Flexible Runtime Security Enforcement with Tagged C

Sean Anderson[0009−0006−7681−3683], Allison Naaktgeboren[0009−0004−0405−9306], and Andrew Tolmach[0000−0002−0748−2044]
{ander28,naak,tolmach}@pdx.edu

Portland State University
Portland, OR, USA

Abstract. We introduce Tagged C, a novel C variant with built-in tag-based reference monitoring that can be enforced by hardware mechanisms such as the PIPE (Processor Interlocks for Policy Enforcement) processor extension. Tagged C expresses security policies at the level of C source code. It is designed to express a variety of dynamic security policies, individually or in combination, and enforce them with compiler and hardware support. Tagged C supports multiple approaches to security and varying levels of strictness. We demonstrate this range by providing examples of memory safety, compartmentalization, and secure information flow policies. We also give a full formalized semantics and a reference interpreter for Tagged C.

1 Introduction

Many essential technologies rely on new and old C code. Operating systems (Linux, Windows, OSX, BSD), databases (Oracle, sqlite3), the internet (Apache, NGNIX, NetBSD, Cisco IOS), and the embedded devices that run our homes and hospitals are built in and on C. The safety of these technologies depends on the security of their underlying C codebases. Insecurity can arise from C undefined behavior (UB) such as memory errors (e.g. buffer overflows, use-after-free, double-free), logic errors (e.g. SQL injection, input-sanitization flaws), or larger-scale architectural flaws (e.g. over-provisioning access rights).

Although static analyses can detect and mitigate many C insecurities, a last line of defense against undetected or unfixable vulnerabilities is runtime enforcement of *security policies* using a reference monitor [1]. In particular, many useful policies can be specified in terms of flow constraints on *metadata tags*, which augment the underlying data with information like type, provenance, ownership, or security classification. A tag-based policy takes the form of a set of rules that check and update the metadata tags at key points during execution; if a rule violation is encountered, the program *failstops*. Although monitoring based on metadata tags is less flexible and powerful than monitoring based on the underlying data values, it can still enforce many useful security properties, including both low-level concerns such as memory safety and high-level properties such as secure information flow [13] or mandatory access control [20].

Tag-based policies are especially well-suited for efficient hardware enforcement, using processor extensions such as ARM MTE [2], STAR [17], and PIPE. PIPE[1] (Processor Interlocks for Policy Enforcement) [4,5],the specific motivator for our work, is a programmable hardware mechanism that supports monitoring at the granularity of individual instructions. Each value in memory and registers is extended with a metadata tag. Before executing each instruction, PIPE checks the opcode and the tags on its operands to see if the operation should be permitted according to a tag rule, and if so, what tags should be assigned to the result. PIPE is highly flexible: it supports arbitrary software-defined tag rules over large (word-sized) tags with arbitrary structure, which enables fine-grained policies and composition of multiple policies. This flexibility is useful because security needs may differ among codebases, and even within a codebase. A conservative, one-size-fits-all policy might be too strong, causing failstops during normal execution. Sensitive code might call for specialized protection.

But PIPE policies can be difficult for a C engineer to write: their tags and rules are defined in terms of individual machine instructions and ISA-level concepts, and in practice they depend on reverse engineering the behavior of specific compilers. Moreover, some security policies can only be expressed in terms of high-level code features that are not preserved at machine level, such as function arguments, structured types, and structured control flow.

To address these problems, we introduce a *source-level* specification framework, *Tagged C*, which allows engineers to describe policies in terms of familiar C-level concepts, with tags attached to C functions, variables and data values, and rules triggered at *control points* that correspond to significant execution events, such as function calls, expression evaluation, and pointer-based memory accesses. Control points resemble "join points" in aspect-oriented programming, but the "advice" in this case can only take the form of manipulating tags, not data. In previous work on the Tagine project [10], we outlined such a framework for a toy source language and showed how high-level policies could be compiled to ISA-level policies and enforced using PIPE-like hardware. Here we extend this approach to handle the full, real C language, by giving a detailed design for the necessary control points and showing how they are integrated into C's dynamic semantics. Although motivated by PIPE, Tagged C is not tied to any particular enforcement mechanism. We currently implement it using a modified C interpreter rather than a compiler. We validate the design of Tagged C by using it to specify a range of interesting security policies, including compartmentalization, memory protection, and secure information flow.

Formally, Tagged C is defined as a variant C semantics that instruments ordinary execution with control points. At each control point, a user-defined set of tag rules is consulted to propagate tags and potentially halt execution. In the limiting case where no tag rules are defined, the semantics is similar to that of ordinary C, except that the memory model is very concrete; data pointers are just integers, and all globals, dynamically-allocated objects, and address-

[1] Variants of PIPE have been called PUMP [15] or SDMP [27]and marketed commercially under the names Dover CoreGuard and Draper Inherently Secure Processor.

taken objects are allocated in the same integer-addressed memory space. Memory behaviors that would be undefined in standard C are given a definition consistent with the behavior of a typical compiler. We build the Tagged C semantics on top of the CompCert C semantics, which is formalized as part of the CompCert verified compiler [22,23]. For prototyping and executing example policies, we provide a reference interpreter [2], also based on that of CompCert, written in the Gallina functional language of the Coq Proof Assistant [12]. Tag types and rules are also written directly in Gallina.

The choice of control points and their associations with tag rules, as well as the tag rules' signatures, form the essence of Tagged C's design. We have validated this design on the three classes of policies explored in this paper, and, outside of a few known limitations related to `malloc` (Section 4.2), we believe it is sufficiently expressive to describe most other flow-based policies, although further experience is needed to confirm this.

Contributions In summary, we offer the following contributions:

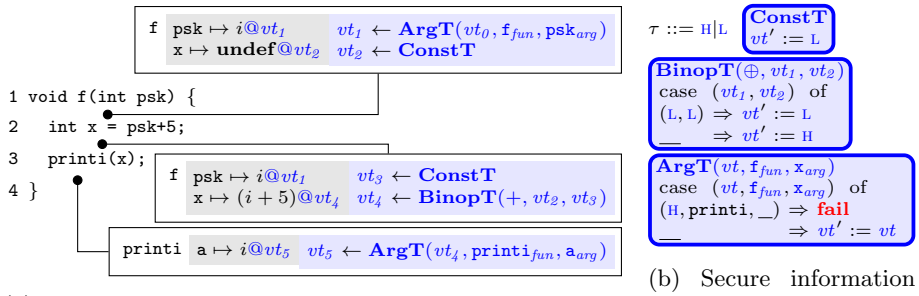
- The design of a comprehensive set of *control points* at which the C language interfaces with a tag-based policy. These expand on prior work by encompassing the full C language while being powerful enough to enable a range of policies even in the presence of C's more challenging constructs (e.g., `goto`, conditional expressions, etc.).
- Tagged C policies enforcing: (1) compartmentalization, including a novel compartmentalization policy with separate public and private memory; (2) memory safety, with realistic memory models that support varying kinds of low-level idioms; and (3) secure information flow.
- A full formal semantic definition for Tagged C, formalized in Coq, describing how the control points interact with programs, and an interpreter, implemented and verified against the semantics in Coq and extracted to OCaml.

