Policies

ANONYMOUS AUTHOR(S)

1 INTRODUCTION

[Motivation goes here]

In this paper we introduce Tagged C, a variant of C with a built-in reference monitor that supports a wide range of user-defined security policies.

Absent a security policy, Tagged C gives definition to C's memory-undefined-behaviors in the form of a concrete memory model with integer pointers. This permissive environment can be restricted using a *memory safety* policy to give a stricter definition in which some memory UBs are defined as failstops.

We describe two additional classes of policy that can be implemented in Tagged C using input from an engineer familiar with the codebase: *secure information flow* (SIF), often called information flow control, and *compartmentalization*. Both are desirable forms of security. Compartmentalization divides the system into components with limited access to one another's memory, helping isolate errors. SIF allows fine-grained control of which function inputs and outputs are allowed to influence other functions and/or memory objects.

Contributions.

- A full formal semantics for Tagged C, formalized in Coq
- 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) based on minimal user annotations, and (3) compartmentalization for fault isolation

Paper Map. In the next section, we give a full account of the formal semantics of Tagged C, including its control points. Then in section 3, 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, we give a similar description of a secure information flow policy. We round out our policies in section 5 with a compartmentalization policy.

In **??** we give a case study of applying Tagged C policies to a real C codebase: the Apache webserver.

2 FORMAL SEMANTICS

Tagged C uses a small-step reduction semantics, given in full in **??**.

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, $v@vt$; 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 inductively:

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

$$
\begin{aligned}
\odot ::= &\ !\\
&\mid \sim\\
&\mid -\\
&\mid abs
\end{aligned}
$$

$$
\begin{aligned}
\oplus ::= &+\\
&\mid -\\
&\mid \times
\end{aligned}
$$

$$
\begin{aligned}
s ::= &\texttt{skip}\\
&\mid e;\\
&\mid s_1\ s_2\\
&\mid \texttt{if}(e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{join}\ L\\
&\mid \texttt{while}(e)\ \texttt{do}\ s\ \texttt{join}\ L\\
&\mid \texttt{do}\ s\ \texttt{while}\ (e)\ \texttt{join}\ L\\
&\mid \texttt{for}(s_1; e; s_2)\ \texttt{do}\ s_3\ \texttt{join}\\
&\mid \texttt{break}\\
&\mid \texttt{continue}\\
&\mid \texttt{return}\\
&\mid \texttt{switch}\ e\ \{\ \overline{(L, s)}\ \}\\
&\mid L\ :\ s\\
&\mid \texttt{goto}\ L
\end{aligned}
$$

$e ::= v@vt$	Value
$\mid x$	Variable
$\mid e[id]$	Index
$\mid \mid e \mid$	R-value of Address
$\mid * e$	Dereference
$\mid \& e$	Address Of
$\mid \odot\ e$	Unary Operator
$\mid e_1 \oplus e_2$	Binary Operator
$\mid (ty)e$	Cast
$\mid e_1 \wedge e_2$	Sequential And
$\mid e_1 \vee e_2$	Sequential Or
$\mid e_1\ ?\ e_2\ :\ e_3$	Conditional
$\mid \texttt{size}(ty)$	Size of Type
$\mid \texttt{align}(ty)$	Alignment of Type
$\mid e_1 := e_2$	Assignment
$\mid e_1\ [\oplus]= e_2$	Operator Assignment
$\mid e \oplus\oplus$	Post-Increment/Decrement
$\mid e_1, e_2$	Expression Sequence
$\mid e_f(\overline{e}_{args})$	Function Call
$\mid \underline{l}@lt$	Memory Location
$\mid (ty)(e)$	Parenthetical Cast

Fig. 1. Syntax

$$
\frac{m, e \longrightarrow_L m', e'}{\mathcal{E}\ (m, ge, le \mid ctk(e); \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\ (m', ge, le \mid ctk(e'); \gg k@\mathcal{P})}
$$

$$
\frac{\mathcal{P}, m, e \longrightarrow_R \mathcal{P}', m', e'}{\mathcal{E}\ (m, ge, le \mid ctk(e); \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\ (m', ge, le \mid ctk(e'); \gg k@\mathcal{P}')}
$$

Fig. 2. Bookkeeping rules

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

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

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

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

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

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

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

Fig. 3. Sequencing rules

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

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

Fig. 4. Conditionals

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. I write $m[l]_s = v@vt@\overline{lt}$ to denote loading *s* bytes, starting at location *l*, and interpreting them as a value *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, I 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.)

