Flexible Runtime Security Enforcement with Tagged C

Sean Anderson, Allison Naaktgeboren, and Andrew Tolmach

Portland State University

Abstract. Today's computing infrastructure is built atop layers of legacy C code, often insecure, poorly understood, and/or difficult to maintain. These foundations may be shored up with dynamic security enforcement, which spares legacy code owners from having to modify their code. Tagged C is a C variant with a built-in *tag-based reference monitor* for use in expressing a variety of dynamic security policies and enforcing them with compiler and hardware support.
Tagged C is comprehensive in the policies that it can support. In this paper we will discuss *memory safety*, *compartmentalization*, and *secure information flow* (SIF) policies. It is flexible in covering the tradeoff between conservative policies that may halt too many programs and more permissive ones. And as a source language, it should be more accessible to C programmers than existing assembly-level tag-based reference monitors.

1 Introduction

Many essential technologies rely on new and old C code. Operating systems (Linux, Windows, OSX, BSD), databases (Oracle, sqlite3), the internet & web (Apache, NGNIX, NetBSD, Cisco IOS), the Internet of Things (IoT), and the embedded devices that run our homes and hospitals are built in and on C [9]. C is not a relic; more than a third of professional programmers report active developing in C today [10]. The safety of public and private systems we depend on every day in turn depends on the security of their underlying C codebases. Insecurity might take the form of undefined behavior such as memory errors, of logic errors such as sql injection or other input-sanitization flaws, or of larger-scale architectural flaws that over-provision components of the system with privilege.

Although static analyses can detect and mitigate many C insecurities, the last line of defense against undetected or unfixable vulnerabilities is dynamic enforcement at runtime. Ideally an engineer can tune enforcement to the security needs of the system, rather than apply one-size-fits-all conservative restrictions. To allows developers to define flexible security policies in terms of a familiar source language, we introduce Tagged C: a general-purpose dynamic tag-based enforcement language. Applications of Tagged C include giving precise definition to undefined behaviors (UBs) involving memory, specifying detailed information flow policies, and enforcing arbitrary mandatory access control tables.

Importantly, a security policy can be modified without recompilation unless it is optimized as described in section 5.

The novelty of Tagged C is that it is a *general-purpose* scheme for specifying security policies at the source level, using a tag-based reference monitor. This style of monitor 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, or halts if the program would violate a rule. 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]. While our scheme is general and could be implemented in software, we are motivated by PIPE and aim for compatibility with it as a likely hardware target.

Tagged C consists of an underlying semantics that establishes the baseline concrete behavior of programs with no policies, and a set of *control points* at which the semantics consult a user defined set of *tag rules*. For convenience we build our underlying semantics on the CompCert C semantics, which are formalized as part of the CompCert verified compiler [8]. We provide a reference interpreter also based on that of CompCert, for use executing prototype policies.

Contributions We offer the following contributions:

- A full formal semantic definition for Tagged C, formalized in Coq
- The design of a comprehensive set of *control points* at which the language interfaces with the policy
- A Tagged C interpreter, implemented in Coq and extracted to Ocaml
- Tagged C policies implementing (1) compartmentalization, (2) a realistic, permissive memory model from the literature (PVI), and (3) Secure Information Flow (SIF)

In the next section, we give a high-level introduction of metadata-tagging: how it works, and how its use can improve security. Then in **??**, we briefly discuss the language as a whole, before moving into policies in section 4. Finally, in **??** we discuss the degree to which the design meets our goals of flexibility and applicability to realistic security concerns.

2 What is Metadata Tagging?

Consider a very simple security requirement: "do not leak the password." For simplicity, we will suppose that `pwd` is an integer in this case, and consider a number of ways that it might be leaked (fig. 1). In example (a), we store it in a local variable, then pass it to `printf`. In (b), we perform some arithmetic on it before printing it, but an observer could easily determine the original value. In (c), we store it to a volatile variable representing an mmapped region. And in (d), we loop over it, and store the loop counter in the same mmapped variable, indirectly revealing the value.

To prevent example (a), we need to keep track of the value of the input as it moves from its initial temporary variable to the stack and then is passed, and we need to check the origins of any value we pass to `printf`. Example (b) further requires that we track the provenance of the value as we perform arithmetic on it.

Example (c) reveals that we need to separately track some information about memory locations in addition to the values that they hold, and example (d) asks us to track how the overall state has been influenced by `pwd`.

```
void main(int pwd) {
  int x = pwd;
  printf("%d", x);
}
```

(a)

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

(b)

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

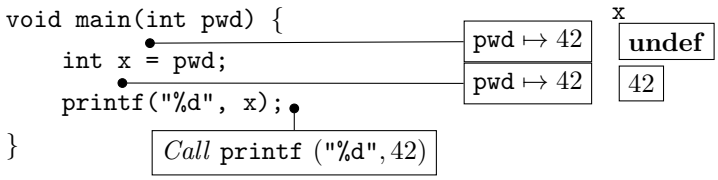
(c)

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

(d)

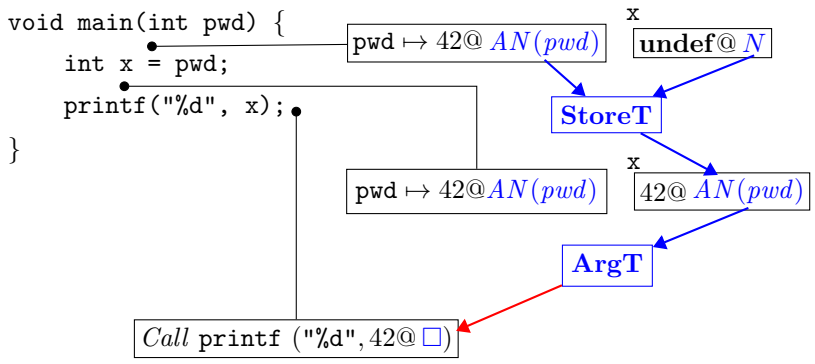
Fig. 1: Examples of leaking `pwd`

Lets examine the state at various points in example 1a. At first, the temporary environment maps `pwd` to 42, and the value of `x` in memory is undefined. We perform the assignment to `x`, and then step to a special *Call* state where we marshall the arguments before stepping into `printf`.



