Lazy Stack Properties

Sean Anderson, Andrew Tolmach

April 27, 2020

1 Introduction

Subroutine and coroutine models suggest logical encapsulation of stack frames that is not typically enforced in low level code. With an additional enforcement mechanism such as a tag-based reference monitor, we can prove integrity and confidentiality properties of subroutines and coroutines. The most obvious integrity and confidentiality properties reflect the behavior of eager enforcement mechanisms that prevent illegal writes and writes immediately. But lazy tag policies (see Roessler and DeHon, Section IV.A [**?**]) do not enforce these eager properties. In this paper we present properties that capture the inability of unauthorized reads and writes to interfere with observable behavior.

In Section 2 we model a machine as a transition system with built-in coroutine and subroutine operations. Section 3 gives the specification of integrity in terms of accessible and inaccessible memory, then Sections 4 and 5 translates this specification into a property that captures when a violation would effect observable behavior.

Our lazy integrity property, *observable integrity*, defines an idealized "safe" variant machine that fulfills eager integrity properties, but otherwise behaves identically to the original machine. The safe machine rolls back writes to inaccessible locations when they become accessible. Our lazy property is that the observable behavior of the idealized machine extends that of the reference machine.

Eager vs. Lazy Enforcement For illustrative purposes, consider the following program fragments in a simple language with subroutines. Suppose that addresses 20-29 are the stack frame of our main routine:

```
0:  call 10                          0:  call 10
1:  if *20 goto 4                    1:  if *20 goto 4
2:  print 0                          2:  print 0
3:  halt                             3:  halt
4:  print 1                          4:  print 1
5:  halt                             5:  halt
⋮                                    ⋮
10:  *21 := 0                        10:  *20 := 0
11:  return                          11:  return
```

The subroutine at 10 writes into its caller's stack, which is erroneous. An eager enforcement mechanism might be to tag 20-29 with a unique *color* matching a tag on the program counter (the *PC Tag*), change the tag on the program counter during the call out, and cause the system to fail-stop anytime a write occurs to an address not matching the program counter's tag. Then both programs will fail-stop. But this would require writing the color to all 10 addresses, which is expensive if most of those addresses are not actually used.

In the lazy variation we construct our system to initially give all memory a tag of *init* and cause writes to tag the written location with a color matching that of the program counter. We further cause reads to fail-stop if the value being read does not match the program counter's. So failure occurs only when a location is read by a routine other than the one that wrote it. Note that the left program will not fail-stop under this policy, as address 21 is overwritten by the callee but never read. The right program, on the other hand, still fails.

Both of these policies will be described in more detail in Section 6. The lazy mechanism enforces a weaker property than that of the eager mechanism since it fails fewer programs. Intuitively, it seems clear that this property still reflects a desirable notion of integrity; while the left program erred, it did so invisibly.

2 The Model

First, some types. `PC` is the program counter. We have words and addresses:

$$w, a \in \mathcal{W}$$

Registers:

$$r \in \mathcal{R}$$

Stack names:

$$S \in \mathcal{S}$$

A stack is specified by a 4-tuple of addresses: base, frame, pointer, and top. Stack names are mapped to stacks by a stack configuration:

$$\sigma \in \Sigma = \mathcal{S} \to \mathcal{W} \times \mathcal{W} \times \mathcal{W} \times \mathcal{W}$$

Machine Components:

$$K \in \mathcal{W} \cup \mathcal{R} \cup \{\texttt{PC}\} \cup \{\texttt{FP}\} \cup \{\texttt{SP}\}$$

States:

$$M \in \mathcal{M} = \mathcal{K} \to \mathcal{W}$$

Machine configurations:

$$T \in \mathcal{T} = \mathcal{M} \times \Sigma \times \mathcal{S}$$

If $T = M, \sigma, S$ and $\sigma(S) = b, f, p, t$, then $M(\texttt{FP})$ is f and $M(\texttt{SP})$ is p. These special registers only change when σ or S changes.