Tags are processed by special functions called *tag rules*, which are called at *control points*. Formally, a control point consists of a return type and a list of input types, which are usually tags or lists of tags, but may be optional as described below. The control points and their types are listed in Figure ??.

The tuple of a tag rule's name and its inputs is a *tag continuation* (in the sense of a curried function) ranged over by T or written as a function, such as $\mathbf{LoadT}(\mathcal{P}, pt, vt, \overline{lt})$. We write that the exection of the tag rule succeeds, and bind its results to tag variables, with $\leftarrow$, as $vt' \leftarrow \mathbf{LoadT}(\mathcal{P}, pt, vt, \overline{lt})$.

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

$$\frac{\mathit{boolof}\,(v) = \mathbf{t}}{\mathcal{E}\,(m, ge, le \mid\ v@vt; \gg \mathit{KwhileTest}(e)\ \{\ s\ \}\ \mathit{join}\ L;\ k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m, ge, le \mid s; \gg \mathit{KwhileLoop}(e)\ \{\ s\ \}\ \mathit{join}\ L;\ k(}$$

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

$$\frac{s = \texttt{skip} \lor s = \texttt{continue}}{\mathcal{S}\,(m, ge, le \mid s \gg \mathit{KwhileLoop}(e)\ \{\ s\ \}\ \mathit{join}\ L;\ k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m, ge, le \mid \texttt{while}(e)\ \texttt{do}\ s\ \texttt{join}\ L \gg k@\mathcal{P})}$$

$$\frac{}{\mathcal{S}\,(m, ge, le \mid \texttt{break} \gg \mathit{KwhileLoop}(e)\ \{\ s\ \}\ \mathit{join}\ L;\ k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m, ge, le \mid \texttt{skip} \gg k@\mathcal{P})}$$

$$\frac{s = \texttt{do}\ s\ \texttt{while}\ (e)\ \texttt{join}\ L}{\mathcal{S}\,(m, ge, le \mid s \gg k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m, ge, le \mid s \gg \mathit{KdoWhileLoop}(e)\ \{\ s\ \}\ \mathit{join}\ L;\ k@\mathcal{P})}$$

$$\frac{s' = \texttt{skip} \lor s' = \texttt{continue}}{\mathcal{S}\,(m, ge, le \mid s' \gg \mathit{KdoWhileLoop}(e)\ \{\ s\ \}\ \mathit{join}\ L;\ k@\mathcal{P}) \longrightarrow \mathcal{E}\,(m, ge, le \mid e; \gg \mathit{KdoWhileTest}(e)\ \{\ s\ \}\ \mathit{join}\ L;}$$

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

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

$$\frac{}{\mathcal{S}\,(m, ge, le \mid \texttt{break} \gg \mathit{KdoWhileLoop}(e)\ \{\ s\ \}\ \mathit{join}\ L;\ k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m, ge, le \mid \texttt{skip} \gg k@\mathcal{P})}$$

$$\frac{s = \texttt{for}(s_1; e; s_2)\ \texttt{do}\ s_3\ \texttt{join}\ L \qquad s_1 \neq \texttt{skip}}{\mathcal{S}\,(m, ge, le \mid s \gg k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m, ge, le \mid s_1 \gg \mathit{Kseq}\ \texttt{for}(\texttt{skip}; e; s_2)\ \texttt{do}\ s_3\ \texttt{join}\ L;\ k@\mathcal{P})}$$

