed via a pointer tagged with the same identifier (the "key.") For a simple example, consider the following code:

```
void main() {
  int a[10];
  int b[10];
  a[10] = 42;
}
```

In a typical stack allocator—such as the one used by my interpreter—`a` and `b` will be allocated next to one another on the stack, like this:



To prevent the expression `a[10] = 42` from overwriting `b[0]`, we give `a` and `b` unique *color tags* when they are allocated. In this case, we'll tag `a` with *dyn 0*, indicating that it's the first dynamically allocated object, and `b` with *dyn 1*. Then, when we evaluate the left-hand expression `a` into its memory location l, we tag l with *dyn 0*. When we take the offset $l + 10$, we keep that tag. And when we perform the assignment, we check that the location tag at l matches. It doesn't, so we failstop.

The same principle applies for this code:

```
void main() {
  int* a = malloc(10 * sizeof(int));
  int* b = malloc(10 * sizeof(int));
  *(a + (b - a)) = 42;
}
```

In this case, a and b could be allocated anywhere in the heap, and in Tagged C the expression *(a + (b - a)) = 42 will always write to *b. While this might be intentional on the part of the programmer, it is also undefined behavior in the C standard, and in some (but not all; see below) formal C semantics. Likewise, if a and b are next to each other or in some other predictable arrangement, arithmetic like our first example can apply. The memory safety policy works just the same in this scenario, with the tags being attached by the call to malloc, once again using the *dyn* label in a global count of allocated blocks. Meanwhile, values that are not derived from valid pointers at all are tagged *X*, and can never be read or written through, to avoid pointer forging, like this:

```
void main() {
  int* a = malloc(10 * sizeof(int));
  // We happen to know that a will be at address 1000
  *1000 = 42;
}
```

Both stack and heap allocations use the *dyn* label and have a color that can grow arbitrarily high. This is because over a program's execution, it might allocate an unbounded number of heap- or stack-allocated objects, and each needs a unique identifier. Existing work has shown that in practice, tag colors can be "garbage collected" and reused, but in Tagged C we assume them to be infinite and unique.

Lastly, we have global variables. While "global safety" is not as prominent a topic as heap or stack safety, overrunning a global buffer is still a problem. It is also easy to forge a pointer to a global, and when this happens it can undermine assumptions about the behavior of linked libraries whose globals are not exported. Globals do not need dynamic colors, but can use their identifiers as tags, of the form *glob id*.

Memory Safety: PVI and PNVI. Our policies aim to enforce two memory models in particular: *PVI* (provenanace via integer) and *PNVI* (provenance not via integer) from Memarian et al. [?]. They propose PVI and PNVI as memory models that support common idioms that are undefined in the C standard, but are still restrictive enough as to support useful alias analysis for optimization. This application is orthogonal to security, and violations of either memory model are treated as undefined behavior, just as in the C standard. Our goal is to turn that UB into failstop behavior, so that undefined programs cannot accidentally undermine their own security.

Both memory models represent pointers as integers, just as Tagged C does, with additional provenance associated with each object. An integer cast to a pointer in PVI retains this provenance, enabling integer operations to be performed on it prior to it being cast back to a pointer. 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 2.

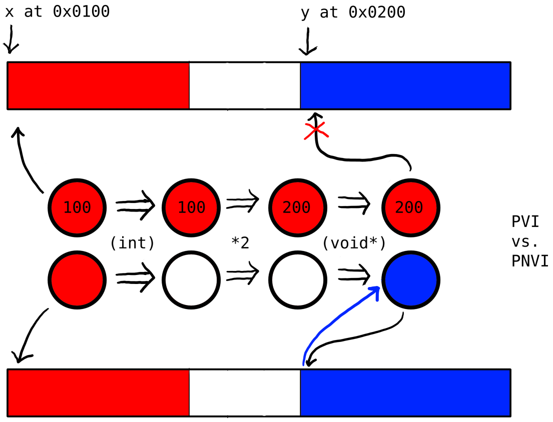


Fig. 2. Integer-pointer casts in PVI and PNVI

We will aim to prove that for any program, if it is run in both the PVI semantics and in Tagged C with our PVI policy, it either produces identical output, or it is both undefined in the PVI semantics and failstops in Tagged C. Likewise for PNVI, except that some UB in PNVI is non-deterministic, and we only require that it failstop in an execution that would *reach* the UB.

4.1 PVI Definitions

Here we give the relevant tag rules for the PVI policy, and describe the control points that they attach to. We will, for each rule, first give the control point(s) that use it, along with a brief explanation of what the surrounding semantics rules do, and then give the rule. For these policies, all control points appear in expression reduction steps. The machine state consists of the PC tag $\mathcal{P}$, a memory m, the global environment ge, and a local environment le. These are contextual semantics, so each expression is situated in some context ctx.

The core of the PVI policy is the *provenance color*, represented by a natural number.

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

Color generation. New colors are generated when objects are allocated. When exactly that occurs depends on where the object lives. Global variables are a special case: they are allocated during program initialization, before execution begins. As such they do not have a control point per se, but a rule that functions similarly, while being more expressive.