The step relation $\longrightarrow$ represent the execution of a single instruction or pseudo-instruction, $\cdot \overset{\cdot}{\to} \cdot \subseteq \mathcal{T} \times (\mathcal{W} \cup \{\tau\}) \times \mathcal{T}$. It may be labeled $\xrightarrow{w}$ or silent $\xrightarrow{\tau}$, which we will write $\longrightarrow$. For generality we leave the set of instructions open except to specify the existence of pseudo-instructions for coroutine yields, subroutine calls and returns, and output.

A yield instruction `Yield` S saves the `PC` to the current stack, changes the current stack to S, and loads the `PC` from the new stack.

The call instruction `Call` w a pushes a frame of size w onto the stack and sets the `PC` to address a. The return instruction `Return` pops the top frame of the current stack and sets the `PC` to the next instruction from a matched call – see below.

An output instruction, `Output` r, takes a register r and steps by a transition labeled with the contents of r. It is the only instruction that can produce a labeled transition.

An execution is a sequence of configurations and labels, possibly infinite, of the form $T_0 w_0 T_1 w_1 \ldots$, such that $T_i \xrightarrow{w_i} T_{i+1}$. We will omit w_i when it is τ. The behavior of an execution, or its trace, written $\mathit{behav}(\overline{T})$, is the projection of the labels of $\overline{T}$. A *property* is a set of executions, and a system enjoys those properties for which all of its possible executions are in the property.

Pseudoinstruction Semantics Here we describe the behavior of the call and return pseudoinstructions. Stacks grow upward; initially each stack's state is $b, b, b+2, t$ where b is its base and t its top, and no stacks overlap. A separate code section is non-overlapping with the stacks. The program counter is initially in the code section.

$$\frac{\begin{array}{cc} M_1(M_1(\texttt{PC})) = \texttt{Call w a} & \sigma(S) = b, f_1, p_1, t \\ f_2 = p_1 & p_2 = f_2 + \texttt{w} + 2 \\ M_2 = M_1[f_2 \mapsto M_1(\texttt{PC})][f_2 + 1 \mapsto f_1][\texttt{PC} \mapsto a] & \sigma_2 = \sigma_1[S \mapsto b, f_2, p_2, t] \end{array}}{M_1, \sigma_1, S \longrightarrow M_2, \sigma_2, S}$$

$$\frac{\begin{array}{ccc} M_1(M_1(\texttt{PC})) = \texttt{Return} & \sigma(S) = b, f_1, p_1, t & \\ M_2 = M_1[\texttt{PC} \mapsto M_1(f_1)] & f_2 = M_1(f_1 + 1) & \sigma_2 = \sigma_1[S \mapsto b, f_2, f_1, t] \end{array}}{M_1, \sigma_1, S \longrightarrow M_2, \sigma_2, S}$$

In an execution $\overline{T} = \ldots T_1 \ldots T_2 \ldots$, if $T_1 = M_1, \sigma_1, S$ and $T_2 = M_2, \sigma_2, S$, where $\sigma_1(S) = \sigma_2(S)$ and $T_1(\texttt{PC}) = T_2(\texttt{PC})$ — and if for all states $T' = M', \sigma', S$ between T_1 and T_2, $\sigma'(S) \neq \sigma_1(S)$ — T_1 and T_2 are said to be *call brackets*. We will use this concept to define an eager notion of integrity.

Now we describe the behavior of the yield pseudoinstruction.

$$\frac{\begin{array}{cc} M_1(M_1(\texttt{PC})) = \texttt{Yield} S_2 & \sigma(S_1) = b, f_1, p_1, t \\ \sigma(S_2) = b_2, f_2, s_2, t_2 & M_2 = M_1[s_2 \mapsto M_1(\texttt{PC})][\texttt{PC} \mapsto M_1(s_2)] \end{array}}{M_1, \sigma, S_1 \longrightarrow M_2, \sigma, S_2}$$

In an execution $\overline{T} = \ldots T_1 \ldots T_2 \ldots$, the T_1 and T_2 are considered *yield brackets* if the have the same active stack, and if every state between them has a different stack. Again this will be used in eager notions of integrity.