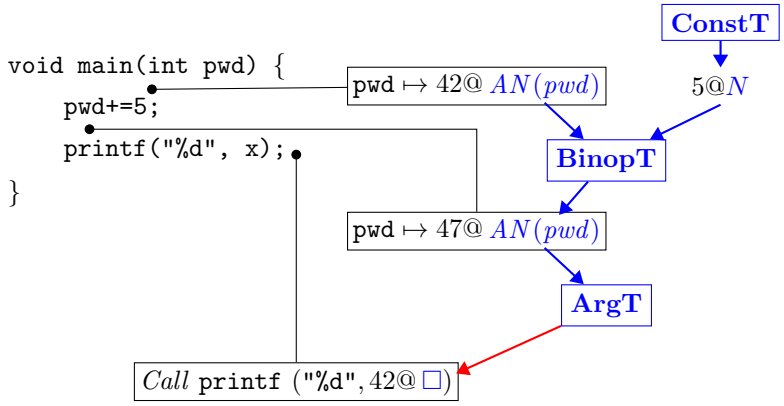
We can dynamically track the confidential status of `pwd` by associating metadata with its value; namely, that it originates in `pwd`. We will write this "tagged" value $42@AN(pwd)$, with the @ symbol denoting a value tagged with metadata. All other values will be tagged N, for "not `pwd`." Then when we copy `pwd` into `x`, we need to come up with a tag for the result. We do so by consulting the *tag rule* **StoreT**, which tells us to transmit $AN(pwd)$. Likewise, when we pass `x`, we consult the **ArgT** rule to determine if it gets a new tag. However, a tag rule may also refuse to proceed, causing the system to failstop. In this case, because we

are trying to pass a value tagged $AN(pwd)$ to an output function, **ArgT** should failstop, indicated by the red arrow.

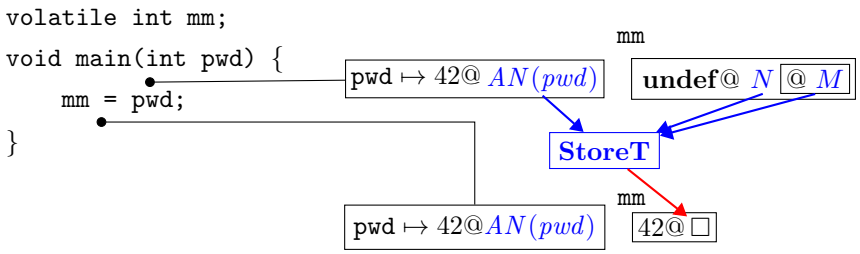


These points at which the tags are checked and either propagated or updated by tag rules are termed *control points*. Collectively, the tag rules form a *policy*.

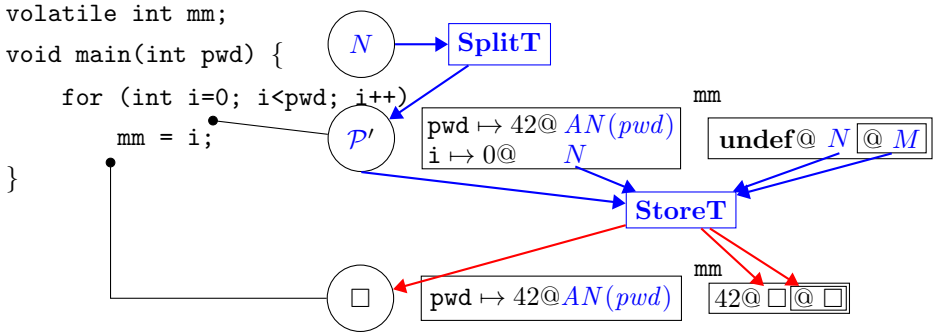
In example 1b, we update `pwd` in place, and the **BinopT** tag rule checks the tags on both it and the constant being added to it. Since one side of the expression is tagged $AN(pwd)$, that must be propagated, even though the other side is N. Once again, **ArgT** will failstop.



Example 1c adds a new wrinkle: some addresses have special behavior associated with them. So we extend our tagging with an additional "location tag" on any memory location. By convention we will write lt for such tags and vt for tags associated with values (pt if the value is a pointer.) In this case we will tag `mm` with a special tag M designated it to be memory-mapped. The **StoreT** tag rule now checks the tag on the value being written against the location tag and will failstop rather than write a $AN(pwd)$ value to an M location.



Finally, we may be concerned with implicit leaks, such as the one in example 1d. In order to prevent this, we must keep track of the privilege of the current context, by carrying an additional tag associated with the global state. This is called the PC Tag, written $\mathcal{P}$. When we perform the check to enter the for loop, `i<pwd`, the resulting value is tagged $AN(pwd)$ (due to the **BinopT** rule above.) We consult the **SplitT** tag rule, and it tells us that $\mathcal{P}$ should be $AN(pwd)$ as well (abbreviated $\mathcal{P}'$ for space.) We extend **StoreT** to check the PC Tag as well as the value being written, and failstop if either is $AN(pwd)$ and the memory location is M.



3 The Language, Informally

Tagged C uses the full syntax of CompCert C [8] with minimal modification (fig. 7). There are two notable syntactical differences in the language, relative to CompCert C: conditionals and loops take an optional *join point* label, and parenthetical expressions an optional "context tag."

Our semantics are a small-step reduction semantics which differ from CompCert C's in two key respects. These are given in full in the appendix. First, Tagged C's semantics contain *control points*: hooks within the operational semantics at which the tag policy is consulted and either tags are updated, or the system failstops. (Control points resemble "advice points" in aspect-oriented programming, but narrowly focused on the manipulation of tags.) A control point consists of the name of a *tag rule* and the bindings of its inputs and outputs; a tag rule is a partial function. The names and signatures of the tag rules, and their corresponding control points, are listed in Table 1.

Second, there is no memory-undefined behavior: the source semantics reflect a concrete target-level view of memory as a flat address space. Without memory safety, programs that exhibit memory-undefined behavior will act as their compiled equivalents would, potentially corrupting memory; we expect that a memory safety policy will be a standard default, but that the strictness of the policy may need to be tuned for programs that use low-level idioms.