The paper is organized as follows. Section 2 gives a high-level introduction of metadata tagging by example. Section 3 summarizes the Tagged C language as a whole and its control points. Section 4 describes three example policies and how their needs inform our choices of control points. Section 5 describes the Coq-based implementation of Tagged C. Section 6 discusses related work, and Section 7 describes future work.

2 Metadata Tags and Policies, by Example

Consider a straightforward security requirement for a program that handles sensitive passkeys: "do not leak passkeys on insecure channels." This is an instance of a broad class of *secure information flow* (SIF) policies. Suppose the code on the left in Fig. 1a is part of such a system, where `psk` is expected to be a passkey and `printi` prints an integer to an insecure channel, so `f` indirectly performs a

[2] Available at `https://github.com/SNoAnd/Tagged-C`



(a) Example execution showing tag rules and propagation.

(b) Secure information flow instantiation (pt. 1)

Fig. 1: Tag Rules and Instantiation

leak via the local variable `x`. We now explain how a monitor specified in Tagged C could detect such a leak. (Of course, this particular leak could also be easily found using static analysis.)

In Tagged C, all values carry a metadata tag. Whenever execution reaches a control point, it consults an associated tag rule, to check whether the next execution step should be allowed to continue and if so, to update the tags. A policy consists of a tag type definition and instantiations of the tag rules for every control point. For a simple SIF policy like this one, the tag type is an enumeration containing H (high security) and L (low security). In this case, the input `psk` arrives in `f` with the tag H. This tag will be propagated along with the value through variable accesses, assignments, and arithmetic, according to generic rules that are not specific to this program. Finally, a program-specific argument-handling rule for `printi` will check that the tag is L; since it is not, the rule will cause a failstop.

To explain the mechanics of Tagged C, we first show in Fig. 1a the policy-independent framework under which tag rules are triggered in this program: the initial tag on `psk` (vt_0) passes through the **ArgT** tag rule, is combined with the tag on the constant 5 via the **BinopT** tag rule, and then is passed to **ArgT** again on the call to obtain the tag on the parameter inside `printi` (here called `a`). Figure 1a maps three points in the execution of `f` to descriptions of the corresponding program states, with the input value and all tags treated symbolically. In each state, the first column (white) shows the active function, the second (gray) gives the symbolic values and tags of variables in the local environment, and the third (blue) shows the rules that produce those tags. Throughout the paper, we highlight tag-related metavariables, rules, etc. in blue. We write $v@vt$ for value v tagged with vt. Tags that are derived from identifiers are subscripted with the identifier namespace, e.g. $\texttt{f}_{fun}$ is the tag associated with the function name `f`. **undef** denotes an uninitialized value.

The SIF policy described informally above is implemented by instantiating the tag rules as shown in Fig. 1b. The resulting behavior is best understood by mentally "weaving" together the two figures. Suppose `f` is called with an argument value $i@\text{H}$. This first invocation of **ArgT** simply passes the H tag on

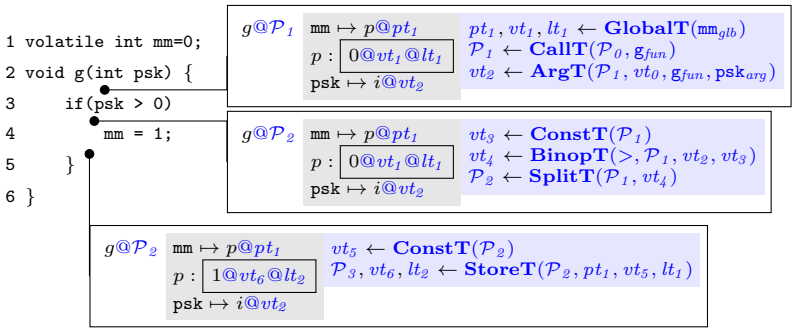


Fig. 2: Second example showing tag rules and tag propagation.

to the output vt_1, because the name of the function being called does not match `printi`. In tag rules, the assignment operator := denotes an assignment to the named tag-rule output, by convention written as primed metavariables t'. The initial tag vt_2 on local variable `x` and the tag vt_3 on the constant `5` come from **ConstT**, which tags all constants as L. The result of the addition on line 2 is tagged by **BinopT** as the higher of the two inputs, so vt_4 is H. Finally, upon entry into `printi`, **ArgT** is invoked again; this time it failstops. Note that in this policy, **ArgT** is code-specific (it checks for a particular function name `printi`) whereas the other rules are generic.[3]

As a second example, Fig. 2 steps through the execution of a function `g` that adds two new wrinkles: we need to keep track of metadata associated with addresses and with the program's control-flow state. We suppose `mm` is a memory-mapped device register that can be read from outside the program, so we want to avoid storing the passkey there; therefore we need a way to monitor stores to memory. Furthermore, although this code does not leak the passkey directly, it does so indirectly: since the store to `mm` is conditional on testing `psk`, an outside observer of `mm` can deduce one bit of the key (an *implicit flow* [13]).

In addition to tags on values, Tagged C attaches tags to memory locations (*location tags*, ranged over by lt) and tracks a special global tag called the PC tag (ranged over by $\mathcal{P}$, and attached to the function name in our diagrams). Tagged C initializes the tags on `mm` with the **GlobalT** rule. The PC tag at the point of call, $\mathcal{P}_0$, is fed to **CallT** to determine a new PC tag inside of `g`. And the `if`-statement consults the **SplitT** rule to update the PC tag inside of its branch based on the value-tag of the expression `psk < 0`. Once inside the conditional, when the program assigns to `mm`, it must consult the **StoreT** rule.

Figure 3 shows an instantiation of these rules that extend our previous SIF policy. The rule for globals initializes the location tag of `mm` to L, as a low-security output channel, and marks all other addresses H. **CallT** sets the PC tag to L

[3] For simplicity, we omit showing tag rules that play no interesting role in this example: **AccessT** and **AssignT**, which are triggered each time a variable is read and assigned, respectively, and **CallT**, which is triggered by the call itself.

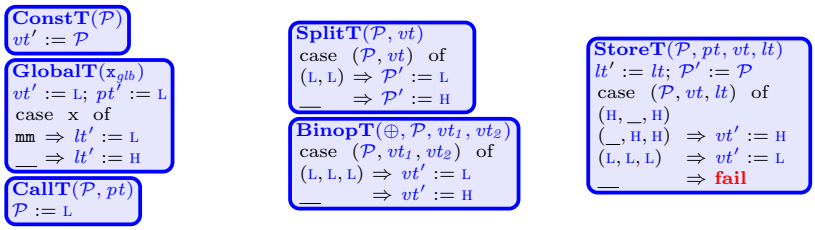


Fig. 3: Tag rule instantiations for secure information flow (pt. 2)

on entry to each function. Whenever execution branches on a high-tagged value, the PC tag will be set to H. We modify the previous rules so that all expressions propagate the higher of the PC tag and the relevant value tag(s). This is shown for the updated **BinopT** in Fig. 3; the **ArgT** rule needs a similar adjustment. When an assignment is to a memory location, the store rule will check the tag on that location against the value being written, and failstop if a high value would be written to a low location. For this program, **SplitT** will set the PC tag to H, as it branches on a value derived from `psk`; then, at the write to `mm`, **StoreT** will fail rather than write to a low address in a high context.