Finally, output instructions generate a labeled transition.

$$\frac{M(M(\texttt{PC})) = \texttt{Output } r}{M, \sigma, S \xrightarrow{M(r)} M[\texttt{PC} \mapsto \texttt{PC} + 1], \sigma, S}$$

3 Security Levels and Contours

At any given state we assign each component a security level. It is either accessible to the active subroutine within the active coroutine (`A`), accessible but unallocated (`AL`), inaccessible due to belonging to a caller in the local stack (`IL`), inaccessible due to belonging to a foreign coroutine(`IF`), or control/code (`K`). A contour $C \in \mathcal{C} = \mathcal{K} \to \{\mathtt{A}, \mathtt{AL}, \mathtt{IL}, \mathtt{IF}, \mathtt{K}\}$ maps components to security levels, and within our model there is a clear mapping from pairs of stack configurations and active stack ids to contours. Here *control*(T) is the set of addresses reserved for control information: the first two addresses of each frame and the top of each inactive stack area.

$$\mathit{contour}(\sigma, S) = \text{Let } \sigma(S) = b, f, p, t \text{ in}$$
$$\lambda K. \begin{cases} \mathtt{A} & \text{if } K \notin \mathcal{W} \\ \mathtt{K} & \text{else if } K \in \mathit{control}(T) \\ \mathtt{A} & \text{else if } f \leq K < p \\ \mathtt{AL} & \text{else if } p \leq K < t \\ \mathtt{IL} & \text{else if } b \leq K < f \\ \mathtt{IF} & \text{else} \end{cases}$$

In this context we are concerned with protecting data, and will assume control is protected by some lower level policy such that only calls, returns, and yields may update `K` components.

4 Integrity Properties

Local, Frame, and Stack Integrity From a contour as defined above, we have an obvious notion of integrity: a transition system enjoys *local integrity* when, for all $T_1 = M_1, \sigma_1, S_1$ and $T_2 = M_2, \sigma_2, S_2$, if $T_1 \xrightarrow{w} T_2$ and $M_1(K) \neq M_2(K)$, $\mathit{contour}(\sigma_1, S_1)(K) = \{\mathtt{A}, AL\}$. In other words, individual state transitions respect the contour of the first state, only modifying memory locations within the active stack above the frame pointer. This property is very strong and convenient to test for eager enforcement mechanisms as it only talks about a single step.

Local integrity implies a pair of marginally weaker properties that we call frame integrity and stack integrity. A system enjoys *frame integrity* if for all K and $\overline{T} = \ldots T_1 \ldots T_2 \ldots$ where T_1 and T_2 are a call bracket, if $\mathit{contour}(T_1)(K) = \mathtt{A}$ then $T_1(K) = T_2(K)$. In other words, when we make a call, our accessible state is unchanged on return.

A system has *stack integrity* if, for all K and all $\overline{T} = \ldots T_1 \ldots T_2 \ldots$ where T_1 and T_2 are a yield bracket for all K, if $\mathit{contour}(\sigma_1, S_1)(K) \in \{\mathtt{IL}, \mathtt{A}, \mathtt{AL}\}$, then $T_1(K) = T_2(K)$. In short, when a we yield control, our entire stack is unchanged when control is yielded back.

Frame integrity weakens local integrity by focusing on changes visible to a caller, and stack integrity on changes visible to an entire coroutine. A system that fulfills both properties may write outside of accessible memory, but only if that memory will somehow be restored by the time it becomes accessible.

Observable Integrity In order for a lazy policy to fulfill an integrity property, the property must be weaker still — specifically, it must allow for writes that violate the contour and remain after control changes, but only in ways that are still invisible. We construct this property by comparing the behavior of a system to the behavior of an idealized system that fulfills frame and stack integrity by construction and that otherwise behaves like $\longrightarrow$. We call this system $\xrightarrow{\textsc{Safe}}$, and it enforces both forms of integrity by rolling back illegal writes to an inactive frame when it becomes active again.