The choice of control points and their associations with tag rules, as well as the tag rules' signatures, are a crucial design element. Our proposed design is sufficient for the three classes of policy that we explore in this paper, but it may not be complete.

4 Tags and Policies

Tagged C can enforce a wide range of policies, as follows. A policy consists of a tag type τ, a default tag inhabiting that type, and an instantiation of each tag rule identified in table 1. Tags in gray boxes are optional, as discussed in section 5.

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*. We give name tags to the following constructions and identify them as follows:

- Function identifiers, $FN(f)$
- Function arguments, $AN(f, x)$
- Global variables, $GN(x)$
- Labels, $LN(L)$
- Types, $TN(ty)$

4.1 PVI Memory Safety

Our first policy is a form of memory safety that uses the "provenance via integer" (PVI) memory model of Memarian et al. []. Variations of memory safety have been enforced in PIPE already, but usually using an ad hoc memory model. PVI has the virtue of giving definition to many memory UBs in which a pointer is cast to an integer, subjected to various arithmetic operations, and cast back to a pointer. Their second memory model, *PNVI* (provenance not via integer), is even more permissive. We can also enforce it in Tagged C, though its security value is questionable, and we will not describe it in this paper.

When we say that we want to enforce this specific memory model, we mean that our policy should not failstop on any program that is defined in PVI, and that it should failstop if and when a program reaches undefined behavior. So, in the following examples, we would like `mark` to proceed without failing, while `overflow` should failstop.

Rule Name	Inputs	Outputs	Control Points
LoadT	$\mathcal{P}, pt, vt, \overline{lt}$	vt'	Memory Loads
StoreT	$\mathcal{P}, pt, vt_1, vt_2, \overline{lt}$	$\mathcal{P}', vt', \overline{lt}'$	Memory Stores
UnopT	$\odot, \mathcal{P}, vt$	vt	Unary Operation
BinopT	$\oplus, \mathcal{P}, vt_1, vt_2$	vt'	Binary Operation
ConstT		vt	Applied to Constants/Literals
ExprSplitT	$\mathcal{P}, vt$	$\mathcal{P}'$	Control-flow split points in expressions
ExprJoinT	$\mathcal{P}, \mathcal{P}', vt$	$\mathcal{P}'', vt'$	Join points in expressions
SplitT	$\mathcal{P}, vt, \boxed{L}$	$\mathcal{P}'$	Control-flow split points in statements)
LabelT	$\mathcal{P}, LN(L)$	$\mathcal{P}'$	Labels/arbitrary code points
CallT	$\mathcal{P}, FN(f), FN(f')$	$\mathcal{P}'$	Call
ArgT	$\mathcal{P}, vt, FN(f), AN(x), TN(ty)$	$\mathcal{P}', pt, vt', \overline{lt}$	Call
RetT	$\mathcal{P}_{CLE}, \mathcal{P}_{CLR}, vt, FN(f)$	$\mathcal{P}', vt'$	Return
GlobalT	$GN(x), TN(ty)$	$pt, vt, \overline{lt}$	Program initialization
LocalT	$\mathcal{P}, TN(ty)$	$pt, vt, \overline{lt}$	Call
DeallocT	$\mathcal{P}, TN(ty)$	$vt, \overline{lt}$	Return
ExtCallT	$\mathcal{P}, FN(f), FN(f'), \overline{vt}$	$\mathcal{P}'$	Call to linked code
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`
FieldT	$pt, TN(ty), GN(x)$	pt'	Field Access
PICastT	$\mathcal{P}, pt, \boxed{vt, \overline{lt}}$	vt	Cast from pointer to scalar
IPCastT	$\mathcal{P}, vt_1, \boxed{vt_2, \overline{lt}}$	pt	Cast from scalar to pointer
PPCastT	$\mathcal{P}, pt, \boxed{vt, \overline{lt}}$	pt'	Cast between pointers
IICastT	$\mathcal{P}, vt_1$	pt	Cast between scalars

Table 1: Full list of tag-rules in control points

```
int mark(uintptr ptr) {
  if (!(ptr & 0x00000001))
    *(int*) (ptr | 0x11111110) = 0;

  return ((uintptr) ptr | 0x00000001);
}
```

```
void overflow() {
  int[3] x;
  int y;
  *x = 0;
  *(x+10) = 0;
}
```

Both code snippets are undefined behavior in C, but `mark` is a common low-level idiom seen in, for example, implementations of Cheney's algorithm for garbage collection. Taking advantage of the fact that objects in memory are four-aligned, it uses the lower-order bits of the pointer to store a flag. In this case, it first checks the flag, and if it is not set, it zeroes the memory and then sets it. This is generally considered to be harmless, in contrast to `overflow`, where `y` is being overwritten due to being adjacent in memory to `x`.

We can prevent situations like this using a *memory safety* policy. In brief, whenever an object is allocated, it is assigned a unique "color," and its memory locations as well as its pointer are tagged with that color. Pointers maintain their

tags under arithmetic operations, and loads and stores are legal if the pointer matches the target memory location.

Our set of tags consists of N, for non-pointers, and pointer "colors" $c \in \mathbb{N}$. The PC Tag ($\mathcal{P}$) tracks the next color to allocate, so it's initialized to 0, and everything else is N. N is the default for constants.

Next, the program stores a 0 to address 1000[APT: 92(?)]. The constant 0 takes on the default tag, I[APT: ??]. The policy needs to check that this store is valid, in addition to determining the tags on the value that is stored. This check is performed by comparing the tag on the pointer to the tags on memory—each byte being written, in case the pointer is misaligned.[APT: unclear] Then the tag on the value being stored is propagated with it into memory, unchanged. [APT: explain what StoreT means] [TODO: arrows aren't pointing to quite the right places right now, but one can imagine.]

Next, on the last line, we add 1 to `x`, which invokes the **BinopT** tag rule to combine the tags on the arguments. **BinopT** takes as argument the operation $\oplus$. In memory safety terms, we can add a pointer to a non-pointer in either order, and we can subtract a non-pointer from a pointer (but not the reverse), to yield a pointer to the same object. We can subtract two pointers to the same object from one another to yield a non-pointer, the offset between them. All other binary operations are only permitted between non-pointers.