3 The Tagged C Language: Syntax and Semantics

Tagged C contains almost all features of full ISO C 99.[4] Its semantics is based on that of CompCert C [22], a formalization of the C standard into a small-step reduction semantics. Tagged C's semantics differs from CompCert C's in two key respects: tag support and memory model.

Tags Tagged C's values and states are annotated with metadata tags, and its reductions contain control points, which are hooks within the operational semantics at which the tag policy is consulted and either tags are updated, or the system failstops. Tagged C relies on a fixed number of predefined control points, which we keep small in order to simplify and organize the task of the policy designer. A control point consists of the name of a *tag rule* and the bindings of its inputs and outputs. For example, consider the expression step reduction for binary operations:

$$\frac{v_1 \langle\oplus\rangle v_2 = v' \qquad e = \mathit{Ebinop}\ \oplus\ v_1\ v_2}{m, e \Rightarrow_{\mathrm{RH}} m, v'} \qquad \frac{v_1 \langle\oplus\rangle v_2 = v'\ \ vt' \leftarrow \mathbf{BinopT}(\oplus, \mathcal{P}, vt_1, vt_2) \qquad e = \mathit{Ebinop}\ \oplus\ v_1@vt_1\ v_2@vt_2}{\mathcal{P}, m, te, e \Rightarrow_{\mathrm{RH}} \mathcal{P}, m, te, v'@vt'}$$

On the left, the ordinary "tagless" version of the rule reduces a binary operation on two inputs to a single value. On the right, the Tagged C version adds tags to the operands and tags the result based on the **BinopT** rule.

[4] It inherits the limitations of CompCert C, primarily that `setjump` and `longjump` may not work, and variable-length arrays are not supported.

Rule Name	Inputs	Outputs	Control Points
CallT	$\mathcal{P}, pt$	$\mathcal{P}'$	Update PC tag at call
ArgT	$\mathcal{P}, vt, \mathtt{f}_{fun}, \mathtt{x}_{arg}$	$\mathcal{P}', vt'$	Per argument at call
RetT	$\mathcal{P}_{CLE}, \mathcal{P}_{CLR}, vt$	$\mathcal{P}', vt'$	Handle PC tag, return value
LoadT	$\mathcal{P}, pt, vt, lt$	vt'	Memory
StoreT	$\mathcal{P}, pt, vt, lt$	$\mathcal{P}', vt', lt'$	Memory stores
AccessT	$\mathcal{P}, vt$	vt'	Variable accesses
AssignT	$\mathcal{P}, vt_1, vt_2$	$\mathcal{P}', vt'$	Variable assignments
UnopT	$\odot, \mathcal{P}, vt$	vt'	Unary operation
BinopT	$\oplus, \mathcal{P}, vt_1, vt_2$	vt'	Binary operation
ConstT	$\mathcal{P}$	vt'	Applied to constants/literals
InitT	$\mathcal{P}$	vt'	Applied to fresh variables
SplitT	$\mathcal{P}, vt, \mathtt{L}_{lbl}$	$\mathcal{P}'$	Statement control split points
LabelT	$\mathcal{P}, \mathtt{L}_{lbl}$	$\mathcal{P}'$	Labels/arbitrary code points
ExprSplitT	$\mathcal{P}, vt$	$\mathcal{P}'$	Expression control split points
ExprJoinT	$\mathcal{P}, vt$	$\mathcal{P}', vt'$	Join points in expressions
GlobalT	$\mathtt{x}_{glb}, \mathtt{ty}_{typ}$	pt', vt', lt'	Program initialization
LocalT	$\mathcal{P}, \mathtt{ty}_{typ}$	$\mathcal{P}', pt', lt'$	Stack allocation (per local)
DeallocT	$\mathcal{P}, \mathtt{ty}_{typ}$	$\mathcal{P}', vt', lt'$	Stack deallocation (per local)
ExtCallT	$\mathcal{P}, pt, \overline{vt}$	$\mathcal{P}'$	Call to linked code
MallocT	$\mathcal{P}, pt, vt$	$\mathcal{P}', pt', vt', lt'$	Call to `malloc`
FreeT	$\mathcal{P}, pt, vt$	$\mathcal{P}', vt', lt'$	Call to `free`
FieldT	$\mathcal{P}, pt, \mathtt{ty}_{typ}, \mathtt{x}_{glb}$	pt'	Structure/union field access
PICastT	$\mathcal{P}, pt, lt, \mathtt{ty}_{typ}$	vt'	Cast from pointer to scalar
IPCastT	$\mathcal{P}, vt, lt, \mathtt{ty}_{typ}$	pt'	Cast from scalar to pointer
PPCastT	$\mathcal{P}, pt_1, pt_2, lt_1, lt_2, \mathtt{ty}_{typ}$	pt'	Cast between pointers
IICastT	$\mathcal{P}, vt_1, \mathtt{ty}_{typ}$	pt'	Cast between scalars

Table 1: Full list of tag-rule signatures and control points.

The tag rule itself is instantiated as a partial function; if a policy leaves a tag rule undefined on some inputs, then those inputs violate the policy, sending execution into a special failstop state. The names and signatures of all the tag rules, and their corresponding control points, are listed in Table 1. In these signatures, we use the metavariable $\mathcal{P}$ to denote the PC tag, pt for tags that will be attached to pointer values, vt for tags that will be attached to values in general, and lt for tags that are associated with specific addresses in memory. We also range over different classes of identifiers with the metavariables: $\mathtt{f}_{fun}$, function identifiers; $\mathtt{x}_{arg}$, function arguments; $\mathtt{x}_{glb}$, global variable names; $\mathtt{L}_{lbl}$, labels; and $\mathtt{ty}_{typ}$, types. We briefly summarize the rules below, and give motivating examples of their use in Section 4.

Memory Unlike CompCert C, Tagged C has no memory-related UB. CompCert C models memory as a collection of disjoint blocks, and treats each variable as having its own block. Pointers are described by a (block, offset) pair, and invalid pointer accesses (out-of-bounds, use after free, etc.) produce UB. Tagged C in-

stead separates variables into public and private data. Public data (all heap data, globals, arrays, structs, and address-taken locals) share a single flat address space (possibly with holes), and pointers are offsets into this space. Pointer accesses outside of this space cause an explicit failstop, rather than UB. Private data (non-address-taken locals, parameters) live in a separate, abstract environment. Program-specified stores, e.g. writes through pointers, can cause arbitrary damage to public data, but do not affect private data. This model is strong enough to support a reasonable notion of semantics-preserving compilation, without making any commitment about fine-grained memory safety, which is intentionally left for explicit tag policies to specify (see Section 4.1). In an implementation that compiles to PIPE, the private data can be protected by a small number of "built-in" tags.