Let $\cdot,\cdot \xrightarrow{\textsc{Safe}} \cdot,\cdot \subseteq \mathcal{T} \times \mathcal{M} \times \mathcal{T} \times \mathcal{M}$. We carry an extra memory configuration remembering the last known value for each component outside the active frame. If $M_1, \sigma_1, S_1 \longrightarrow M_2, \sigma_2, S_2$, then for all $\hat{M}_1$, $(M_1, \sigma_1, S_1), \hat{M}_1 \xrightarrow{\textsc{Safe}} (M_2', \sigma_2, S_2), \hat{M}_2$, where:

$$\hat{M}_2(K) = \begin{cases} M_2(K) & \text{if } \mathit{contour}(\sigma_1, S_1)(K) \in \{\mathtt{A}, \mathtt{AL}\} \\ \hat{M}_1(K) & \text{else} \end{cases}$$

$$M_2'(K) = \begin{cases} \hat{M}_1(K) & \text{if } \mathit{contour}(\sigma_1, S_1)(K) \in \{\mathtt{IL}, \mathtt{IF}\} \text{ and } \mathit{contour}(\sigma_2, S_2)(K) \in \{\mathtt{A}, \mathtt{AF}\} \\ M_2(K) & \text{else} \end{cases}$$

A series of safe states $\overline{U} = (T_0, \hat{M}_0), \ldots$ with adjacent states related by $\xrightarrow{\textsc{Safe}}$ is a *safe execution*, and its behavior is defined as the behavior of the execution formed by projecting the state components of the series:

$$\mathit{behav}(\overline{U}) = \mathit{behav}(\mathit{map}(\mathit{fst}, \overline{U}))$$

A system enjoys *observable integrity* if, for any normal execution $\overline{T} = T_0, \ldots$ with $T_0 = M_0, \sigma_0, S_0$, there exists a safe execution $\overline{U} = (T_0, M_0), \ldots$ such that $\mathit{behave}(\overline{T})$ is a prefix of $\mathit{behave}(\overline{U})$. That is, the safe execution may not get stuck when the normal execution does, but up to that point the behaviors are identical.

Observable Integrity Closer to the Development In the interest of matching the coq development, here is an alternate formulation of observable integrity that talks just about subroutines and is just phrased as operating on traces. Assume that there is just one stack, S. In this version, a system enjoys *observable stack integrity* if for all $\overline{T} = \ldots T_1 T_2 \ldots T_3 \overline{T}_4$ such that $T_1 = M_1, \sigma_1, S$, $T_2 = M_2, \sigma_2, S$, $T_3 = M_3, \sigma_3, S$, and T_1 and T_3 bracket a call, there is a safe state $T_3' = M_3', \sigma_3, S$ where:

$$M_3'(K) = \begin{cases} M_1(K) & \text{if } \mathit{contour}(\sigma_1, S)(K) = \mathtt{A} \text{ and } \mathit{contour}(\sigma_2, S)(K) = \mathtt{IL} \\ M_3(K) & \text{else} \end{cases}$$

And the observable trace $\mathit{behav}(T_3' \overline{T}_4')$ is prefixed by $\mathit{behave}(T_3 \overline{T}_4)$.

This variant of the property ought to be combined with a coroutine equivalent, and it captures the same intuition as the Safe execution version, though it would need to be proven that the two are equivalent.

4.1 Example

Here we examine a more detailed example. We will have two coroutines, with stacks S_1 and S_2, and three routines A, B, and C. Coroutine 1 will start in A, call B, and yield to coroutine 2, which starts in routine C. Let $\sigma_{init}(S_1) = 0, 0, 0, 10$ and $\sigma_{init}(S_2) = 11, 11, 11, 21$. Subroutine A will start at address 30, B at 40, and C at 50. Here is our code, in a RISC-like style that folds some arithmetic into the instructions for simplicity:

```
⋮
30  add SP 3 SP        | allocate some space for local storage
31  store 1 (FP+2)     | FP+2 is 2
32  store 1 (FP+3)     | FP+3 is 3
33  call 2 40          | call B
34  load (FP+2) r1     |
35  load (FP+3) r2     |
36  brne r1 r2 38      | if our first two locals are equal,
37  output 1           | we output 1,
38  halt               | and either way we halt
⋮
40  store 5 (FP-1)     | FP-1 is 4
41  load (FP-2) r3     | FP-2 is 3
42  mov FP r4          | save frame pointer for second coroutine
42  yield S_1          | yield to second coroutine
43  return             |
⋮
50  store r3 (r4-2)    | store the value read from 3 back to 3
51  yield S_1          |
⋮
```