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

$$\mathcal{P}' \longleftarrow \mathcal{P}$$

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

So, when we try to write through the address 96@1, we compare its tag to that of the memory locations, which are tagged 0. Therefore the policy must failstop on the store.

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

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

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

In our previous memory safety example, we began with the active stack frame already allocated. When it is allocated at runtime, its tags are initialize by the **LocalT** rule. Stack-allocated locals and heap-allocated objects are both tagged with unique colors; the PC Tag carries a counter of the dynamic objects allocated so far, and it is incremented each time.

LocalT$(\mathcal{P}, TN(ty))$

$\mathcal{P}' \leftarrow \mathcal{P} + 1$
$pt \leftarrow dyn\ \mathcal{P}$
$vt \leftarrow N$
$lt \leftarrow [dyn\ \mathcal{P}]$

MallocT$(\mathcal{P}, vt)$

$\mathcal{P}' \leftarrow \mathcal{P} + 1$
$pt \leftarrow dyn\ \mathcal{P}$
$vt \leftarrow N$
$lt \leftarrow [dyn\ \mathcal{P}]$

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

LoadT$(\mathcal{P}, pt, vt, \overline{lt})$

assert $\forall lt \in \overline{lt}.pt = lt$
$vt' \leftarrow vt$

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

assert $\forall lt \in \overline{lt}.pt = lt$
$\mathcal{P}' \leftarrow \mathcal{P}$
$vt' \leftarrow vt_2$
$\overline{lt}' \leftarrow \overline{lt}$

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

UnopT$(\odot, \mathcal{P}, vt)$

$\mathcal{P}' \leftarrow \mathcal{P}$
$vt' \leftarrow vt$

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

$\mathcal{P}' \leftarrow \mathcal{P}$

$$vt' \leftarrow \begin{cases} t & \text{when } (\oplus,\ vt_1,\ vt_2) \text{ is } (\_, t, N) \\ t & \text{is } (\_, N, t) \\ N & \text{is } -, dyn\ n_1, dyn\ n_2 \\ \textbf{fail} & \text{is otherwise} \end{cases}$$

4.2 Compartmentalization

In a perfect world, all C programs would be memory safe. But it is unfortunately common for a codebase to contain undefined behavior that will not be

fixed, including memory undefined behavior. This may occur because developers intentionally use low-level idioms that are UB [**?**]. Or the cost and potential risk of regressions may make it undesirable to fix bugs in older code, as opposed to code under active development that is held to a higher standard [3].

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

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

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

Coarse-grained Protection The core of a compartmentalization scheme is once again memory protection. For the simplest version, we will enforce that memory allocated by a function is only accessible by functions that share its compartment. To do that, we need to keep track of which compartment we're in, using the PC Tag.

Calls and returns each take two steps: first to an intermediate call or return state, and then to the normal execution state, as shown in fig. 2 with some function f calling f'. Three of these steps feature control points. In the initial call step, **CallT** uses the name-tags of the caller and callee to update the PC Tag. Then, in the step from the call state, we place the function arguments in the temp environment, tagging their values with the results of **ArgT**, and we allocate our stack locals, tagging their values and locations with the results of **LocalT**. And on return, **RetT** updates both the PC Tag and the tag on the returned value.

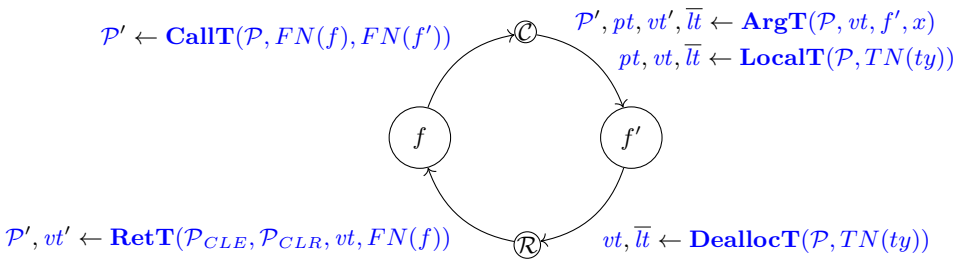


Fig. 2: Structure of a function call

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

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

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

CallT$(\mathcal{P}, FN(f), FN(f'))$

let $C := comp(f')$ *in*

$\mathcal{P}' \longleftarrow \mathcal{P}$

RetT$(\mathcal{P}_{CLE}, \mathcal{P}_{CLR}, vt, FN(f))$

$\mathcal{P}' \longleftarrow \mathcal{P}_{CLR}$

$vt' \longleftarrow vt$

Now that we know which compartment we're in, we can make sure that its memory is protected. This will essentially work just like the basic memory safety policy, except that coarse-grained protection means that the "color" we assign to an allocation is the active compartment. And during a load or store, we compare the memory tags to the PC Tag, not the pointer.

MallocT$(\mathcal{P}, vt)$

$\mathcal{P}' \longleftarrow N$

$pt \longleftarrow N$

$vt \longleftarrow [\mathcal{P}]$

$lt \longleftarrow \mathcal{P}$

LoadT$(\mathcal{P}, pt, vt, \overline{lt})$

assert $\forall lt \in \overline{lt}.\mathcal{P} = lt$

$vt' \longleftarrow vt$

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

assert $\forall lt \in \overline{lt}.lt = \mathcal{P}$

$\mathcal{P}' \longleftarrow \mathcal{P}$

$vt' \longleftarrow N$

$\overline{lt}' \longleftarrow \overline{lt}$

Sharing Memory The above policy works if our compartments only ever communicate by passing non-pointer values. In practice, this is far too restrictive! Many library functions take pointers and operate on memory shared with the caller. External libraries are effectively required for most software to function yet represent a threat. Isolating external libraries from critical code prevents vulnerabilities in the library from compromising critical code and deprives potential attackers of ROP gadgets and other tools if there is an exploit in the critical code.