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

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

$$\frac{s = \texttt{for}(\texttt{skip}; e; s_2)\ \texttt{do}\ s_3\ \texttt{join}\ L \qquad \mathit{boolof}\,(v) = \mathbf{t}}{\mathcal{E}\,(m, ge, le \mid\ v@vt; \gg \mathit{Kfor}\ s;\ k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m, ge, le \mid s_3 \gg \mathit{Kfor}\ s;\ k@\mathcal{P})}$$

$$\frac{s = \texttt{for}(\texttt{skip}; e; s_1)\ \texttt{do}\ s_2\ \texttt{join}\ L \qquad s = \texttt{skip} \lor s = \texttt{continue}}{\mathcal{S}\,(m, ge, le \mid s \gg \mathit{Kfor}\ s;\ k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m, ge, le \mid s_1 \gg \mathit{KforPost}\ \texttt{for}(\texttt{skip}; e; s_1)\ \texttt{do}\ s_2\ \texttt{join}\ L;\ k@\mathcal{P})}$$

$$\frac{s = \texttt{for}(\texttt{skip}; e; s_1)\ \texttt{do}\ s_2\ \texttt{join}\ L}{\mathcal{S}\,(m, ge, le \mid \texttt{break} \gg \mathit{Kfor}\ s;\ k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m, ge, le \mid \texttt{skip} \gg k@\mathcal{P})}$$

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

Fig. 5. Loops

$$\frac{\begin{array}{c}\mathcal{P}', pt, \boxed{vt}, \boxed{lt} \leftarrow \textbf{MallocT}(\mathcal{P}, vt) \qquad m, p \leftarrow heap\_alloc\ size\ m \\ m'' = m'\ [p+i \mapsto (\textbf{undef}, vt, lt) \mid 0 \le i < s]\end{array}}{\mathcal{E}\,(m, ge, le \mid builtin\ malloc(size@st); \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\,(m'', ge, le \mid p@pt; \gg k@\mathcal{P}')}$$

$$\frac{m[l]_{|ty|} = v@vt@\overline{lt} \qquad vt' \leftarrow \textbf{LoadT}(\mathcal{P}, pt, vt, \overline{lt})}{\mathcal{E}\left(m, ge, le \mid |\underline{l}@pt|; \gg k@\mathcal{P}\right) \longrightarrow \mathcal{E}\left(m, ge, le \mid v@vt'; \gg k@\mathcal{P}\right)}$$

$$\frac{m[l]_{|ty|} = v_1@vt@\overline{lt} \quad \oplus \in \{+,-,*,/,\%,<<,>>,\&,\hat{}\,,|\} \qquad vt' \leftarrow \textbf{LoadT}(\mathcal{P}, pt, vt, \overline{lt})}{\mathcal{E}\left(m, ge, le \mid \underline{l}@pt\ [\oplus]= v_2@vt_2; \gg k@\mathcal{P}\right) \longrightarrow \mathcal{E}\left(m, ge, le \mid \underline{l}@pt := v_1@vt' \oplus v_2@vt_2; \gg k@\mathcal{P}\right)}$$

$$\frac{\begin{array}{cc} m[l] = v@vt@\overline{lt} & \oplus \in \{+,-\} \\ vt' \leftarrow \textbf{LoadT}(\mathcal{P}, pt, vt, \overline{lt}) & vt \leftarrow \textbf{ConstT}\end{array}}{\mathcal{E}\left(m, ge, le \mid \underline{l}@pt \oplus\oplus; \gg k@\mathcal{P}\right) \longrightarrow \mathcal{E}\left(m, ge, le \mid \underline{l}@pt := v@vt' \oplus 1@\textbf{ConstT}, v@vt'; \gg k@\mathcal{P}\right)}$$