This program makes multiple illegal writes. B writes into a location in A's stack frame, and so does C, but in the former case the location is never read, and in the latter the value written will be the same value that was there before. Under our $\xrightarrow{\textsc{Safe}}$ system, the values at addresses 3 and 4 are rolled back to their initial values, so 3 maps to 1 and 4 maps to 0. Under $\longrightarrow$, 3 maps to 1 and 4 maps to 5. The value at 3 still matches the value at 2 in either run, so we output 1 and then halt. The programs have the same behavior, and observational integrity is maintained.

5 Confidentiality Properties

Confidentiality in general is commonly modeled as a *noninterference* property: if some components are considered secret, a system enjoys noninterference when it produces the same behavior regardless of the initial values of the secret components. We will present confidentiality analogs of each of our integrity properties.

Local Noninterference The strongest form of confidentiality is the dual of local integrity: *local noninterference.* For any transition $T_1 \xrightarrow{w} T_2$, we use the contour of T_1 to determine the accessibility of each component. Local noninterference holds if any inaccessible component can be varied in T_1 without changing the values of any accessible component in T_2. Formally, we define view-preserving equivalence relations between states:

$$\begin{aligned} M_1, \sigma_1, S_1 \sim_L M_2, \sigma_2, S_2 \triangleq\; & S_1 = S_2 \text{ and } \sigma_1(S_1) = \sigma_2(S_2) \\ & \text{and for all } K, \text{ if } \mathit{contour}(\sigma_1, S_1)(K) \neq \texttt{IL} \\ & \text{then } M_1(K) = M_2(K) \end{aligned}$$

$$\begin{aligned} M_1, \sigma_1, S_1 \sim_F M_2, \sigma_2, S_2 \triangleq & S_1 = S_2 \text{ and } \sigma_1(S_1) = \sigma_2(S_2) \\ & \text{and for all } K, \text{ if } \mathit{contour}(\sigma_1, S_1)(K) \neq \texttt{IF} \\ & \text{then } M_1(K) = M_2(K) \end{aligned}$$

$$T_1 \sim T_2 \triangleq T_1 \sim_L T_2 \text{ and } T_1 \sim_F T_2$$

Then a system enjoys local noninterference if for any transition $T_1 \xrightarrow{w} T_2$, for all T_1' such that $T_1 \sim T_1'$, there exists some T_2' such that $T_2 \sim T_2'$ and $T_1' \xrightarrow{w} T_2'$.

Frame and Stack Interference Local noninterference is very strong. Suppose a function writes outside of its accessible memory, then immediately reads from that location. We would not consider that read to be a failure of confidentiality in any reasonable attacker model. We can weaken our notion of confidentiality to reflect that information originating in the active routine need not be secret from it, even if stored in a secret location. We do this by varying the state only once for each activation. Then we need to check that the variation has not impacted the behavior of the execution. Conservatively, we will do so by comparing states at the end of the activation.

The analog of frame integrity is *frame noninterference.* A system enjoys frame noninterference when for any $\overline{T} = \ldots T_1 T_2 \overline{T}' T_3 T_4 \ldots$ where T_1 and T_4 bracket a subroutine, and any T_2' such that $T_2 \sim_L T_2'$, there is some T_3' such that $T_3 \sim_L T_3'$, $T_2' \xrightarrow{\overline{w}}{}^{*} T_3'$, and $\overline{W} = \mathit{behav}(T_2 \overline{T}' T_3)$. Note that we use $\sim_L$, because a subroutine containing a yield might well have its behavior depend on a foreign stack, and protecting from that is a separate property.