To allow intentional sharing of memory across compartments, a more flexible policy is needed. Suppose for example the hostname needs to conform to an expected pattern, such as in an enterprise network, to differentiate between different classes of computers (employee, server, contractor, etc). The standard library, over in its own compartment, has helpful functions, provided the caller provides the buffers from which to set or get the hostname.

```
void configure_enterprise(char* intended_name) {
  int ret = 0;
  char curr_name = malloc(HOST_NAME_MAX + 1);
  ret = gethostname( &curr_name, HOST_NAME_MAX + 1 );
```

```
  if (! ret && curr_name != intended_name) { // !ret == (ret==0)
    ret = sethostname(intended_name), strlen(intended_name);
    ....
  }
  ....
}
```

[TODO: update this section for the new example code]

The literature contains two main approaches to this problem: *mandatory access control* and *capabilities*. The former explicitly enumerates the access rights of each compartment, while the latter turns passed pointers into unforgeable tokens of privilege, so that the act of passing one implicitly grants the recipient access.

We can handle a mix of allocations that will be passed and those that will not by creating a variant identifier for `malloc`, `malloc_share`. This identifier maps to the same address (i.e., it is still calling the same function) but its name tag differs and can therefore parameterize the tag rule. The source must have the malloc name changed for every allocation that might be shared. The annotation could be performed manually, or perhaps automatically using some form of escape analysis.

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

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

A capability model resolves both issues by treating shared pointers as tokens of privilege. If a compartment can obtain a shared pointer, we assume that it is allowed to access the associated memory. But the access must be performed using that pointer, ruling out other methods such as pointer forging.

$$
\begin{aligned}
\tau ::= & N \\
& \mathit{glob}\ C \\
& \mathit{dyn\_local}\ C \\
& \mathit{dyn\_share}\ c \\
& \mathit{comp}\ C\ c
\end{aligned}
$$

We define a predicate *cap* that holds when the target address is local to the active compartment or is shared and accessed through an appropriate capability.

4.3 Secure Information Flow

Memory safety and compartmentalization both either prevent or mitigate memory errors. But programs can be memory safe and still do insecure things! Consider the following code, in which we have some error-handling code that writes to a log. [TODO: more realistic example]

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

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

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

This is an instance of problematic information-flow. The solution is to implement a *secure information flow* (SIF) policy in Tagged C. SIF is a variant of *information flow control* (IFC) described in the venerable Denning and Denning [5]. At its simplest, if we classify inputs and outputs to the program into secure ("high") and public ("low") classifications, then the high inputs do not influence the low outputs. This generalizes to an arbitrary set of security classes, but out first example is concerned with just two: the value returned from `read_and_parse` and the output to the log. In our treatment of this example, we will describe a policy tailored to this particular set of security classes.

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

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

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

$$\begin{array}{ll} \tau ::= N & \\ \quad vtaint & \\ \quad pc\ f\ ets\ \overline{L} & \\ \quad vol & \end{array} \qquad \begin{array}{ll} \multicolumn{2}{l}{\textbf{ExtCallT}(\mathcal{P}, FN(f), FN(f'), \overline{vt})} \\ \multicolumn{2}{l}{\quad \textbf{assert}\ \forall vt \in \overline{vt}.vt = N \wedge \mathcal{P} = N} \\ \boldsymbol{\mathcal{P}'} \longleftarrow & \mathcal{P} \\ \boldsymbol{vt'} \longleftarrow & \text{case } f \text{ of} \\ & read\_and\_parse \Rightarrow vtaint \\ & \_ \Rightarrow vt \end{array}$$

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

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

The policy needs to failstop if a tainted value becomes visible to the outside world. That can happen when the value is passed as an argument to an external function, as we saw above, or when it is stored to volatile memory (typically representing a file or external device that might be read or might transfer.

$$\begin{array}{ll} \multicolumn{2}{l}{\textbf{BinopT}(\oplus, \mathcal{P}, vt_1, vt_2)} \\ \mathcal{P}' \longleftarrow & \mathcal{P} \\ vt' \longleftarrow & vt_1 \sqcup vt_2 \end{array}$$

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

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

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

In many simple programs, the join point of a conditional or loop is obvious: the point at which the chosen branch is complete, or the loop has ended. Such a simple example can be seen in fig. 3; `public1` must be tagged with the taint tag of `secret`, while it is safe to tag `public2` N, because that is after the join point, `J`. The same goes for fig. 4, if we are in a *termination-insensitive* setting [2]. In termination-insensitive noninterference, we allow an observer to glean information by the termination or non-termination of the program. So, it is safe to assume that the post-dominator J of the while loop is reached.

[TODO: implicit flow rules for statements]

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

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

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

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

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

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

  scanf("%19f", name);
```

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

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

J:  public2 = 42;

    return public2;
}
```

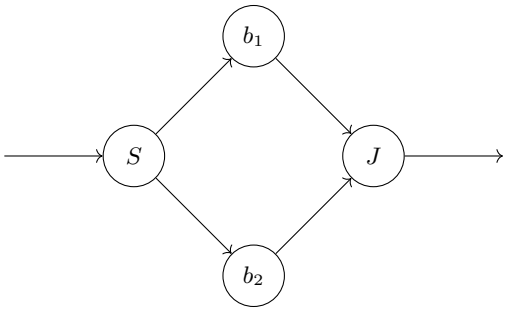


Fig. 3: Leaking via if statements

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

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

J:  public2 = 42;