$$\frac{\begin{array}{c} m[l]_{|ty|} = v_1@vt_1@\overline{lt} \qquad m' = m[l \mapsto v_2@vt'@\overline{lt}'] \\ \mathcal{P}', vt', \overline{lt}' \leftarrow \textbf{StoreT}(\mathcal{P}, pt, vt_1, vt_2, \overline{lt})\end{array}}{\mathcal{E}\left(m, ge, le \mid \underline{l}@pt := v_2@vt_2; \gg k@\mathcal{P}\right) v_2@vt_2 \longrightarrow \mathcal{E}\,(m', ge, le \mid v_2@vt_2; \gg k@\mathcal{P}')}$$

$$\frac{le[id] = (l, \_, pt, ty) \qquad pt \leftarrow \textbf{VarT}(\mathcal{P}, vt)}{\mathcal{E}\,(m, ge, le \mid id; \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\left(m, ge, le \mid \underline{l}@pt; \gg k@\mathcal{P}\right)}$$

$$\frac{\langle\odot\rangle\, v = v' \qquad vt' = \textbf{UnopT}(\mathcal{P}, vt)\mathcal{P}\, vt}{\mathcal{E}\,(m, ge, le \mid \odot v@vt; \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\,(m, ge, le \mid v'@vt'; \gg k@\mathcal{P})}$$

$$\frac{v_1 \langle\oplus\rangle\, v_2 = v' \qquad vt' = \textbf{BinopT}(\mathcal{P}, vt_1, vt_2)\mathcal{P}\, vt_1\, vt_2}{\mathcal{E}\,(m, ge, le \mid v_1@vt_1 \oplus v_2@vt_2; \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\,(m, ge, le \mid v'@vt'; \gg k@\mathcal{P})}$$

Fig. 6. Expression rules

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

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

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. I 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.

$$
\begin{aligned}
\textbf{LoadT}: &\quad \mathcal{P}, pt, vt, \overline{lt} \rightharpoonup vt' \\
\textbf{StoreT}: &\quad \mathcal{P}, pt, vt_1, vt_2, \overline{lt} \rightharpoonup \mathcal{P}', vt', \overline{lt}' \\
\textbf{ConstT}: &\quad \rightharpoonup vt \\
\textbf{UnopT}: &\quad \mathcal{P}, vt \rightharpoonup vt \\
\textbf{BinopT}: &\quad \mathcal{P}, vt_1, vt_2 \rightharpoonup vt' \\
\textbf{GlobalT}: &\quad id \in ident, s \in \mathbb{N} \rightharpoonup pt, vt, \overline{lt} \\
\textbf{LocalT}: &\quad \mathcal{P}, id \in ident, s \in \mathbb{N} \rightharpoonup pt, vt, \overline{lt} \\
\textbf{VarT}: &\quad \mathcal{P}, vt \rightharpoonup pt \\
\textbf{MallocT}: &\quad \mathcal{P}, vt \rightharpoonup \mathcal{P}', pt, \boxed{vt}, \boxed{lt} \\
\textbf{PICastT}: &\quad \mathcal{P}, pt, \boxed{vt, \overline{lt}} \rightharpoonup \mathcal{P}', vt \\
\textbf{IPCastT}: &\quad \mathcal{P}, vt_1, \boxed{vt_2, \overline{lt}} \rightharpoonup \mathcal{P}', pt \\
\textbf{SplitT}: &\quad \mathcal{P}, vt, \boxed{t} \rightharpoonup \mathcal{P}', t' \\
\textbf{JointT}: &\quad \mathcal{P}, \boxed{t} \rightharpoonup \mathcal{P}', t' \\
\textbf{ArgT}: &\quad \mathcal{P}, vt, x \rightharpoonup vt' \\
\textbf{CallerRetT}: &\quad \mathcal{P}, \mathcal{P}', vt \rightharpoonup vt'
\end{aligned}
$$

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 I 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

2.1 Writing Policies

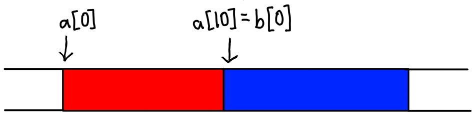
We currently write policies by embedding them directly in Coq. That is not going to change in the near future, but we should tell a story about how we envision it being done in general, in a production version.