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.

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

$$\begin{array}{rcl} & & \mathbf{GlobalT}(id, s) \\ \boldsymbol{pt} & \longleftarrow & glob\ id \\ \boldsymbol{vt} & \longleftarrow & X \\ \overline{\boldsymbol{lt}} & \longleftarrow & [glob\ id \mid 0 \le i < s] \end{array}$$

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".)

Since allocations occur at runtime, the value and location tags that initialize the allocated memory are optional. They would be realized by initializing the entire allocated object at allocation-time, which adds linear overhead if the object was not otherwise being initialized.

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

In the tag rule, the PC Tag carries the "next" color to be assigned. We mark both the pointer tag (which is stored in the local environment) with that color, along with the location tags on the allocated memory. Then we increment the PC Tag to give the next allocation a unique color.

$$\begin{array}{rcl} & & \mathbf{LocalT}(\mathcal{P}, id, s) \\ \boldsymbol{pt} & \longleftarrow & dyn\ \mathcal{P} \\ \boxed{\boldsymbol{vt}} & \longleftarrow & X \\ \boxed{\overline{\boldsymbol{lt}}} & \longleftarrow & [dyn\ \mathcal{P} \mid 0 \le i < s] \\ \boldsymbol{\mathcal{P}'} & \longleftarrow & \mathcal{P} + 1 \end{array}$$

Heap objects are the most interesting: they are allocated via calls to malloc. In Tagged C, malloc is modeled as an external call to a built-in, so this takes the form of a special case of that expression. Where *heap_alloc* is some allocation function (a parameter of the memory model) that takes a size and a memory and returns an address:

$$\frac{\begin{array}{c} \mathcal{P}', pt, \boxed{vt, \overline{lt}} \leftarrow \mathbf{MallocT}(\mathcal{P}, vt) \qquad m', p \leftarrow heap\_alloc\ size\ m \\ m'' = m'\ [p+i \mapsto (\mathbf{undef}, vt, lt) \mid 0 \le i < s] \end{array}}{\mathcal{E}\ (m \mid ctx\ [malloc(size@t)] \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\ (m'' \mid ctx\ [Eval\ p@pt] \gg k@\mathcal{P}')}$$

And the tag rule is identical to **LocalT**, except that it always treats the allocated object as an array of bytes (making the location tags are always identical.)

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

Color Checking. 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. For instance, in a normal *valof* expression, which accesses a left-hand value:

$$\frac{m[l]_{|ty|} = v@vt@\overline{lt} \qquad\qquad vt' \leftarrow \textbf{LoadT}(\mathcal{P}, pt, vt, \overline{lt})}{\mathcal{E}\ (m \mid ctx\ [EvalOf\ \ Eloc\ l@pt] \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\ (m \mid ctx\ [Eval\ v@vt'] \gg k@\mathcal{P})}$$

We want to both check that the pointer tag matches all of the location tags, and propagate the value tag on the value in memory alongside that value.

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

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”; I follow CompCert 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.

$$\frac{\begin{array}{c} m[l]_{|ty|} = v_1@vt@\overline{lt} \qquad\qquad \oplus \in \{+, -, *, /, \%, <<, >>, \&, \hat{}\ , |\} \\ vt' \leftarrow \textbf{LoadT}(\mathcal{P}, pt, vt, \overline{lt}) \qquad e = Eassign\ Eloc\ l@pt\ Ebinop\ \oplus\ \ Eval\ v_1@vt'\ \ Eval\ v_2@vt_2 \end{array}}{\mathcal{E}\ (m \mid ctx\ [EassignOp\ \oplus\ \ Eloc\ l@pt\ Eval\ v_2@vt_2] \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\ (m \mid ctx\ [e] \gg k@\mathcal{P})}$$

$$\frac{\begin{array}{c} m[l] = v@vt@\overline{lt} \quad \oplus \in \{+, -\} \qquad\qquad vt' \leftarrow \textbf{LoadT}(\mathcal{P}, pt, vt, \overline{lt}) \\ vt \leftarrow \textbf{ConstT}\ e = Ecomma\ Eassign\ Eloc\ l@pt\ Ebinop\ \oplus\ \ Eval\ v@vt'\ \ 1@\textbf{ConstT}\ Eval\ v@vt' \end{array}}{\mathcal{E}\ (m \mid ctx\ [EpostInc\ \oplus\ \ Eloc\ l@pt] \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\ (m \mid ctx\ [e] \gg k@\mathcal{P})}$$

On the flip side, we store values to memory using the *assign* expression:

$$\frac{\begin{array}{c} m[l]_{|ty|} = v_1@vt_1@\overline{lt} \qquad\qquad m' = m[l \mapsto v_2@vt'@\overline{lt}'] \\ \mathcal{P}', vt', \overline{lt}' \leftarrow \textbf{StoreT}(\mathcal{P}, pt, vt_1, vt_2, \overline{lt}) \end{array}}{\mathcal{E}\ (m \mid ctx\ [Eassign\ Eloc\ l@pt\ Eval\ v_2@vt_2] \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\ (m' \mid ctx\ [Eval\ v_2@vt_2] \gg k@\mathcal{P}')}$$