Then the analog to stack integrity is *stack noninterference*, which a system enjoys if for any execution $\overline{T} = T_1 T_2 \overline{T}' T_3 T_4$ such that every state from T_2 to T_3 shares the same stack and neither T_1 nor T_4 do, for any T_2', if $T_2 \sim_F T_2'$ there exists some T_3' such that $T_3 \sim_F T_3'$, $T_2' \xrightarrow{\overline{w}} T_3'$, and $\overline{w} = \mathit{behav}(T_2 \overline{T}' T_3)$.

Local noninterference implies both frame and stack noninterference. Naturally lazy enforcement does not enjoy local noninterference, but what about frame and stack noninterference? Stack noninterference would be violated by a coroutine that wrote outside the active stack, yielded, and then copied the value from that location into its active stack. But if no other coroutine tried to overwrite that location, it would not violate the policy. So stack noninterference would still be too strong. Frame noninterference does not have this issue, provided that the lazy policy distinguishes between distinct activations of the same frame depth. This is our final policy in the micropolicy section, below. We are still developing a property that works for coroutines.

Observable Frame Noninterference Suppose a function reads a value from outside of its accessible memory into a register, then immediately overwrites the register without influencing observations. This is benign, but our frame and stack noninterference properties do not allow it. They take a shortcut by checking the entire accessible state at the end of the activation. Our final policy enforces the frame noninterference property, but in principle we would like to present a looser property that captures the intuition that hidden information should not interfere with observable behavior.

Here we attempt to define a looser noninterference property. For simplicity now let us assume a system with only one stack, and we will deal only with calls and returns. A system enjoys *observable frame noninterference* if, for all $\overline{T} = \overline{T}_0 (M_1, \sigma_1, S)(M_2, \sigma_2, S) \ldots (M_3, \sigma_3, S) \overline{T}_4$ such that M_1, σ_1, S and M_3, σ_3, S bracket a call, the following holds for any $T_2' = M_2', \sigma_2, S$ such that $M_2, \sigma_2, S \sim_L T_2'$.

- There is some $T_3' = M_3', \sigma_3, S$ such that $T_2' \xrightarrow{\overline{w}}{}^{*} T_3'$
- T_3' has the same frame pointer as T_1
- Let $T_3'' = M_3'', \sigma_3, S$ where

$$M_3''(K) = \begin{cases} M_1(K) & \text{if } M_2(K) \neq M_2'(K) \\ M_3'(K) & \text{else} \end{cases}$$

- $\mathit{behav}(\overline{T}) = \mathit{behav}(\overline{T}_0 \mathbin{+\!\!+} \overline{w} \mathbin{+\!\!+} \mathit{behav}(T_3''\overline{T}_4')$ where $T_3''\overline{T}_4'$ is an execution.

So, T_3'' preserves those values from T_3' that were not changed in the variation. Note that this is a rollback as seen in $\xrightarrow{\text{Safe}}$, so this property subsumes observable integrity. (We could narrow it to handle only confidentiality by not rolling back values that have been written to after being varied.)

Safe Initialization Not strictly a type of confidentiality, *safe initialization* – the requirement that data be written before it is read – is nevertheless important. In particular, frame noninterference avoids the flaw of stack noninterference by virtue of subroutine activations that share a stack frame not being treated as the same entity. Nevertheless, an activation using an overlapping frame to a prior activation could well be influenced by it. For that matter a caller could influence its callee improperly by writing to the callee's accessible memory, if the callee reads uninitialized memory. So we give a safe initialization property.

For an execution $\overline{T} = \ldots(M_1, \sigma_1, S_1)(M_2, \sigma_2, S_2)\overline{T}_2$, let M_1' be a variant of M_1 defined as follows. For all K, if $\mathit{contour}(\sigma_1, S_1)(K) = \texttt{AL}$ and $\mathit{contour}(\sigma_2, S_2) = \texttt{A}$, $M_1'(K)$ may have any value; otherwise $M_1'(K) = M_1(K)$. In other words, M_1' varies from M_1 at only those components that have just been allocated. Then there is an execution $(M_1', \sigma_1, S_2)T_2'\overline{T}_2'$ such that $\mathit{behav}((M_2, \sigma_2, S_2)\overline{T}_2) = \mathit{behav}(T_2'\overline{T}_2')$.