3 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));
  // I 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

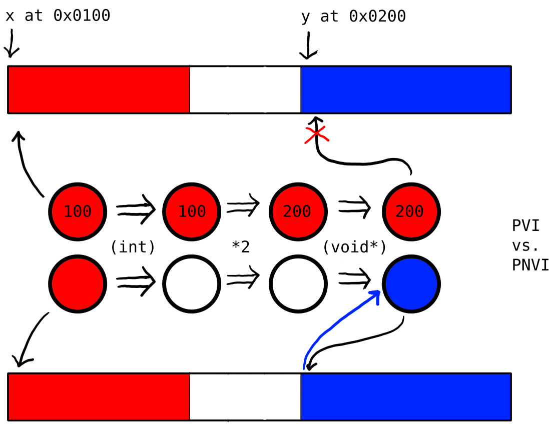


Fig. 7. Integer-pointer casts in PVI and PNVI

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 7.

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.

3.1 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.

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

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.

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

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

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.

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

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:

$$\frac{\mathcal{P}', pt, \boxed{vt}, \boxed{\overline{lt}} \leftarrow \textbf{MallocT}(\mathcal{P}, vt) \qquad m, p \leftarrow heap\_alloc\ size\ m \qquad m'' = m'\ [p + i \mapsto (\textbf{undef}, vt, lt) \mid 0 \le i < s]}{\mathcal{E}\ (m, ge, le \mid builtin\ malloc(size@st); \gg k@\mathcal{P}) \longrightarrow \mathcal{E}\ (m'', ge, le \mid p@pt; \gg k@\mathcal{P}')}$$

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:

$$\frac{m[l]_{|ty|} = v@vt@\overline{lt} \qquad vt' \leftarrow \textbf{LoadT}(\mathcal{P}, pt, vt, \overline{lt})}{\mathcal{E}\ \left(m, ge, le \mid |\underline{l}@pt|; \gg k@\mathcal{P}\right) \longrightarrow \mathcal{E}\ (m, ge, le \mid v@vt'; \gg k@\mathcal{P})}$$

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.

$$\frac{m[l]_{|ty|} = v_1@vt@\overline{lt} \quad \oplus \in \{+,-,*,/,\%,<<,>>,\&,^\wedge,|\} \qquad vt' \leftarrow \textbf{LoadT}(\mathcal{P}, pt, vt, \overline{lt})}{\mathcal{E}\left(m, ge, le \mid \underline{l}@pt\ [\oplus]= v_2@vt_2; \gg k@\mathcal{P}\right) \longrightarrow \mathcal{E}\left(m, ge, le \mid \underline{l}@pt := v_1@vt' \oplus v_2@vt_2; \gg k@\mathcal{P}\right)}$$

$$\frac{\begin{array}{ll} m[l] = v@vt@\overline{lt} & \oplus \in \{+,-\} \\ vt' \leftarrow \textbf{LoadT}(\mathcal{P}, pt, vt, \overline{lt}) & vt \leftarrow \textbf{ConstT} \end{array}}{\mathcal{E}\left(m, ge, le \mid \underline{l}@pt \oplus\oplus; \gg k@\mathcal{P}\right) \longrightarrow \mathcal{E}\left(m, ge, le \mid \underline{l}@pt := v@vt' \oplus 1@\textbf{ConstT}, v@vt'; \gg k@\mathcal{P}\right)}$$

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

$$\frac{\begin{array}{c} m[l]_{|ty|} = v_1@vt_1@\overline{lt} \qquad m' = m[l \mapsto v_2@vt'@\overline{lt}'] \\ \mathcal{P}', vt', \overline{lt}' \leftarrow \textbf{StoreT}(\mathcal{P}, pt, vt_1, vt_2, \overline{lt}) \end{array}}{\mathcal{E}\left(m, ge, le \mid \underline{l}@pt := v_2@vt_2; \gg k@\mathcal{P}\right) v_2@vt_2 \longrightarrow \mathcal{E}\left(m', ge, le \mid v_2@vt_2; \gg k@\mathcal{P}'\right)}$$

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.