As before, we check that the pointer tag matches the locations tags, and then propagate the value tag (ignoring and overwriting the original value tag.) In addition, we propagate the PC Tag.

$$\begin{array}{rcl} & \multicolumn{2}{l}{\textbf{StoreT}(\mathcal{P}, pt, vt_1, vt_2, \overline{lt})} \\ & & \textbf{assert}\ \forall lt \in \overline{lt}.pt = lt \\ \boldsymbol{\mathcal{P}'} & \longleftarrow & \mathcal{P} \\ \boldsymbol{vt'} & \longleftarrow & vt_2 \\ \overline{\boldsymbol{lt'}} & \longleftarrow & \overline{lt} \end{array}$$

Color Propagation. When a value moves from one location to another, it carries the same tag. We already saw this in the load and store rules: they maintain the relationship between the pointer and its tag. Of note here is the **VarT** control point, which transmits the pointer tag from the environment onto the location expression. In this policy, it propagates the color unchanged.

$$\frac{le[id] = (l, \_, pt, ty) \qquad\qquad pt \leftarrow \textbf{VarT}(\mathcal{P}, vt)}{\mathcal{E}\,(m \mid ctx\,[Evar\ id] \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\,(m \mid ctx\,[Eloc\ l@pt] \gg k@\mathcal{P})}$$

Then the color is propagated via all unary operations and all binary operations where exactly one argument has a color. Performing an operation with two values with color tags (i.e., two cast pointers) clears the tag on the result. It can still be used as an integer, but if cast back to a pointer it will be invalid.

$$\frac{\langle \odot \rangle\, v = v' \qquad\qquad vt' = \textbf{UnopT}(\mathcal{P}, vt)\mathcal{P}\,vt}{\mathcal{E}\,(m \mid ctx\,[Eunop\ \odot\ Eval\ v@vt] \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\,(m \mid ctx\,[Eval\ v'@vt'] \gg k@\mathcal{P})}$$

$$\frac{v_1\,\langle \oplus \rangle\, v_2 = v' \qquad\qquad vt' = \textbf{BinopT}(\mathcal{P}, vt_1, vt_2)\mathcal{P}\,vt_1\,vt_2}{\mathcal{E}\,(m \mid ctx\,[Ebinop\ \oplus\ Eval\ v_1@vt_1\ Eval\ v_2@vt_2] \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\,(m \mid ctx\,[Eval\ v'@vt'] \gg k@\mathcal{P})}$$

$$\begin{array}{rcl} & \multicolumn{2}{l}{\textbf{UnopT}(\mathcal{P}, vt)} \\ \boldsymbol{\mathcal{P}'} & \longleftarrow & \mathcal{P} \\ \boldsymbol{vt'} & \longleftarrow & vt \end{array} \qquad\qquad \begin{array}{rcl} & \multicolumn{2}{l}{\textbf{BinopT}(\mathcal{P}, vt_1, vt_2)} \\ \boldsymbol{\mathcal{P}'} & \longleftarrow & \mathcal{P} \\ \boldsymbol{vt'} & \longleftarrow & \text{case } (vt_1,\ vt_2) \text{ of} \\ & & dyn\ n, X \Rightarrow dyn\ n \\ & & glob\ id, X \Rightarrow glob\ id \\ & & X, t \Rightarrow t \end{array}$$

4.2 PNVI Definitions

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.

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

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

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.

$$\textbf{PICastT}(\mathcal{P}, pt, \boxed{vt, \overline{lt}})$$
$$\begin{aligned} \boldsymbol{\mathcal{P}'} &\longleftarrow \quad \mathcal{P} \\ \boldsymbol{vt} &\longleftarrow \quad X \end{aligned}$$

On the other hand, when casting back to a pointer, we need to check the color of the object that it points to.

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

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.

5 SECURE INFORMATION FLOW

To motivate our next policy, let's consider an erroneous piece of code:

```
void sanitize(src, dst);
char* sql_query(char* query);

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

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

```
   break;
  case 2:
   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;".

We model this as a form of *secure information flow* (SIF), a variant of *information flow control* (IFC), as described in the venerable Denning and Denning [?]. Specifically, this is an *intransitive* SIF setting: 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.

SIF is specified by a set of flow rules between what we will term *sources* and *sinks*. A source σ can be an argument of a function, its return value, or a global. A sink ψ can additionally be the set of heap objects allocated by a given function. We write these as follows:

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

In classic SIF theory, we specify an *information flow policy* (IFPol)—not to be confused with a tag policy—as a relation $\cdot \rightsquigarrow \cdot \in \sigma \times \psi$. However, manually defining such a policy is challenging, especially in an intransitive setting. We envision the IFPol being initially stated in negative terms, with the "no-flow" relation $\not\rightsquigarrow$. That is, we will assume by default that for any source σ and any sink ψ, $\sigma \rightsquigarrow \psi$, unless the user has explicitly declared the contrary.