Control Points In our scheme, pure expressions take as arguments the PC tag $\mathcal{P}$ and any operand tags, and produce a tag for the result of the expression (**ConstT** for constants, **UnopT** and **BinopT** for operations, **FieldT** for struct and union fields, and **AccessT** and **LoadT** as described below). Impure expressions additionally produce a new PC tag (**AssignT**, **StoreT**, and **ExprSplitT**).

The distinction between **AccessT** and **LoadT**, and between **AssignT** and **StoreT**, corresponds to private (non-address-taken) and public (allocated in public memory) variables. All reads of variables invoke **AccessT**, and all assignments invoke **AssignT**, so that the behavior of the variable itself is independent of where it is stored. Public variables additionally use the **LoadT** and **StoreT** rules to add restrictions to how variables in memory are accessed.

The **ExprSplitT** tag rule updates the PC tag when an expression branches based on a value; it is paired with **ExprJoinT**, which updates the PC tag again when the branches have rejoined. Similarly, **SplitT** updates the PC during branching statements. The **LabelT** rule can changes the PC tag at any labeled point in execution, and handles join points following branch statements.

CallT and **RetT** update the PC tag on entry and exit from a function. **CallT** is parameterized by the function pointer being called and the PC tag. **RetT** is parameterized by both the caller's PC tag ($\mathcal{P}_{CLR}$) and the callee's ($\mathcal{P}_{CLE}$); it can also update the return value's tag. **ArgT** updates tags on any arguments, based on the function and argument names.

Newly initialized variables are tagged according to the **InitT** rule, as well as **LocalT** if they are public locals; the lt' tag returned by the latter is used to tag the memory occupied by the variable. Similarly, global variables are initialized before runtime based on the **GlobalT** rule. **DeallocT** returns an lt' tag used to re-tags the memory of deallocated locals.

The heap equivalents of **LocalT** and **DeallocT** are **MallocT** and **FreeT**. Again, the lt' tags returned by these functions are used to tag and re-tag the allocated memory. Other library functions have the tags of their results tagged by the **ExtCallT** tag rule.

The cast rules are specialized based on whether the original type or the new type is a pointer, or both, because casts to and from pointers can make use of the location tags at their targets. This enables our PNVI memory safety policy

(Section 4.1), and more generally policies that keep track of the correspondence between pointers and their targets.

There is always the chance that new policies might arise for which our current set of control points proves to be inadequate. There is no conceptual reason why control points cannot be added or given modified signatures as needed, but extending the interpreter and (eventually) the compiler would be non-trivial. Care needs to be taken in designing control points that are amenable to compilation for PIPE: tag rule evaluation has a complicated interaction with compiler optimization [10], and some potentially useful tag rule signatures (such as updating tags on operation inputs to enforce non-aliasing of pointers) would require the compiler to generate extra instructions to work around limitations of the PIPE hardware.

Combining Policies Multiple policies can be enforced in parallel. If policy A has tag type τ_A and policy B has τ_B, then policy $A \times B$ should have tag type $\tau = \tau_A \times \tau_B$. Its tag rules should apply the rules of A to the left projection of all inputs and the rules of B to the right projection to generate the components of the new tag. If either side failstops, the entire rule should failstop.

This process can be applied to any number of different policies, allowing, for instance, a combination of a baseline memory safety policy with several more targeted information-flow policies. Alternatively, a policy can delegate to tag rules from other, related policies, as illustrated in Section 4.2, below.

4 Example Policies

In this section, we discuss concrete policy implementations and how they motivate Tagged C's control point design. Memory safety policies inform our requirements for memory tags and type casts. Compartmentalization policies depend on the call- and return-related control points, to keep track of the active compartment. Secure information flow policies expose the many places where the user may need to reference identifiers from their program in the policy itself. Taken together, these example policies illustrate Tagged C's breadth of application.

4.1 Memory Safety

Tagged C can be used to enforce memory safety with respect to different *memory models*—formal or informal descriptions of how C should handle memory. Here we discuss the CompCert C memory model and two models proposed by Memarian et al. [24] for the purposes of supporting low-level idioms in the presence of compiler optimization, focusing in particular on how they handle casts from pointers to integers and back.

While the idea of a valid pointer may seem obvious, the precise definition can vary. The C standard does not support arbitrary arithmetic on pointers or their integer casts. In practice, it is common for programs to violate the C standard to various degrees; see Fig. 4. For example, if objects are known to be aligned

to 2^n-byte boundaries, the low-order n bits of pointers can be "borrowed" to store other data [25]. The possible presence of these low-level idioms means that there is no one-size-fits all memory safety policy. CompCert C's definition of a valid pointer allows the pointer to be cast into an integer and back, but only if its value does not change in the interim. This is very strict! Programs that use low-level idioms would failstop if run under a policy that enforces this.

Memarian et al.'s first memory model, *provenance via integer* (PVI), treats memory as a flat address space, and pointers as integers with additional provenance information associating them to their objects. Pointers maintain this provenance even through casts to integers and the application of arithmetic operations. When cast back, the pointers will still be associated with the same object. This enables many low-level idioms, while still forbidding memory-safety violations like buffer overflows.

On the other hand, their second model, *provenance not via integer* (PNVI), clears the provenance of a pointer when it is cast to an integer. When an integer is cast to a pointer (whether or not it was previously derived from a pointer), it takes on the provenance of whatever it points to at that time. The security properties of this memory model are questionable, but it is a realistic option for a compiler to choose and can support idioms that PVI cannot.

Implementation The basic idea for enforcing any of the above memory safety variants is a "lock and key" approach [11,5]. When an object is allocated, it is assigned a unique "color," and its memory locations as well as its pointer are tagged with that color, written $\textsc{clr}(c)$. The default tag $\textsc{n}$ indicates a non-pointer or non-allocated location. The PC tag will also be $\textsc{clr}(c)$, tracking the next available color for new allocations. These rules are given in Figs. 5 and 6. Operations that are valid on pointers in a given memory model maintain the pointer's color, and loads and stores are legal if it matches the target memory location tag. (The `assert` command failstops if its argument does not hold.)

The specific memory models (Fig. 6) behave differently when pointers are cast to integers and back. The CompCert C variant marks that the integer has been cast from a valid pointer, and restores that provenance when cast back. But **BinopT** will failstop if the integer is actually modified between the casts. PVI simply keeps the provenance and allows all operations between casts. PNVI accesses the memory pointed to by the cast pointer and takes its location tag.

Memory safety considerations also inform the design of control points related to allocating, deallocating, and accessing memory. Drawing from CompCert C, Tagged C abstracts over "external" functions like `malloc` and `free`, rather than treat their implementation as ordinary code. In a concrete system, these would be replaced by their library equivalents. **ExtCallT** models the desired tag behavior of general external functions in the Tagged C semantics, but $\textbf{MallocT}(\mathcal{P}, pt, vt)$ and $\textbf{FreeT}(\mathcal{P}, pt, vt)$ are special cases because they also need to retag memory; the location tag lt returned by each of them is copied across the allocated region.