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

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.

$$\frac{le[id] = (l, \_, pt, ty) \qquad pt \leftarrow \textbf{VarT}(\mathcal{P}, vt)}{\mathcal{E}\left(m, ge, le \mid id; \gg k@\mathcal{P}\right) \longrightarrow \mathcal{E}\left(m, ge, le \mid \underline{l}@pt; \gg k@\mathcal{P}\right)}$$

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.

$$\frac{\langle\odot\rangle\, v = v' \qquad vt' = \textbf{UnopT}(\mathcal{P}, vt)\mathcal{P}\, vt}{\mathcal{E}\left(m, ge, le \mid \odot v@vt; \gg k@\mathcal{P}\right) \longrightarrow \mathcal{E}\left(m, ge, le \mid v'@vt'; \gg k@\mathcal{P}\right)}$$

$$\frac{v_1 \langle\oplus\rangle\, v_2 = v' \qquad vt' = \textbf{BinopT}(\mathcal{P}, vt_1, vt_2)\mathcal{P}\, vt_1\, vt_2}{\mathcal{E}\left(m, ge, le \mid v_1@vt_1 \oplus v_2@vt_2; \gg k@\mathcal{P}\right) \longrightarrow \mathcal{E}\left(m, ge, le \mid v'@vt'; \gg k@\mathcal{P}\right)}$$

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

3.2 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.

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

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

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.

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

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

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

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 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:

$$\begin{array}{rll} \sigma ::= x & & \text{Global} \\ & f(x) & \text{Argument x of f} \\ & f.ret & \text{Return value of f} \end{array}$$

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

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:

$$I \subseteq \{\}$$

[TODO: specification]

Tagging Taint. 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 *taint* $\overline{\sigma}$ has been influnced 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 $\sigma \in \overline{\sigma}$, $\sigma \not\rightsquigarrow \psi$ is in our IFPol.

We identify sinks through static tags on functions and through memory location tags, that encode the "forbidden" sources. So, our tag set is as follows, with X being the default tag that indicates no (interesting) source has tainted the value. Note that our tags may carry multiple sources.

$$\begin{aligned} T ::= & \mathit{taint}\ \overline{\sigma} \\ & \mathit{forbid}\ \overline{\sigma} \\ & X \end{aligned}$$

We define three important operations on tags: *merge* ($\sqcup$), *minus* ($-$), and *check* ($\sqsubseteq$), all partial functions.

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

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

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

Introducing Taint. Each no-flow rule specifies a source that is either a function argument or return value. These attach to the *call-state* and *return-state* rules, respectively. 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.

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

And 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.

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

In the case of an IFC policy, both control points are parameterized by a set of IFC rules, I.

$$\mathbf{ArgT}(\mathcal{P}, vt, x) I \mathcal{P}\, vt\, nt$$
$$\mathit{let}\ \ vt' \ := \ vt - \{\sigma \mid \sigma/nt \in I\}\ \ \mathit{in}$$
$$\mathit{let}\ vt'' := vt' \sqcup \mathit{tainted}\ \{nt \mid nt \not\rightsquigarrow \_ \in I\}\ \mathit{in}$$
$$\boldsymbol{vt'} \longleftarrow \quad vt''$$

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:

$$\mathbf{UnopT}(\mathcal{P}, vt) \qquad\qquad \mathbf{BinopT}(\mathcal{P}, vt_1, vt_2)$$
$$\boldsymbol{vt'} \longleftarrow \quad vt \qquad\qquad \boldsymbol{vt'} \longleftarrow \quad vt_1 \sqcup vt_2$$