So, in the above example, the user would declare that name $\not\rightsquigarrow$ sql_query. But, in the case of sanitize, we want it to be the case that name can flow to sql_query only via sanitize. We therefore need to allow the user to declare a *declassification* rule. In general we will write σ/σ' to indicate that σ' supersedes σ: if a value that has been influenced by σ influences σ', we can safely ignore its history with σ. We may write $*/\sigma$ to say that σ declassifies anything.

For example, suppose that in the following code, we want to enforce a no-flow rule between the argument x of f and the global variable z (f.x $\not\rightsquigarrow$ z), and a declassification rule $*/$g.a.

```
 int z;

 int g(int a);
```

```
void f(int x, int y) {
  z = x;                     // violation
  z = x + y;                 // violation
  if(x) z = 1; else z = 0;  // violation
  z = g(x);                  // violation, unless f.x / g.a
}
```

The first three lines of f violate the no-flow relation by storing values derived from x into z. The third line is especially interesting: although x is not stored directly, the value that is stored is conditioned upon it, and can be used to deduce information about the original value. This is termed an *implicit flow*. Finally, in the last line, the value of g(x) depends on x, which is a violation unless it is subject to a declassification rule.

We can therefore define an IFPol as a set of rules of each kind:

$$I \subseteq \{\sigma \not\rightsquigarrow \psi \mid \sigma \neq \psi\} \cup \{\sigma/\sigma' \mid \sigma \neq \sigma'\}$$

We do not need to distinguish between rules that notionally represent "integrity" versus "confidentiality" concerns. The SQL injection example is an instance of integrity, ensuring that an input cannot influence data in an undesired way, but the same concept can be used to prevent data from influencing the program's output inappropriately.

SIF, formally. We can characterize the protection offered by a SIF policy in terms of a *non-interference* property along the lines of Bay and Askarov []. We annotate our transitions with events α, each representing the transmission of a value through a source or sink—possibly several. We write the projection of data relevant to a particular source σ or sink ψ as $\pi_\sigma(\alpha)$ or $\pi_\psi(\alpha)$.

$$\alpha ::=$$

[TODO: add the relevant events to their transitions in the semantics.]

Now, we define the knowledge that an observer monitoring a particular sink can extrapolate about the state of the system as a whole, as a set of states that are consistent with the events it observes. Given some initial state S, this is precisely the set of other initial states that might produce the same trace (or an extension thereof) and that are equivalent.

$$\mathbf{K}(S, \overline{\alpha}, \sigma, \psi) \triangleq \{S' \mid S \sim_\sigma S' \wedge S' \hookrightarrow_\psi \overline{\alpha} \cdot \alpha\}$$

Then, absent any declassification rules, we can define non-interference as holding between σ and ψ if, for any states S_1 and S_2 such that $S_1 \xrightarrow{\overline{\alpha} \cdot \alpha} S_2$, $\mathbf{K}(S_2, \overline{\alpha} \cdot \alpha, \sigma, \psi) \supseteq \mathbf{K}(S_1, \overline{\alpha}, \sigma, \psi)$. That is, every world that was possible before α remains possible after.

In the presence of declassification, we add an exception for the case where α reflects an event that should indeed allow an observer to gain some information. We extend the above definition to talk about all of I.

$$NI_I \triangleq \forall S_1, S_2, \overline{\alpha}, \alpha. \begin{cases} \mathbf{K}(S_2, \overline{\alpha} \cdot \alpha, \sigma, \psi) \supseteq \mathbf{K}(S_1, \overline{\alpha}, \sigma, \psi) & \text{if } \sigma \not\rightsquigarrow \psi \in I \\ \mathbf{K}(S_2, \overline{\alpha} \cdot \alpha, \sigma, \psi) \supseteq \mathbf{K}(S_1, \overline{\alpha}, \sigma \sqcup \sigma', \psi & \text{if } \sigma/\sigma' \in I \wedge \sigma' \rightsquigarrow \psi \in I \end{cases}$$

Tagging SIF. We track the influence of a particular source, or its "taint," through the system in the form of tags on values. A value that is tagged *vtaint* $\overline{\sigma}$ has been influenced by all of the sources in $\overline{\sigma}$. We also define a set of tags that indicate that a particular function argument or the memory location of an object represents a sink that is the target of one or more no-flow rules. If a sink ψ is tagged *forbid* $\overline{\sigma}$, then for all $\sigma \in \overline{\sigma}$, $\sigma \not\rightsquigarrow \psi$ must be in our IFPol. Finally, the PC Tag must carry additional information: when the PC Tag is tainted, it must keep a record of the scope of the taint, in the form of a label. We will explain below how this scope is computed.

$$\begin{aligned} T ::= & \, vtaint\ \overline{\sigma} \\ & forbid\ \overline{\sigma} \\ & pctaint\ \overline{(L, \sigma)} \end{aligned}$$

We define four important operations on tags: *join* ($t_1 \sqcup t_2$), *bounded join* ($t_1[L \rightarrowtail t_2]$), *minus* ($t_1 - t_2$), and *check* ($t_1 \models t_2$), all partial functions.