    return public2;
}
```

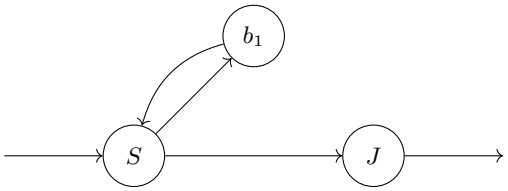


Fig. 4: Leaking via while statements

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

    while (secret) {
        goto b1;
    }

b2: public1 = 1;
    goto J;

b1: public1 = 1;

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

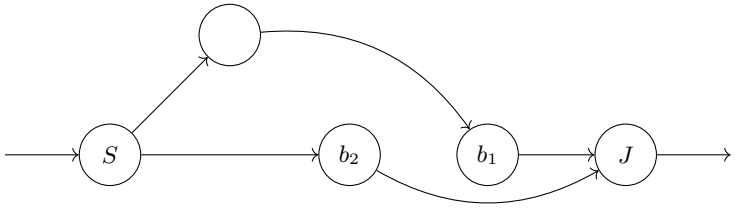
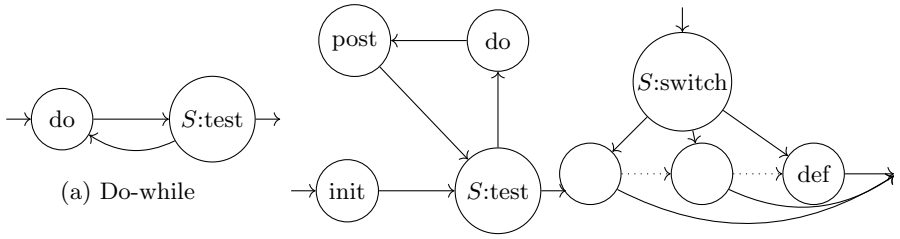


Fig. 5: Cheating with go-tos



(a) Do-while (b) For (c) Switch

Fig. 6: Remaining Branch Statements

```
  sanitize(name);
  strncat(query, name, strlen(name));