Temporal Memory Safety The tag rules described so far only enforce spatial memory safety, but Tagged C can also enforce temporal safety. A full memory

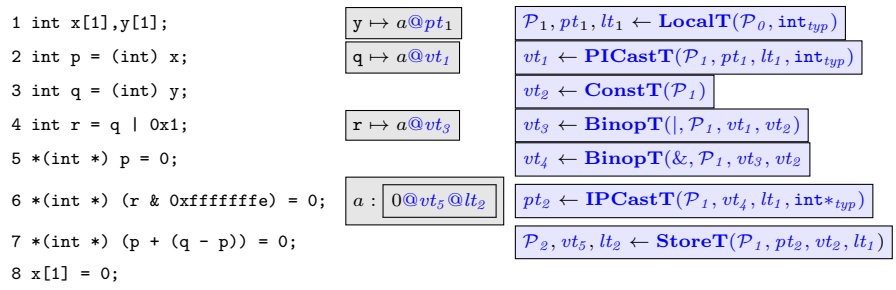


Fig. 4: Memory safety and pointer casts, tracing `y`, `q`, and `r`. (Assume `int` and pointers are 32 bits.) Line (5) is always legal, (6) is illegal in CompCert C due to bitwise arithmetic not preserving provenance, (7) is also illegal in PVI due to combining provenance of multiple objects, and (8) is illegal in all models.

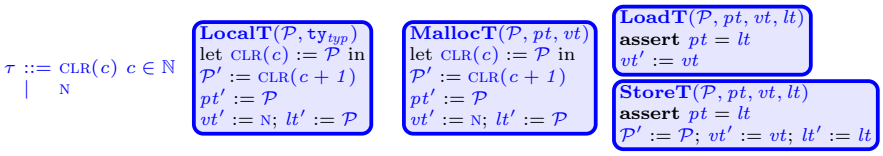


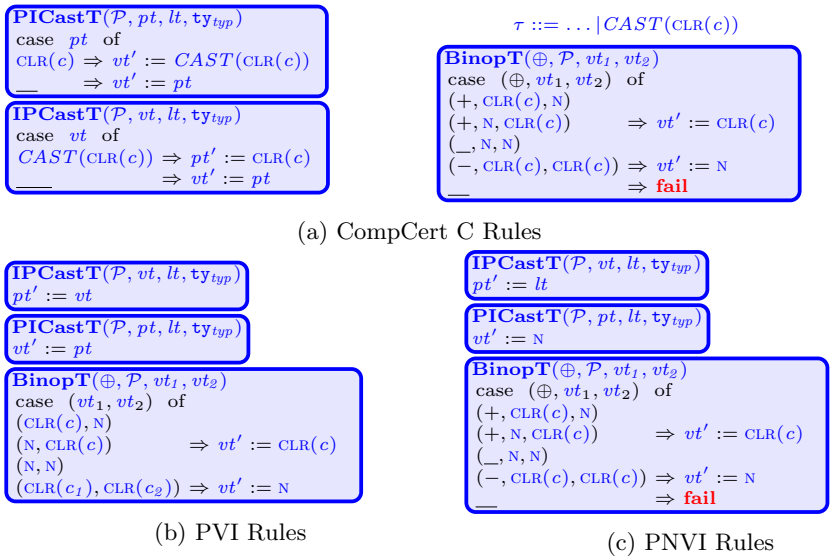
Fig. 5: Generic Memory Safety Rules

safety policy prevents use-after-free and double-free errors by either retagging a deallocated region, or using the PC tag to track the set of live objects and revoking permissions on an object as soon as it is freed.

4.2 Compartmentalization

In principle, the monitoring techniques in the previous section could be used to detect all unintended memory safety violations (albeit only at run time) and ultimately to fix them. But in reality, the cost and risk of regressions may make it undesirable to fix bugs in older code [7]. A compartmentalization policy can isolate potentially risky code, such as code with unfixed (or intentional) UB, from safety-critical code, and enforce the *principle of least privilege*. Even in the absence of language-level errors, compartmentalization can usefully restrict how code in one compartment may interact with another. External libraries are effectively required for most software to function yet represent a supply-chain threat; isolating them prevents vulnerabilities in the library from compromising critical code, and limits the tools available to attackers in the event of a compromise.

Assume we have been given a compartmentalization policy, with at least two compartments, to add to the system after development. The compartments and what belongs in them are represented in the policy by a set of compartment identifiers, ranged over by C, and a map from function and global identifiers to compartments, written as $comp(id)$.



(a) CompCert C Rules

(b) PVI Rules

(c) PNVI Rules

Fig. 6: Specialized Memory Safety Rules

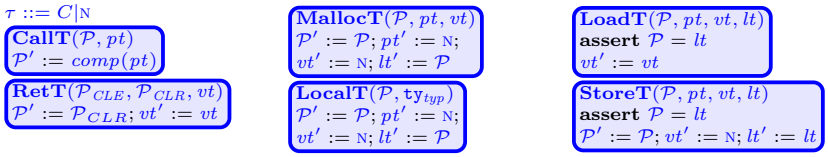


Fig. 7: Simple Compartmentalization Policy

Implementation Compartmentalization requires the policy to keep track of the active compartment, which means keeping track of function pointers. In Tagged C, function pointers are the exception to the concrete memory model. They carry symbolic values that refer uniquely to their target functions. If `f` is located at the symbolic address α, then the expression `&f` evaluates to $\alpha @ \mathtt{f}_{fun}$. When the function pointer is called, Tagged C invokes **CallT**$(\mathcal{P}, pt)$, where pt is the function pointer's tag, to update the PC tag. On return, in addition to handling the return value (if any), **RetT**$(\mathcal{P}_{CLE}, \mathcal{P}_{CLR}, vt)$ determines a new PC tag based on the one before the call ($\mathcal{P}_{CLR}$) and the one at the time of return ($\mathcal{P}_{CLE}$). In our compartmentalization policy (Fig. 7), we define a tag to be a compartment identifier or the default N tag. The PC tag always carries the compartment of the active function, kept up to date by the **CallT** and **RetT** rules.

Once the policy knows which compartment is active, it must ensure that compartments do not interfere with one another's memory. A simple means of doing so is given in Fig. 7: any object allocated by a given compartment, whether on the stack or via `malloc`, is tagged with that compartment's identity, and can only be accessed while that compartment is active. This is very limiting,

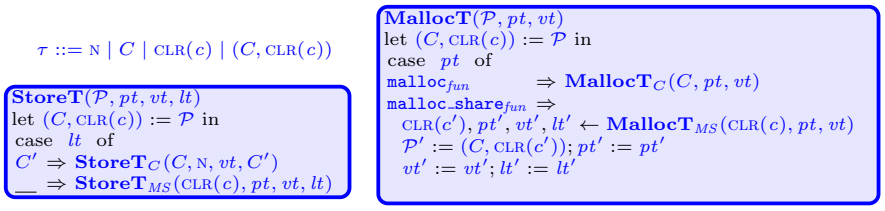


Fig. 8: Selected rules for Compartmentalization with Shared Capabilities, combining Compartmentalization rules (subscript C, Fig. 7) with Memory Safety rules (subscript MS, Fig. 5)

however! In practice, compartments need to be able to share memory, such as in the common case where libraries have separate compartments from application code. One solution is to allow compartments to share selected objects by passing their pointers, treating them as *capabilities*—unforgeable tokens of privilege.