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

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

$$t - \sigma \triangleq \begin{cases} taint\ (\overline{\sigma}' - \sigma) & \text{if } t = taint\ \overline{\sigma}' \\ \bot & \text{otherwise} \end{cases}$$

$$t_2 \models t_1 \triangleq \begin{cases} \mathbf{t} & \text{if } t_1 = taint\ \overline{\sigma}_1, t_2 = forbid\ \overline{\sigma}_2, \text{ and } \overline{\sigma}_1 \cap \overline{\sigma}_2 = \emptyset \\ \mathbf{f} & \text{if } t_1 = taint\ \overline{\sigma}_1, t_2 = forbid\ \overline{\sigma}_2, \text{ and } \overline{\sigma}_1 \cap \overline{\sigma}_2 \neq \emptyset \\ \bot & \text{otherwise} \end{cases}$$

Tainting and Checking Arguments and Returns. Now we can begin to give our policy, given an arbitrary IFPol I.

A function argument or return value can be either a source or a sink. So, when they are processed by the *call-state* and *return-state* rules, we must both check that the value being passed or returned is not tainted by a forbidden source, and then add the current source to its taint. The call-state rule executes at the beginning of a call, moving all of its arguments into the local environment, using the **ArgT** tag rule. The return-state rule executes after the call returns, inserting the result into the context saved in the continuation. The program counter on return and the result's tag are set by the **CallerRetT** tag rule. Both are given in fig. 3.

$$\begin{array}{ll} \multicolumn{2}{l}{\mathbf{ArgT}(\mathcal{P}, vt, f, x)} \\ & \textit{let } t := \textit{forbid } \{\sigma \mid \sigma \not\rightsquigarrow f(x) \in I\} \textit{ in} \\ & \textbf{assert } t \models \mathcal{P} \sqcup vt \\ & \textit{let } vt_1 := vt - \{\sigma \mid \sigma / f(x) \in I\} \textit{ in} \\ & \textit{let } vt_2 := vt_1 \sqcup \textit{tainted } \{f(x)\} \textit{ in} \\ \boldsymbol{vt'} \longleftarrow & vt_2 \end{array}$$

$$\begin{array}{ll} \multicolumn{2}{l}{\mathbf{CallerRetT}(\mathcal{P}, \mathcal{P}', vt)} \\ & \textit{let } t := \textit{forbid } \{\sigma \mid \sigma \not\rightsquigarrow f.ret \in I\} \textit{ in} \\ & \textbf{assert } t \models \mathcal{P} \sqcup vt \\ & \textit{let } vt_1 := vt - \{\sigma \mid \sigma / f.ret \in I\} \textit{ in} \\ & \textit{let } vt_2 := vt_1 \sqcup \textit{tainted } \{f.ret\} \textit{ in} \\ \boldsymbol{vt'} \longleftarrow & vt_2 \end{array}$$

$$\frac{def(f) = (xs, s) \qquad le' = le[\![x \mapsto v@vt' \mid (x, v@vt) \leftarrow zip(xs, args), vt' \leftarrow \mathbf{ArgT}(\mathcal{P}, vt, f, x)]\!]}{\mathcal{C}\ (f, m, ge \mid le(args) \gg k@\mathcal{P}) \longrightarrow \mathcal{S}\ (m \mid ge \gg le'@\mathcal{P})\ sk}$$

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

Fig. 3. Call and Return Steps

Global variables are also possible sources or sinks. In this case, we initialize their tags to carry this information.

$$\begin{array}{rl} & \mathbf{GlobalT}(id, s) \\ \boldsymbol{pt} \longleftarrow & tainted\ \emptyset \\ \boldsymbol{vt} \longleftarrow & tainted\ \{x\} \\ \overline{\boldsymbol{lt}} \longleftarrow & [forbidden\ \{\sigma \mid x \not\rightsquigarrow x \in I\}] \end{array}$$

Introducing Dynamic Sinks. 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.

$$\begin{array}{rl} & \mathbf{MallocT}(\mathcal{P}, vt) \\ \boldsymbol{pt} \longleftarrow & \mathcal{P} \sqcup tainted\ \emptyset \\ \boxed{\boldsymbol{vt}} \longleftarrow & tainted\ \emptyset \\ \boxed{\overline{\boldsymbol{lt}}} \longleftarrow & [forbidden\ \{\sigma \mid \sigma \not\rightsquigarrow f.m\}] \\ \boldsymbol{\mathcal{P}'} \longleftarrow & \mathcal{P} + 1 \end{array}$$

Propagating Taint Through Expressions. It is simple enough to determine when a value is tainted: at a function call, all function arguments are tagged with their source identity, and the result of any expression is tagged with the union of the sources of its operands. If the expression involves a store or function call itself, we must check the taints on the value being stored or passed against the forbidden list of the target.