  sprintf(buf, sql_query(query));
  return;
}
```

This function sanitizes its input `name`, then appends the result to an appropriate SQL query, storing the result in `buf`. But, in the default case, the programmer has accidentally used the unsanitized string! This creates the opportunity for an SQL injection attack: a caller to this function could (presumably at the behest of an outside user) call it with `field` of 3 and `name` of "Bobby; drop table;".

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

In this context, we consider the functions `scanf` and `sanitize` as sources of information, and the input to `sanitize` and `sql_query` as information "sinks."

Let Σ be the set of these three identifiers, ranged over by σ. A value may be tainted any subset of Σ, written $\mathit{vtaint}\ \overline{\sigma}$. The PC Tag tracks the current function identifier and an association list of labels and sources. Each pair (L, σ) in the association list indicates that until reaching label L, the state itself has been influenced by σ.

$$\begin{aligned}\tau ::= & \mathit{vtaint}\ \overline{id} \\ & \mathit{pc}\ f\ \mathit{ets}\ \overline{(L, id)} \\ & N\end{aligned}$$

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

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

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

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

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

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

$$\mathbf{ArgT}(\mathcal{P}, \mathit{vt}, FN(f), AN(x), TN(ty))$$
$$\textbf{assert } \mathcal{P} \sqcup \mathit{vt} \sqsubseteq (\mathbf{f}, \mathbf{x})$$
$$\mathcal{P}' \longleftarrow$$

$$\mathbf{RetT}(\mathcal{P}_{CLE}, \mathcal{P}_{CLR}, \mathit{vt}, FN(f))$$
$$\textbf{assert } \mathcal{P} \sqcup \mathit{vt} \subseteq$$
$$\mathcal{P}' \longleftarrow$$
$$\mathit{vt}' \longleftarrow$$

The branching constructs are rather complicated, involving multiple steps and manipulations of the continuation that are not that relevant to their control points. Rather than give their semantics in full, it suffices to identify which transitions contain **SplitT** control points. In fig. 6, these are the transitions from the state marked S. Their semantics are given in full in the appendix.

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

The other implementation detail to consider are the label tags. These resemble instruction tags, and that is exactly how they would be implemented:

as a special instruction tag on the appropriate instruction, which might be an existing instruction or a specially added no-op, that the processor handles by introducing a tag corresponding to that label.

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

5 Implementing Tagged C with PIPE

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.

Chhak et al. [4] give a general strategy for mapping Tagged C's tag rules onto instructions in a PIPE target. But as they note, translating tag rules in full generality requires adding extra instructions that may be unnecessary for some policies. A Tagged-C control point may require a tag from a location that is not read under a normal compilation scheme, or must update tags in locations that would otherwise not be written.

To mitigate this, control points whose compilation would add potentially extraneous instructions take optional parameters or return optional results. We will explain how the rule should be implemented in the target if the options are used. If a policy does not make use of the options, it will be sound to compile without the extra instructions. Optional inputs and outputs are marked with boxes.

6 Evaluation

Tagged C aims to combine the flexibility of tag-based architectures with the abstraction of a high-level language. How well have we achieved this aim?

[Here we list criteria and evaluate how we fulfilled them]

- Flexibility: we demonstrate three policies that can be used alone or in conjunction
- Applicability: we support the full complement of C language features and give definition to many undefined C programs
- Practical security: our example security policies are based on important security concepts from the literature

6.1 Limitations of the Tag Mechanism

By committing to a tag-based mechanism, we do restrict the space of policies that Tagged C can enforce. In general, a reference monitor can enforce any policy that constitutes a *safety property*—any policy whose violation can be demonstrated by a single finite trace. This class includes such policies as "no integer overflow" and "pointers are always in-bounds," which depend on the values of variables. Tag-based monitors cannot enforce any policy that depends on the value of a variable rather than its tags.

7 Related Work

Reference Monitors The concept of a reference monitor was first introduced fifty years ago in [1]: a tamper-proof and verifiable subsystem that checks every security-relevant operation in a system to ensure that it conforms to a security *policy* (a general specification of acceptable behavior; see [7].)

A reference monitor can be implemented at any level of a system. An *inline reference monitor* is a purely compiler-based system that inserts checks at appropriate places in the code. Alternatively, a reference monitor might be embedded in the operating system, or in an interpreted language's runtime. A *hardware reference monitor* instead provides primitives at the ISA-level that accelerate security and make it harder to subvert.

Programmable Interlocks for Policy Enforcement (PIPE) [6] is a hardware extension that uses *metadata tagging*. Each register and each word of memory is associated with an additional array of bits called a tag. The policy is decomposed into a set of *tag rules* that act in parallel with each executing instruction, using the tags on its operands to decide whether the instruction is legal and, if so, determine which tags to place on its results. PIPE tags are large relative to other tag-based hardware, giving it the flexibility to implement complex policies with structured tags, and even run multiple policies at once.

Other hardware monitors include Arm MTE, [Binghamton], and CHERI. Arm MTE aims to enforce a narrow form of memory safety using 4-bit tags, which distinguish adjacent objects in memory from one another, preventing buffer overflows, but not necessarily other memory violations. [TODO: read the Binghamton paper, figure out where they sit here.]

CHERI is capability machine [TODO: cite OG CHERI]. In CHERI, capabilities are "fat pointers" carrying extra bounds and permission information,

and capability-protected memory can only be accessed via a capability with the appropriate privilege. This is a natural way to enforce spatial memory safety, and techniques have been demonstrated for enforcing temporal safety [12], stack safety [11], and compartmentalization [TODO: figure out what to cite], with varying degrees of ease and efficiency. But CHERI cannot easily enforce notions of security based on dataflow, such as Secure Information Flow.

In this paper, we describe a programming language with an abstract reference monitor. We realize it as an interpreter with the reference monitor built in, and envision eventually compiling to PIPE-equipped hardware. An inlining compiler would also be plausible. As a result of this choice, our abstract reference monitor uses a PIPE-esque notion of tags.

PIPE Backend Implementation

Aspect Oriented Programming [TODO: do forward search from original AOP paper]

8 Future Work

We have presented the language and a reference interpreter, built on top of the CompCert interpreter [8], and three example policies. There are several significant next-steps.

Compilation An interpreter is all well and good, but a compiler would be preferable for many reasons. A compiled Tagged C could use the hardware acceleration of a PIPE target, and could more easily support linked libraries, including linking against code written in other languages. The ultimate goal would be a fully verified compiler, but that is a very long way off.

Language Proofs There are a couple of properties of the language semantics itself that we would like to prove. Namely (1) that its behavior (prior to adding a policy) matches that of CompCert C and (2) that the behavior of a given program is invariant under all policies up to truncation due to failstop.

Policy Correctness Proofs For each example policy discussed in this paper, we sketched a formal specification for the security property it ought to enforce. A natural continuation would be to prove the correctness of each policy against these specifications.

Policy DSL Currently, policies are written in Gallina, the language embedded in Coq. This is fine for a proof-of-concept, but not satisfactory for real use. We plan to develop a domain-specific policy language to make it easier to write Tagged C policies.

References

1. Anderson, J.P.: Computer Security Technology Planning Study. Tech. rep., U.S. Air Force Electronic Systems Division (10 1972)
2. Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. In: Jajodia, S., Lopez, J. (eds.) Computer Security - ESORICS 2008. pp. 333–348. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
3. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.: A few billion lines of code later: Using static analysis to find bugs in the real world. Commun. ACM **53**(2), 66–75 (feb 2010). https://doi.org/10.1145/1646353.1646374, https://doi.org/10.1145/1646353.1646374
4. Chhak, C., Tolmach, A., Anderson, S.: Towards formally verified compilation of tag-based policy enforcement. In: Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 137–151. CPP 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3437992.3439929, https://doi.org/10.1145/3437992.3439929
5. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7), 504–513 (jul 1977). https://doi.org/10.1145/359636.359712, https://doi.org/10.1145/359636.359712
6. Dhawan, U., Vasilakis, N., Rubin, R., Chiricescu, S., Smith, J.M., Knight Jr., T.F., Pierce, B.C., DeHon, A.: PUMP: A Programmable Unit for Metadata Processing. In: Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy. p. 8:1–8:8. HASP '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2611765.2611773, http://doi.acm.org/10.1145/2611765.2611773
7. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy. pp. 11–20. IEEE Computer Society (1982), http://dblp.uni-trier.de/db/conf/sp/sp1982.html#GoguenM82a
8. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (jul 2009). https://doi.org/10.1145/1538788.1538814, https://doi.org/10.1145/1538788.1538814