Distinguishing shareable memory from memory that is local to a compartment is difficult without modifying source code. In order to be minimally intrusive, we create a variant identifier for `malloc`, `malloc_share`, which maps to the same address (i.e., it still calls the same function) but has a different name tag and can therefore be used to specialize the tag rule. An engineer might manually select which allocations are shareable, or perhaps rely on some form of escape analysis to detect shareable allocations automatically.

The policy in Fig. 8 essentially combines the simple compartmentalization policy and a memory safety policy. The PC tag carries both the current compartment color, for tagging unshared allocations, and the next free color, for tagging shared allocations. **MallocT** applies a color tag to shareable allocations, and N to local ones. During loads and stores, the location tag of the target address determines which parent property applies.

Compartmentalization Variants Using program-specific tags for globals and functions, a policy like the one above can be extended with a Mandatory Access Control (MAC) policy [20]. Here, a table explicitly identifies which compartments may call one another's functions, which global variables they can access, and with which other compartments they can share memory.

Malloc Limitations The way Tagged C currently handles `malloc` is unsatisfactory. First, as noted above, there is no easy way to distinguish different static `malloc` call locations; our use of variant names is something of a hack. A more principled solution might draw on pointcuts to identify specific calls to `malloc`. Further, `malloc` does not get access to type information; it takes just a size and returns a `void *`, which the caller must cast to a pointer of the desired type (at an arbitrary future point). Therefore, Tagged C cannot easily enforce substructural memory safety (i.e. protecting fields within a single struct from overflowing into each other) or other properties that call for allocated regions to be tagged

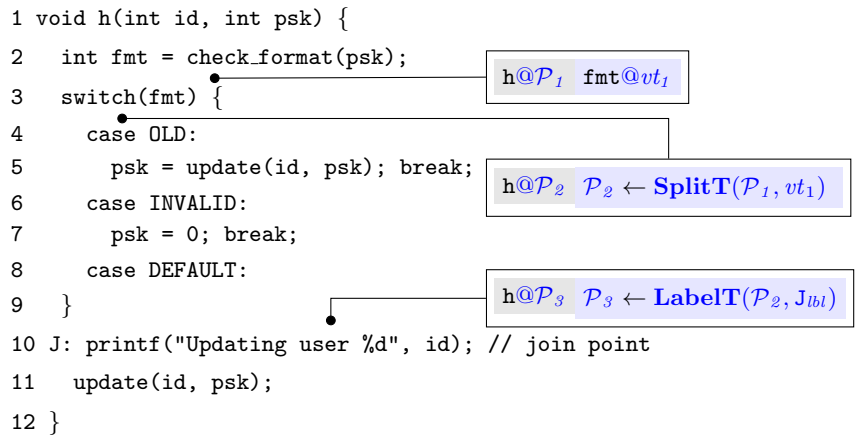


Fig. 9: Not an Implicit Flow

according to their types. This is a well-known impediment to improving C memory safety; previous work (e.g. [26]) has often adopted non-standard versions of `malloc` that take more informative parameters. This is not satisfactory for protecting legacy code, but we do not yet see a good alternative.

4.3 Secure Information Flow

Finally, we return to the family of *secure information flow* (SIF) [14] policies introduced in Section 2. SIF deals with enforcing higher-level security concerns, so it useful even in code with no language-level errors.

In Fig. 9, the program checks the format of the passkey `psk`, which is tagged H, and uses a switch statement to perform operations on it based on the result. As in Fig. 2, this means that the policy should "raise" the PC tag to H to indicated that the program's control-flow depends on `psk`. But after control reaches label J, the PC tag can be lowered again, because code execution from this point on no longer depends on `psk`. J is a *join point*: the point in a control-flow graph where all possible routes from the split to a return have re-converged, which can be identified statically as the immediate post-dominator of the split point [14].

In order to support policies that reason about splits and joins, we introduce the **SplitT** and **LabelT** tag rules. Every transition that tests a value as part of a conditional or loop contains a control point that invokes **SplitT**, passing the label for the corresponding joint point. (This label argument is optional, both because some policies may not care about join points.) This way, the policy can react when execution reaches that label via the **LabelT** rule.

In the full SIF policy, we keep track of the pending join points within the PC tag, and lower the PC tag when execution reaches the join point. (A similar approach applies to conditional expressions, but we omit the details here.) In

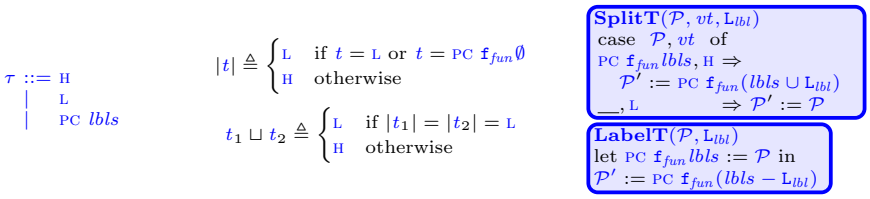


Fig. 10: SIF Conditionals

addition to L and H, the SIF policy tracks a PC tag written using the constructor PC, which carries a set of label identifiers to record the join points of tainted statement scopes. Initially, the PC tag is PC $\mathtt{f}_{fun}\emptyset$, which corresponds to "low" security. The join operator, $\cdot \sqcup \cdot$, takes the higher of its arguments after reducing a PC tag into either H or L.

In order to use this version of SIF, the program must undergo a minor automatic transformation by the compiler or interpreter, introducing explicit labels at all join points that don't already have one. In the example, `J` becomes an explicit label in the code. The internal syntactic form of each conditional statement (`if`, `switch`, `while`, `do-while`, and `for`) carries this label (optionally, since there might not be a join point if all arms of the conditional execute an explicit `return`). If the conditional branches on a high value, **SplitT** adds the label to the set in the PC tag. Later, when execution reaches a label, **LabelT** deletes it from the set. If the set is non-empty, there is at least one high split point that has not yet reached its join point, so we treat the PC tag as high. When execution reaches the last join point and the set is empty, the PC tag is treated as low, because it is no longer possible to deduce which path was taken.

SIF Variants SIF can cover many different policies. We have shown an instance of a confidentiality policy, but SIF can also support integrity ("insecure inputs do not affect secure data"), intransive policies ("data can flow from A to B and B to C, but not from A to C"), and policies with more than two security levels.

To give a couple of more realistic examples, an intransitive integrity policy can be used to protect against SQL injections by requiring unsafe inputs to pass through a sanitizer before they can be appended to a query. Similarly, a more complex SIF policy could ensure that data at rest is always encrypted, by setting a low security level to the outputs of an encryption routine and a high level to the outputs of its corresponding decryption routine.

5 Implementation

Our current Coq-based implementation of Tagged C consists of a formal semantics and a matching interpreter written in Coq's Gallina language (similar to functional languages such as OCaml or Haskell). These are based on the `Csem` and `Cexec` modules from the CompCert compiler (version 3.10) [21]. The in-

terpreter can be extracted to a stand-alone OCaml program that can then be further compiled to machine code.