Unary and binary operations:

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

Loads and stores:

$$\begin{array}{rl} & \mathbf{LoadT}(\mathcal{P}, pt, vt, \overline{lt}) \\ \boldsymbol{vt'} \longleftarrow & \mathcal{P} \sqcup pt \sqcup vt \end{array} \qquad \begin{array}{rl} & \mathbf{StoreT}(\mathcal{P}, pt, vt_1, vt_2, \overline{lt}) \\ & \mathbf{assert}\ \forall lt \in \overline{lt}.lt \models \mathcal{P} \sqcup pt \sqcup vt_2 \\ \boldsymbol{\mathcal{P}'} \longleftarrow & \mathcal{P} \\ \boldsymbol{vt'} \longleftarrow & \mathcal{P} \sqcup pt \sqcup vt_2 \\ \overline{\boldsymbol{lt'}} \longleftarrow & \overline{lt} \end{array}$$

Implicit Flows. Things become trickier when 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.

To be more specific, consider a statement that contains an expression, $s(e)$, such that when filled in with a tainted value:

$$\mathcal{S}\ (m \mid sEval\ v_1@taint\ \sigma \gg k@\mathcal{P}) \longrightarrow \mathcal{S}\ (m_1 \mid s_1 \gg k_1@\mathcal{P}_1)$$

while

$$\mathcal{S}\ (m \mid sEval\ v_1@taint\ \sigma \gg k@\mathcal{P}) \longrightarrow \mathcal{S}\ (m_2 \mid s_2 \gg k_2@\mathcal{P}_2)$$

and where $s_1 \neq s_2$ or $k_1 \neq k_2$. Taking either step should taint the program state itself! We represent this as a taint on the PC Tag. When the PC Tag is tainted, all stores to memory and all updates to environments must also be tainted until all branches eventually rejoin. 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` X, because that is after the join point, `J`. The same goes for fig. 5, because we are in a *termination-insensitive* setting []. This means that we consider only terminating runs. So, we can guarantee 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 doesn't have such a label, that indicates that there is no join point within the function—once the PC Tag is tainted, it must remain so until a return.

When we step into a conditional or loop, we record its join point on the PC Tag, associated with the sources that are tainted. Then, when we reach the label, we will subtract its sources from the PC Tag at that time. This means that if multiple branches share a join point, their taints will be removed simultaneously.

$$\begin{array}{rl} & \textbf{SplitT}(\mathcal{P}, vt, \boxed{L}) \\ \mathcal{P}' \longleftarrow & \mathcal{P}[L \rightarrowtail vt] \end{array} \qquad \begin{array}{rl} & \textbf{JointT}(\mathcal{P}, \boxed{L}) \\ & \textbf{assert}\ \mathcal{P} = pctaint\ \overline{(L, \sigma)}\} \\ \mathcal{P}' \longleftarrow & pctaint\ \{(L', \sigma) \mid (L', \sigma) \in \overline{(L, \sigma)} \land L \neq L'\} \end{array}$$

The **JoinT** control point applies whenever we reach a labeled statement, seen in fig. 7. The remaining 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. 8, 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

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

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

J:  public2 = 42;

    return public2;
}
```

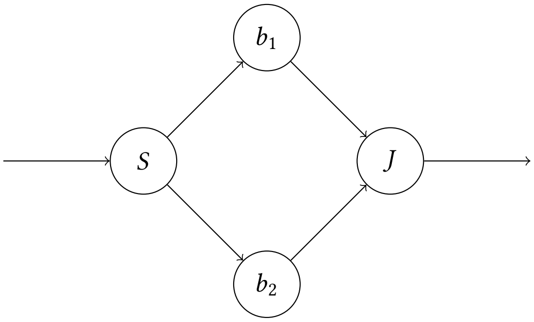


Fig. 4. Leaking via if statements

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

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

J:  public2 = 42;

    return public2;
}
```

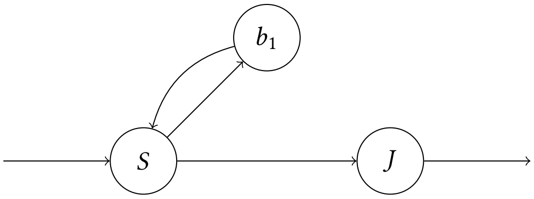


Fig. 5. Leaking via while statements

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

    while (secret) {
        goto b1;
    }

b2: public1 = 1;
    goto J;

b1: public1 = 1;