Loads and stores:

$$\mathbf{LoadT}(\mathcal{P}, pt, vt, \overline{lt})$$
$$\boldsymbol{vt'} \longleftarrow \quad \mathcal{P} \sqcup pt \sqcup vt$$

$$\mathbf{StoreT}(\mathcal{P}, pt, vt_1, vt_2, \overline{lt})$$
$$\textbf{assert}\ \forall lt \in \overline{lt}. \mathcal{P} \sqcup pt \sqcup vt_2 \sqsubseteq lt$$
$$\boldsymbol{\mathcal{P}'} \longleftarrow \quad \mathcal{P}$$
$$\boldsymbol{vt'} \longleftarrow \quad \mathcal{P} \sqcup pt \sqcup vt_2$$
$$\overline{\boldsymbol{lt'}} \longleftarrow \quad \overline{lt}$$

Implicit Flows. Things become trickier when 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:

$$\mathcal{S}\,(m, ge, le \mid s\mathit{v_1}@\mathit{taint}\ \sigma \gg k@\mathcal{P}) \longrightarrow \mathcal{S}\left(m_1, ge_1, le_1 \mid s_1 \gg k_1@\mathcal{P}_1\right)$$

while

$$\mathcal{S}\,(m, ge, le \mid s\mathit{v_1}@\mathit{taint}\ \sigma \gg k@\mathcal{P}) \longrightarrow \mathcal{S}\left(m_2, ge_2, le_2 \mid s_2 \gg k_2@\mathcal{P}_2\right)$$

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.

This presents a problem if the program counter must remain tainted indefinitely. Fortunately, it is safe to remove the taint if all branches eventually rejoin. We term this 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. 8; `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. 9.

But in the presence of unrestricted go-to statements, a join point may not be local—and sometimes may not exist. Consider fig. 10, 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, we annotate our programs with this information via additional labels. Every statement that branches can carry an optional label indicating its corresponding join point. If it doesn't have such a label, that indicates that there is no join point—once the PC Tag is tainted, it must remain so until a return.

When we step into a conditional or loop, we need to record its join point for later. The join point will always be represented by a label, which is an identifier, so we can simply store it in the local environment.

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

The label carries a record of all sources that might currently taint the program's control flow, but will be safe once that label is reached. So, just as $\mathcal{P}$ is merged with vt to produce $\mathcal{P}''$, $\mathcal{P}'$ will be merged with vt to produce $\mathcal{P}'''$, a new tag for L. Then, when we reach the label L, we will retrieve the stored tag and subtract its sources from the PC Tag at that time.

$$\begin{array}{l} \mathbf{SplitT}(\mathcal{P}, vt, \boxed{t}) \\ \mathcal{P}'' \longleftarrow \mathcal{P} \sqcup vt \\ \boxed{\mathcal{P}'''} \longleftarrow \mathcal{P}' \sqcup vt \end{array} \qquad \begin{array}{l} \mathbf{JointT}(\mathcal{P}, \boxed{t}) \\ \quad \mathit{let}\ \mathcal{P}'' := \mathcal{P} - \{\sigma \mid \sigma'\}\ \mathit{in} \\ \mathcal{P}'' \longleftarrow \mathcal{P} \sqcup vt \\ \boxed{\mathcal{P}'''} \longleftarrow \mathcal{P}' \sqcup vt \end{array}$$

The **JoinT** control point applies whenever we reach a labeled statement, like so:

$$\frac{le[L] = \_@\mathcal{P}' \quad \mathcal{P}'', \mathcal{P}''' \leftarrow \mathbf{JointT}(\mathcal{P}, \boxed{t}) \quad le' = le[L \mapsto \mathbf{undef}@\mathcal{P}''']}{\mathcal{S}\,(m, ge, le \mid L : s \gg k@\mathcal{P}) \longrightarrow \mathcal{S}\,(m, ge, le' \mid s \gg k@\mathcal{P}'')}$$