To adapt CompCert, we replace the standard block-offset memory with a concrete one, leaving the block-based system to handle only function pointers. We rework global environments to separate (symbolic) function pointers from other (concrete) pointers. We also add a temporary environment to contain non-memory variables, and semantic rules to deal with them. Most importantly, we thread the PC tag into the state, and add control points to the relevant semantic rules. These changes appear in both the semantics and the corresponding interpreter code. To extend the existing CompCert proof that the interpreter is correct with respect to the semantics requires updating the proof automation to handle concrete memory and tags.

The semantics and interpreter are parameterized by a Coq module type `Policy`, which specifies the type signatures of the tag rules. A policy is written directly in Gallina as a module that instantiates `Policy` by defining the type of tags and the body of each tag rule. A full policy fits into 70 lines of Gallina. To illustrate, Fig. 11 shows fragments of the `Policy` signature, its instantiation in the PVI memory safety policy, and its use in the Coq semantics.

PIPE, tag sizes, and policy states. Ultimately, we wish to compile Tagged C to run efficiently on PIPE-equipped hardware, which raises some issues that are not currently visible at the level of the Coq interpreter. For example, the Coq model of Tagged C allows unbounded tags. In reality, PIPE tags are large, but bounded. This means that, for instance, the naïve implementation of PVI memory safety described here runs the risk of overflowing the number of possible colors. Enforcement of temporal safety will not be feasible in long-running programs that regularly allocate memory, even if their total memory footprint is bounded, unless the policy can reclaim the tags on previously freed objects.

Also, the Coq model assumes that all tag rules are pure functions of the tag inputs; any state carried by the policy must be encoded in the PC tag. In practice, PIPE allows tag rules to be implemented by arbitrary code, which can persist private state (separate from the application being monitored) over multiple rule invocations; this approach may support more efficient policy designs.

6 Related Work

Like many monitoring systems, Tagged C can be seen as a species of aspect-oriented programming (AOP) [19]. An AOP language distributes *join points*[5] throughout its semantics, and the programmer separately writes *advice* in the form of additional code that should execute before or after various join points according to a *pointcut* specification. The compiler or runtime *weaves* the advice together with the main code. Our control points are a kind of join point, and our tag rules combine the roles of pointcuts and advice; weaving is done at runtime

[5] Not to be confused with the control-flow graph join points discussed in Section 4.3.

```
Module Type Policy.
   Parameter tag : Type.                      Inductive PolResult (A: Type) :=
   Parameter def_tag : tag.                   | PolSuccess (res: A)
   Parameter InitPCT : tag.                   | PolFail (r: string)

   Parameter BinopT : binary_operation -> tag -> tag -> tag -> PolResult (tag * tag).
   Parameter LoadT : tag -> tag -> tag -> list tag -> PolResult tag.
   ...

Module PVI:Policy.
   Inductive tag :=                           Definition def_tag := N.
   | N.                                       Definition InitPCT := Dyn 0.
   | Dyn (c:nat).

   Definition BinopT op pct vt1 vt2 :=        Definition LoadT pct pt vt lts :=
     match vt1, vt2 with                        match pt with
     | Dyn c, N => PolSuccess (pct, Dyn c)      | N => PolFail "PVI::LoadT N"
     | N, Dyn c => PolSuccess (pct, Dyn c)      | _ => if forallb (=? pt) lts
     | N, N => PolSuccess (pct, N)                     then PolSuccess vt
     | Dyn c1, Dyn c2 => PolSuccess (pct, N)           else PolFail "PVI::LoadT Ptr"
     end.                                       end.
   ...

Module Csem (Import P: Policy).

  Inductive rred (PCT:tag) : expr -> mem -> trace -> tag -> expr -> mem -> Prop :=
  | red_binop: forall op v1 vt1 ty1 v2 vt2 ty2 ty m v vt' PCT',
    sem_binary_operation (snd ge) op v1 ty1 v2 ty2 m = Some v ->
    PolSuccess (PCT', vt') = BinopT op PCT vt1 vt2 ->
    rred PCT (Ebinop op (Eval (v1,vt1) ty1) (Eval (v2,vt2) ty2) ty) m E0
     PCT' (Eval (v,vt') ty) m
  | red_rvalof: forall ofs pt lts bf ty m tr v vt vt',
    deref_loc ty m ofs pt bf tr (v,vt) lts ->
    PolSuccess vt' = LoadT PCT pt vt lts -> PolSuccess vt'' = AccessT PCT vt'
    rred PCT (Evalof (Eloc ofs pt bf ty) ty) m tr PCT (Eval (v,vt'') ty) m
  | ...
  ...
```

Fig. 11: Fragments of Coq implementation.

according to the tagged semantics. Unlike advice in most AOP systems, our tag rules are constrained to inspect only tags, not arbitrary parts of program state, which limits their expressiveness. Also, tag rules are evaluated separately from the system being monitored and so cannot be used to "correct" bad behavior; all they can do is cause a failstop. These limitations follow from our goal of implementing Tagged C using efficient PIPE hardware.

Many AOP-like systems for C runtime verification treat join points as events in a trace, and specify valid traces using a formalism such as state machines (e.g. RMOR [18] and SLIC [6]) or temporal logics (e.g. [9]). Trace checking of this kind can be implemented on top of Tagged C, as long as events do not rely on values. Events are typically coarse-grained (e.g. function entries and exits), although some systems (e.g. [16]) support very general forms of event definition based on matching syntactic patterns in code. Tagged C is unusual in that it supports very low-level and fine-grained events (e.g. individual arithmetic operations and casts) and because a monitoring action (perhaps a no-op) is specified for every potential event point.

Numerous systems have targeted information flow, memory safety, and compartmentalization in C; we can discuss just a few here. Cassel et al.'s FlowNotations [8] use type annotations to specify "tainted" and "trusted" data, and statically check a program's information flow using the C type system. Their annotation system elegantly connects the C syntax to their enforcement mechanism, and would make a good annotation scheme for a Tagged-C SIF policy, with "tainted" and "trusted" types being transformed into variable-specific tags. Unlike their static approach, our enforcement is dynamic, meaning that it sacrifices flow-sensitivity for permissiveness [28] Dynamic systems also exist, such as Faceted Information Flow [3], which takes advantage of concurrency to simulate multiple simultaneous runs and check directly for leaked data. Faceted IFC has not been applied at the C level, and for our use cases, would suffer from the overhead of running multiple executions simultaneously.

The CHERI hardware capability system has been used by Tsampas et al. for compartmentalization [29], and by Filardo et al. for temporal memory safety [30]. Like PIPE, CHERI can support a range of security policies, although it is ill-suited for information-flow-style policies. Despite this, it would be worth exploring whether a useful subset of Tagged C's control points could be implemented by a CHERI backend.

7 Conclusion and Future Work

We have introduced a C variant that provides a general mechanism to describe security policies, exemplified by memory safety, compartmentalization, and secure-information-flow policies. Each category of policy can be applied flexibly to meet the security needs of a particular program. From this proof of concept, we can see several natural extensions to make Tagged C more practical to use.