6 Micropolicies

Here we describe some tag-based policies that enforce the integrity properties above. First we give the eager version, then the lazy version. In each case we modify the type of a memory configuration to map each component to a word and a tag of type $\mathbb{T}$.

$$t \in \mathbb{T} = \mathcal{S} \times \mathbb{N} + ()$$

$$M \in \mathcal{M} = \mathcal{K} \to \mathcal{W} \times \mathbb{T}$$

When $M(K) = w, t$ we say that t is the tag on K, or K is tagged with t. The tags on special registers are referred to by name: PC tag for the tag on `PC`, FP tag, etc. In our first two policies a tag is a pair representing the current stack and the depth of the active call within the stack, or else the default tag, (). We call setting a tag to () clearing it.

Eager Depth Tracking Initially, the PC tag is $S_{init}, 0$, where S_{init} is the initial stack. The value at the top of a stack S holding its coroutine's initial PC is tagged $S, 0$. On a function call, the return address (at the current `SP` and new `FP`) is tagged with the PC tag, S, d. The new PC tag becomes $S, d + 1$, and every address of the new stack frame is tagged $S, d + 1$. On return, the tag on the return address is restored to the PC tag, and the tags on the rest of the frame are cleared. Yields write the PC tag along with the PC, and restore the new ones from memory. When the PC tag is S, d, only addresses tagged S, d or () may be read or written.

This policy enforces local integrity, and therefore frame and stack integrity. It has a major weakness: it does not permit the dynamic manipulation of the current stack frame by altering the stack pointer. Doing so would leave addresses in the current frame tagged () and therefore accessible by higher stack frames. The most likely solution would be to introduce yet more pseudo-instructions, this time for tagging a range of memory.

On the confidentiality side, this enforces local noninterference, and therefore everything that follows from it. Because the entire new stack frame must have its tags updated on a call and cleared on return, if we use these writes to zero out the values in memory as well, then this policy also enforces temporal safety.

Lazy Depth Tracking The PC tag starts at $S_{init}, 0$ and the values holding coroutine PCs are initialized as above. The policy is identical except that stack frames are not tagged or cleared as a stack is pushed and popped. Rather, a write tags the written location with the current PC tag. A read is permitted if the location read is tagged () or exactly matches the PC tag.

This policy enforces nothing. Clearly it doesn't enforce local integrity. Consider observable integrity: suppose that a function A calls B, which writes to an unused portion of A's frame and returns. Then A calls another function B', and B' reads from that portion and outputs its value. B has successfully interfered with the behavior of the trace by writing to A's frame. Worse, suppose that before calling B', A deallocates some stack memory, moving its frame pointer below the tainted memory. Then a B' would not even need to violate confidentiality to output B's interference.

The same scenario, in which B' reads outside its accessible memory, violates all of our noninterference properties as well. These issues could occur whether or not $B = B'$. Roessler and DeHon add to the tag a function identifier to catch the case of different functions, but not the case in which they're the same.

This policy also does not enforce safe initialization. If data is left behind by B in its own frame, B' can read it if they are at the same depth.

Instance Tracking This variation does enforce observable integrity and frame noninterference. Tags are now a pair of a stack id and a natural number we will call the *instance*. Each time a function call is made, we generate a fresh instance i and set the new PC tag to S, i. The rest of the policy works the same as the Lazy Depth Tracking policy. Sadly, the requirement that we constantly generate fresh instances makes this variation costly, as the underlying tag mechanism relies on caching for good performance, and each new instance will cause compulsory misses.

Performance issues aside, the case above no longer violates. If B has the PC tag S, i then B' will have the tag $S, i + 1$, so B's external writes will not be accessible to B'.