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

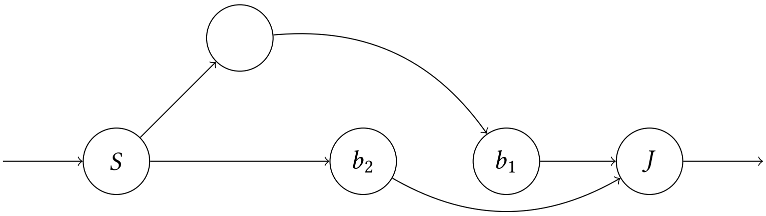


Fig. 6. Cheating with go-tos

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

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

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

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

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

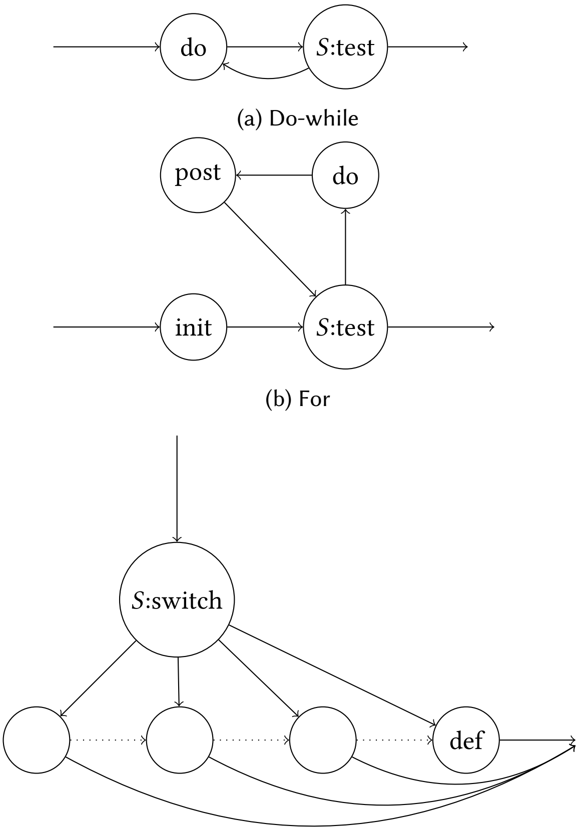
$$\frac{\mathcal{P}' \leftarrow \mathbf{JointT}(\mathcal{P}, \boxed{L})}{\mathcal{S}\,(m \mid L : s \gg k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m \mid ge \gg le'@\mathcal{P}')\ sk}$$

Fig. 7. Selected Conditional Steps

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.



(a) Do-while

(b) For

(c) Switch

Fig. 8. Remaining Branch Statements

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.

6 COMPARTMENTALIZATION

Coverity has examples of: “all the old stuff goes over there, new code has stricter requirements.”

7 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

8 RELATED AND FUTURE WORK

Proofs.

REFERENCES

A CONTINUATIONS

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

B STEP RULES

B.1 Sequencing rules

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

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

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

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

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

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

$$\frac{pop\ k = Kcall\ f'\ ctx\ [\ ]\ k'}{\mathcal{S}\,(m \mid \mathsf{Sreturn}\ Eval\ v@vt \gg k@\mathcal{P}) \longrightarrow \mathcal{R}\,(\mathcal{P}, m, ge \mid Eval\ v@vt \gg ctx\ [@]\ f')\ k}$$

$$\frac{\mathcal{P}' \leftarrow \mathbf{JointT}(\mathcal{P}, \boxed{L})}{\mathcal{S}\,(m \mid L : s \gg k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m \mid ge \gg le'@\mathcal{P}')\ sk}$$

B.2 Conditional rules

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

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

TODO: switch

B.3 Loop rules

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

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

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

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

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

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

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

$$\frac{\mathit{boolof}(v) = \mathbf{f} \qquad k = \mathit{KdoWhileTest}(e)\ \{\ s\ \}\ \mathit{join}\ L;\ k'}{\mathcal{E}\,(m \mid \mathit{Eval}\ v@\mathit{vt} \gg k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m \mid \mathtt{Sskip} \gg k'@\mathcal{P}')}$$

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

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

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

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

$$\frac{\mathit{boolof}(v) = \mathbf{f}}{\mathcal{E}\,(m \mid \mathit{Eval}\ v@\mathit{vt} \gg \mathit{Kfor}\ s;\ k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m \mid \mathtt{Sskip} \gg k@\mathcal{P})}$$

$$\frac{s = \mathtt{Sfor}(\mathtt{Sskip}; e; s_2)\ \mathtt{do}\ s_3\ \mathtt{join}\ L \qquad \mathit{boolof}(v) = \mathbf{t}}{\mathcal{E}\,(m \mid \mathit{Eval}\ v@\mathit{vt} \gg \mathit{Kfor}\ s;\ k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m \mid s_3 \gg \mathit{Kfor}\ s;\ k@\mathcal{P})}$$

$$\frac{s = \mathtt{Sfor}(\mathtt{Sskip}; e; s_1)\ \mathtt{do}\ s_2\ \mathtt{join}\ L \qquad s = \mathtt{Sskip} \lor s = \mathtt{Scontinue}}{\mathcal{S}\,(m \mid s \gg \mathit{Kfor}\ s;\ k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m \mid s_1 \gg \mathit{KforPost}\ \mathtt{Sfor}(\mathtt{Sskip}; e; s_1)\ \mathtt{do}\ s_2\ \mathtt{join}\ L;\ k@\mathcal{P})}$$

$$\frac{s = \mathtt{Sfor}(\mathtt{Sskip}; e; s_1)\ \mathtt{do}\ s_2\ \mathtt{join}\ L}{\mathcal{S}\,(m \mid \mathtt{Sbreak} \gg \mathit{Kfor}\ s;\ k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m \mid \mathtt{Sskip} \gg k@\mathcal{P})}$$

$$\frac{s = \texttt{Sfor}(\texttt{Sskip}; e; s_1)\ \texttt{do}\ s_2\ \texttt{join}\ L}{\mathcal{S}\,(m \mid \texttt{Sskip} \gg \mathit{KforPost}\ s;\ k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m \mid s \gg k@\mathcal{P})}$$