An interpreter is useful for testing policies, but our main goal has always been to produce a compiler from Tagged C to machine code for a PIPE-equipped processor. The basic strategy for compilation was outlined in the Tagine project [10]. We are currently working to extend the CompCert compiler to handle Tagged C, with the ultimate goal of also extending CompCert's semantics preservation guarantees to cover tagged semantics. Policies are also written in Gallina, the language embedded in Coq [12]. This is fine for a proof-of-concept, but not satisfactory for real use by software engineers. We plan to develop a domain-specific policy language to make it easier to write Tagged C policies.

One reason for prototyping Tagged C in the Coq Proof Assistant is to lay the groundwork for formal proofs of its properties. We have not yet proven the correctness of our example policies. For each family of policies that we discuss, we aim to give a higher-level formal specification (e.g., a non-interference property for SIF) and prove that it holds on all programs run under that property.

Acknowledgements We thank the reviewers for their valuable feedback, and Roberto Blanco for his advice during the writing process. This work was supported by the National Science Foundation under Grant No. 2048499, Specifying and Verifying Secure Compilation of C Code to Tagged Hardware.

References

1. Anderson, J.P.: Computer security technology planning study. Technical Report ESD-TR-73-51, U.S. Air Force Electronic Systems Division (Oct 1972), http://csrc.nist.gov/publications/history/ande72.pdf
2. Armv8.5-a memory tagging extension white paper, https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf
3. Austin, T.H., Flanagan, C.: Multiple facets for dynamic information flow. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 165–178. POPL '12, Association for Computing Machinery (2012), https://doi.org/10.1145/2103656.2103677
4. Azevedo de Amorim, A., Collins, N., DeHon, A., Demange, D., Hritcu, C., Pichardie, D., Pierce, B.C., Pollack, R., Tolmach, A.: A verified information-flow architecture. Journal of Computer Security **24**(6), 689–734 (2016), http://dx.doi.org/10.3233/JCS-15784
5. Azevedo de Amorim, A., Dénès, M., Giannarakis, N., Hritcu, C., Pierce, B.C., Spector-Zabusky, A., Tolmach, A.P.: Micro-policies: Formally verified, tag-based security monitors. In: 2015 IEEE Symposium on Security and Privacy. pp. 813–830 (May 2015), http://dx.doi.org/10.1109/SP.2015.55
6. Ball, T., Rajamani, S.: SLIC: A specification language for interface checking (of C). Tech. Rep. MSR-TR-2001-21 (January 2002), https://www.microsoft.com/en-us/research/publication/slic-a-specification-language-for-interface-checking-of-c/
7. 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
8. Cassel, D., Huang, Y., Jia, L.: Uncovering information flow policy violations in C programs (extended abstract). In: Proc. Computer Security – ESORICS 2019, Part II. p. 26–46. Springer-Verlag (2019), https://doi.org/10.1007/978-3-030-29962-0_2
9. Chabot, M., Mazet, K., Pierre, L.: Automatic and configurable instrumentation of c programs with temporal assertion checkers. In: 2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE). pp. 208–217 (2015). https://doi.org/10.1109/MEMCOD.2015.7340488
10. 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 (2021), https://doi.org/10.1145/3437992.3439929
11. Clause, J., Doudalis, I., Orso, A., Prvulovic, M.: Effective memory protection using dynamic tainting. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering. p. 284–292 (2007), https://doi.org/10.1145/1321631.1321673
12. Coq Team: The Coq proof assistant, https://coq.inria.fr
13. Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**(5), 236–243 (May 1976), https://doi.org/10.1145/360051.360056
14. 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

15. Dhawan, U., Hritcu, C., Rubin, R., Vasilakis, N., Chiricescu, S., Smith, J.M., Knight, Jr., T.F., Pierce, B.C., DeHon, A.: Architectural support for software-defined metadata processing. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 487–502 (2015), http://doi.acm.org/10.1145/2694344.2694383
16. Engler, D.R., Chelf, B., Chou, A., Hallem, S.: Checking system rules using system-specific, programmer-written compiler extensions. In: OSDI. pp. 1–16 (2000)
17. Gollapudi, R., Yuksek, G., Demicco, D., Cole, M., Kothari, G.N., Kulkarni, R.H., Zhang, X., Ghose, K., Prakash, A., Umrigar, Z.: Control flow and pointer integrity enforcement in a secure tagged architecture. In: 2023 IEEE Symposium on Security and Privacy (SP). pp. 2974–2989 (May 2023), https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00102
18. Havelund, K.: Runtime verification of C programs. vol. 5047, pp. 7–22 (01 2008). https://doi.org/10.1007/978-3-540-68524-1_3
19. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP'97 — Object-Oriented Programming. pp. 220–242. Springer (1997)
20. Lampson, B.W.: Protection. SIGOPS Oper. Syst. Rev. **8**(1), 18–24 (Jan 1974), https://doi.org/10.1145/775265.775268
21. Leroy, X.: Compcert 3.10, https://github.com/AbsInt/CompCert/releases/tag/v3.10
22. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (Jul 2009), https://doi.org/10.1145/1538788.1538814
23. Leroy, X.: A formally verified compiler back-end. J. Autom. Reason. **43**(4), 363–446 (dec 2009), https://doi.org/10.1007/s10817-009-9155-4
24. Memarian, K., Gomes, V.B.F., Davis, B., Kell, S., Richardson, A., Watson, R.N.M., Sewell, P.: Exploring C semantics and pointer provenance. Proc. ACM Program. Lang. **3**(POPL) (Jan 2019), https://doi.org/10.1145/3290380
25. Memarian, K., Matthiesen, J., Lingard, J., Nienhuis, K., Chisnall, D., Watson, R.N.M., Sewell, P.: Into the depths of C: Elaborating the de facto standards. SIGPLAN Not. **51**(6), 1–15 (Jun 2016), https://doi.org/10.1145/2980983.2908081
26. Michael, A.E., Gollamudi, A., Bosamiya, J., Johnson, E., Denlinger, A., Disselkoen, C., Watt, C., Parno, B., Patrignani, M., Vassena, M., Stefan, D.: MSWasm: Soundly enforcing memory-safe execution of unsafe code. Proc. ACM Program. Lang. **7**(POPL) (Jan 2023), https://doi.org/10.1145/3571208
27. Roessler, N., DeHon, A.: Protecting the stack with metadata policies and tagged hardware. In: Proc. 2018 IEEE Symposium on Security and Privacy, SP 2018. pp. 478–495 (2018), https://doi.org/10.1109/SP.2018.00066
28. Russo, A., Sabelfeld, A.: Dynamic vs. static flow-sensitive security analysis. In: 2010 23rd IEEE Computer Security Foundations Symposium. pp. 186–199 (2010). https://doi.org/10.1109/CSF.2010.20
29. Tsampas, S., El-Korashy, A., Patrignani, M., Devriese, D., Garg, D., Piessens, F.: Towards automatic compartmentalization of C programs on capability machines (2017), https://api.semanticscholar.org/CorpusID:32838507
30. 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