9. Munoz, D.: After all these years, the world is still powered by c programming, https://www.toptal.com/c/after-all-these-years-the-world-is-still-powered-by-c-programming
10. Overflow, S.: 2022 stack overflow annual developer survey (2022), https://survey.stackoverflow.co/2022/
11. Skorstengaard, L., Devriese, D., Birkedal, L.: StkTokens: Enforcing Well-bracketed Control Flow and Stack Encapsulation using Linear Capabilities. Proceedings of the ACM on Programming Languages **3**(POPL), 1–28 (2019)
12. Wesley Filardo, N., Gutstein, B.F., Woodruff, J., Ainsworth, S., Paul-Trifu, L., Davis, B., Xia, H., Tomasz Napierala, E., Richardson, A., Baldwin, J., Chisnall, D., Clarke, J., Gudka, K., Joannou, A., Theodore Markettos, A., Mazzinghi, A., Norton, R.M., Roe, M., Sewell, P., Son, S., Jones, T.M., Moore, S.W., Neumann, P.G., Watson, R.N.M.: Cornucopia: Temporal safety for cheri heaps. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 608–625 (2020). https://doi.org/10.1109/SP40000.2020.00098

A Syntax

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

Fig. 7: Tagged C Abstract Syntax

B Continuations

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

C States

States can be of several kinds, denoted by their script prefix: a *general state* $\mathcal{S}(\dots)$, an *expression state* $\mathcal{E}(\dots)$, a *call state* $\mathcal{C}(\dots)$, or a *return state* $\mathcal{R}(\dots)$. Finally, the special state *failstop* ($\mathcal{F}(\dots)$) represents a tag failure, and carries the state that produced the failure. [Allison: to whatever degree you've figured out what is useful here by publication-time, we can tune this to be more specific.]

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

D Initial State

Given a list xs of variable identifiers id and types ty, a program's initial memory is defined by iteratively allocating each one in memory and updating the global environment with its base address, bound, type, and a static identity tag. Let $|ty|$ be a function from types to their sizes in bytes. The memory is initialized $\mathbf{undef}@vt@\overline{lt}$ for some vt and $\overline{lt}$, unless given an initializer. Let m_0 and ge_0 be the initial (empty) memory and environment. The parameter b marks the start of the global region.

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

E Step Rules

E.1 Sequencing rules

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

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

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

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

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

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

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

E.2 Conditional rules

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

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

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

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

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

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

E.3 Loop rules

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

E.4 Contexts

Our expression semantics are contextual. A context *ctx* is a function from an expression to an expression and a tag. We identify a valid context using the *context* relation over a "kind" (left-hand or right-hand, LH or RH), and an expression.

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

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

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

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

E.5 Expression Rules

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

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

$$\frac{le[id] = \bot \;\; ge[id] = \textsc{Var}(f, pt)}{Evar\ id \Rightarrow_{\textsc{Lh}} Efloc\ l@pt}$$

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

$$\frac{ty = TStruct\ id \vee ty = TUnion\ id\ offset\ id\ fld = \delta \;\; pt' \leftarrow \mathbf{FieldT}(pt, TN(ty), GN(x))}{Efield\ (Eval\ p@pt : ty)\ fld) \Rightarrow_{\textsc{Lh}} Eloc\ (p + \delta)@pt'}$$

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

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

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

$$\frac{v_1\ \langle\oplus\rangle\ v_2 = v' \;\; vt' \leftarrow \mathbf{BinopT}(\oplus, \mathcal{P}, vt_1, vt_2) \quad e = Ebinop\ \oplus\ (Eval\ v_1@vt_1)\ (Eval\ v_2@vt_2)}{(\mathcal{P}, m, e) \Rightarrow_{\textsc{Rh}} (\mathcal{P}, m, Eval\ v'@vt')}$$

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

$$\frac{ty_1 = ptr\ ty_1' \qquad \neg isptr(ty_2) \qquad m[v]_{|ty_1'|} = \_@vt@\overline{lt} \qquad vt \leftarrow \mathbf{PICastT}(\mathcal{P}, pt, \boxed{vt, \overline{lt}})}{(\mathcal{P}, m, Ecast\ (Eval\ v@pt : ty_1)\ ty_2) \Rightarrow_{\textsc{Rh}} (\mathcal{P}, m, Eval\ v@vt' : ty_2)}$$

$$\frac{\neg isptr(ty_1) \qquad ty_2 = ptr\ ty_2' \qquad m[v]_{|ty_2'|} = \_@vt_2@\overline{lt} \qquad pt \leftarrow \mathbf{IPCastT}(\mathcal{P}, vt_1, \boxed{vt_2, \overline{lt}})}{(\mathcal{P}, m, Ecast\ (Eval\ v@vt_1 : ty_1)\ ty_2) \Rightarrow_{\textsc{Rh}} (\mathcal{P}, m, Eval\ v@pt : ty_2)}$$

$$\frac{ty_1 = ptr\ ty_1' \qquad ty_2 = ptr\ ty_2' \qquad m[v]_{|ty_1'|} = m[v]_{|ty_2'|} = \_@vt@\overline{lt} \qquad pt' \leftarrow \mathbf{PPCastT}(\mathcal{P}, pt, \boxed{vt, \overline{lt}})}{(\mathcal{P}, m, Ecast\ (Eval\ v@pt : ty_1)\ ty_2) \Rightarrow_{\textsc{Rh}} (\mathcal{P}, m, Eval\ v@pt' : ty_2)}$$

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

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

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

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

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

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

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

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

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

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

E.6 Call and Return Rules

In order to make a call, we need to reduce the function expression to an *Efloc* _@ value, an abstract location corresponding to a particular function. Then we can make the call.

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

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

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

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

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

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

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

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

And finally, we have the return rules.

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

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

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

F Moved from Intro

[SNA: I'm organizing our diss tracks into paragraphs that we can cut or move as needed]

Why Dynamic? Unfortunately, it is not always possible to fully secure C code before run-time. Ideally, bugs would be quickly identified and then fixed promptly. That is not always possible for a variety of reasons: bugs may escape detection, require significant effort to diagnose, or be impractical to fix. There are many techniques for finding bugs, but there is a shared stumbling block: C is not well defined. We cannot always agree on when something is a bug in C, especially code using Undefined Behaviors (UB) [**?**]. Confusion around expected behavior is no small problem. There are 191 undefined behaviors and 52 unspecified behaviors in the C99 specification [**?**]. Sometimes these behaviors are benign and skillfully used by the developer, other times they are unintended and highly dangerous. Unfortunately the distinction between the two is easily lost. Discerning expert code review is considered best practice, although it is rarely perfectly successful [] even if an expert is available at all. Even when there is both consensus and detection of a bug[APT: ??] AN: we can find it at and we can agree its a problem that should be fixed, changing the code may not be possible because it

is in proprietary 3rd party libraries and drivers, or because regulations prohibit changes [3].

[APT: last clause is mysterious] AN: for example FDA approval used to forbid patching because you'd have to go through recertification. So healthcare wouldn't patch. SNA pointed out the coverity paper comments on this as a reason for bugs not getting fixed

Why C-Level? Tag-based enforcement in general has a significant body of work at the assembly level, especially PIPE (Programmable Interlocks for Policy Enforcement) []. However, even at the assembly-level these systems need the compiler to be in the trusted computing base (TCB), as many policies require knowledge of source-level constructs, even ones that do not depend on detailed knowledge of the program's behavior [cite Nick and Andre; anyone else?]. Moving policy-definition to the source level therefore does not expand the TCB and allows C developers to reason about policies in terms of the language that they program in regularly.

Notations Values are ranged over by v, variable identifiers by x, and function identifiers by f. Tags use a number of metavariables: t ranges over all tags, while we will use vt to refer to the tags associated with values, pt for tags on pointer values and memory-location expressions, lt for tags associated with memory locations themselves, nt for "name tags" automatically derived from identifiers, $\mathcal{P}$ for the global "program counter tag" or PC Tag. An *atom* is a pair of a value and a tag, *Eval* $v@vt$; the @ symbol should be read as a pair in general, and is used when the second object in the pair is a tag. Expressions are ranged over by e, statements by s, and continuations by k. The continuations are defined in appendix B, and step rules in appendix E.

A memory is an array of bytes, where each byte is part of an atom. Each byte is also associated with a "location tag" lt. When a contiguous region of s bytes starting at location l comprise an atom $v@vt$, and their locations tags comprise the list $\overline{lt}$, we write $m[l]_s = v@vt@\overline{lt}$. Likewise, $m[l \ldots l+s \mapsto v@vt@\overline{lt}]_s$ denotes storing that many bytes. Visually, we will represent whole atoms in memory as condensed boxes, with their location tags separate.