B.4 Expression Rules

$$\frac{\begin{array}{c}\mathcal{P}', pt, \boxed{vt, \overline{lt}} \leftarrow \mathbf{MallocT}(\mathcal{P}, vt) \qquad\qquad m', p \leftarrow \mathit{heap\_alloc\ size\ m} \\ m'' = m'\,[p + i \mapsto (\mathbf{undef}, vt, lt) \mid 0 \le i < s]\end{array}}{\mathcal{E}\,(m \mid \mathit{ctx}\,[\mathit{malloc}(\mathit{size}@t)] \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\,(m'' \mid \mathit{ctx}\,[\mathit{Eval}\ p@pt] \gg k@\mathcal{P}')}$$

$$\frac{m[l]_{|ty|} = v@vt@\overline{lt} \qquad\qquad vt' \leftarrow \mathbf{LoadT}(\mathcal{P}, pt, vt, \overline{lt})}{\mathcal{E}\,(m \mid \mathit{ctx}\,[\mathit{EvalOf}\ \mathit{Eloc}\ l@pt] \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\,(m \mid \mathit{ctx}\,[\mathit{Eval}\ v@vt'] \gg k@\mathcal{P})}$$

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

$$\frac{\begin{array}{c}m[l] = v@vt@\overline{lt} \quad \oplus \in \{+, -\} \qquad\qquad vt' \leftarrow \mathbf{LoadT}(\mathcal{P}, pt, vt, \overline{lt}) \\ vt \leftarrow \mathbf{ConstT}\ e = \mathit{Ecomma}\ \mathit{Eassign}\ \mathit{Eloc}\ l@pt\ \mathit{Ebinop}\ \oplus\ \mathit{Eval}\ v@vt'\ 1@\mathbf{ConstT}\ \mathit{Eval}\ v@vt'\end{array}}{\mathcal{E}\,(m \mid \mathit{ctx}\,[\mathit{EpostInc}\ \oplus\ \mathit{Eloc}\ l@pt] \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\,(m \mid \mathit{ctx}\,[e] \gg k@\mathcal{P})}$$

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

$$\frac{le[id] = (l, \_, pt, ty) \qquad\qquad pt \leftarrow \mathbf{VarT}(\mathcal{P}, vt)}{\mathcal{E}\,(m \mid \mathit{ctx}\,[\mathit{Evar}\ id] \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\,(m \mid \mathit{ctx}\,[\mathit{Eloc}\ l@pt] \gg k@\mathcal{P})}$$

$$\frac{\langle\odot\rangle\, v = v' \qquad\qquad vt' = \mathbf{UnopT}(\mathcal{P}, vt)\mathcal{P}\,vt}{\mathcal{E}\,(m \mid \mathit{ctx}\,[\mathit{Eunop}\ \odot\ \mathit{Eval}\ v@vt] \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\,(m \mid \mathit{ctx}\,[\mathit{Eval}\ v'@vt'] \gg k@\mathcal{P})}$$

$$\frac{v_1 \langle\oplus\rangle\, v_2 = v' \qquad\qquad vt' = \mathbf{BinopT}(\mathcal{P}, vt_1, vt_2)\mathcal{P}\,vt_1\,vt_2}{\mathcal{E}\,(m \mid \mathit{ctx}\,[\mathit{Ebinop}\ \oplus\ \mathit{Eval}\ v_1@vt_1\ \mathit{Eval}\ v_2@vt_2] \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\,(m \mid \mathit{ctx}\,[\mathit{Eval}\ v'@vt'] \gg k@\mathcal{P})}$$

$$\frac{}{\mathcal{E}\left(m \mid \mathit{ctx}\,\Big[\mathit{Ecall}\ f'\ \overline{v@vt}\Big] \gg ty@\mathcal{P}\right) k \longrightarrow C\,(f', m, ge \mid le(v@vt) \gg \mathit{Kcall}\ f\ \mathit{ctx}\,[\,]\ k@\mathcal{P})}$$

B.5 Call and Return Rules

$$\frac{\begin{array}{c}\mathit{def}(f) = (xs, s) \\ le' = le[\![x \mapsto v@vt' \mid (x, v@vt) \leftarrow \mathit{zip}(xs, args), vt' \leftarrow \mathbf{ArgT}(\mathcal{P}, vt, f, x)]\!]\end{array}}{C\,(f, m, ge \mid le(args) \gg k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m \mid ge \gg le'@\mathcal{P})\ sk}$$

$$\frac{k = \mathit{Kcall}\ le'\ \mathit{ctx}\ k' \qquad\qquad \mathcal{P}'', vt' \leftarrow \mathbf{CallerRetT}(\mathcal{P}, \mathcal{P}', vt)}{\mathcal{R}\,(m, ge, le \mid \mathit{Eval}\ v@vt \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\,(m \mid \mathit{ctx}[\mathit{Eval}\ v@vt'] \gg k'@\mathcal{P}')}$$