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

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

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

J:  public2 = 42;

    return public2;
}
```

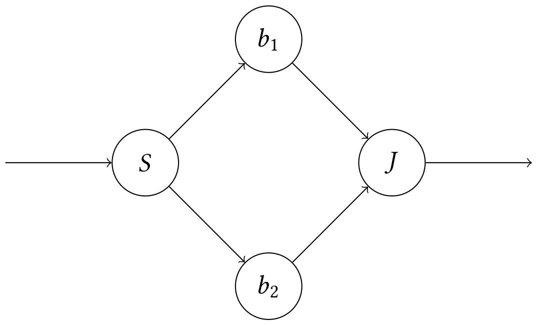


Fig. 8. Leaking via if statements

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

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

J:  public2 = 42;

    return public2;
}
```

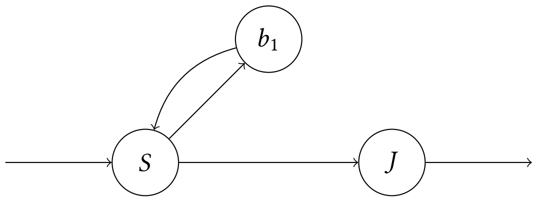


Fig. 9. Leaking via while statements

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

    while (secret) {
        goto b1;
    }

b2: public1 = 1;
    goto J;

b1: public1 = 1;

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

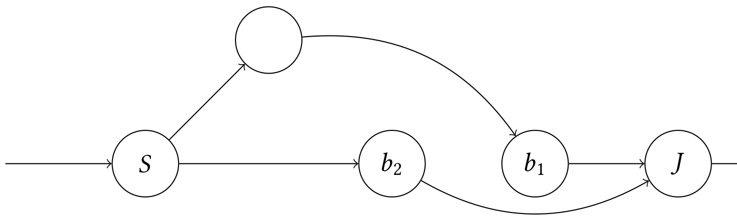


Fig. 10. Cheating with go-tos

semantics in full, it suffices to identify which transitions contain **SplitT** control points. In fig. 11, these are the transitions from the state marked S.

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.

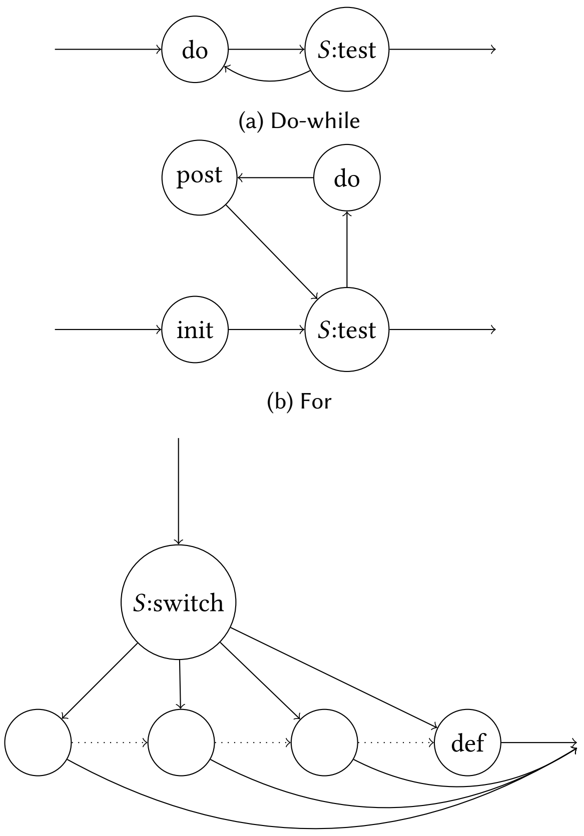
5 COMPARTMENTALIZATION

6 EVALUATION

7 RELATED AND FUTURE WORK

REFERENCES

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



(a) Do-while

(b) For

(c) Switch

Fig. 11. Remaining Branch Statements