Stack Safety as a Security Property

ANONYMOUS AUTHOR(S)

What is stack safety? Numerous mechanisms have been proposed to guard against attacks that corrupt the call stacks of running programs, but a clear and precise definition of stack safety is missing from the literature. We propose a formal definition phrased in the technical terminology of language-based security—as a combination of a confidentiality property (callees should not be able to read their caller's stack frames) and an integrity property (callees should not corrupt their caller's stack frames or control state), together with a straightforward well-bracketedness condition on the program's control flow. We discuss how these properties are enforced or approximated by existing stack protection mechanisms and describe useful stronger and weaker variants.

Additional Key Words and Phrases: Stack Safety, Micropolicies

1 INTRODUCTION

The call stack is a perennial target for attacks based on memory-safety vulnerabilities, with potential consequences ranging from leakage or corruption of private stack data to control-flow hijacking. To detect or prevent such attacks, a menagerie of software and hardware techniques for stack safety have been proposed, including stack canaries [Cowan et al. 1998], bounds checking [Devietti et al. 2008; Nagarakatte et al. 2009, 2010], split stacks [Kuznetsov et al. 2014], shadow stacks [Dang et al. 2015; Shanbhogue et al. 2019], capabilities [Chisnall et al. 2015; Skorstengaard et al. 2019a,b; Woodruff et al. 2014], and hardware tagging [Roessler and DeHon 2018].

But what *is* this stack safety of which people speak? Precise formal definitions are mostly lacking: rather, the notion is commonly defined in a negative way, by enumerating the bad things that a given enforcement mechanism prevents, such as corruption of return addresses, buffer overflows, or use of uninitialized variables. Indeed, we are only aware of one quite recent attempt to frame a positive definition, by Skorstengaard et al. [2019b], which defines stack safety as the conjunction of two properties: *local state encapsulation* and *well-bracketed control flow*. These properties are formalized by defining a capability-based machine that "satisfies [stack safety] by construction." While this work represents an important step in the right direction, we believe there is still room to craft a more abstract characterization of stack safety that is independent of capabilities or any other specific enforcement technique. In other words, we seek formal criteria that can be applied to a mechanism for protecting stack-based low-level programs to judge whether or not it actually "guarantees stack safety." Our key contribution is an abstract presentation of local state encapsulation using the technical framework of language-based security, together with a simple characterization of well-bracketed control flow.

The intuition behind Skorstengaard et al.'s *local state encapsulation* is that callers should be protected from callees: a callee should not be able to access or overwrite private data in the caller's stack frame. We formalize this intuition in (1) a *stack-confidentiality* property, which promises that a callee's observable behavior is not affected by memory outside its stack frame (e.g., a caller's private data or uninitialized memory), plus (2) a *stack-integrity* property, which promises that, if a function call ever returns, the caller's stack data at the point of return will be identical to what it was before the call.

The intuition behind Skorstengaard et al.'s *well-bracketed control flow* is the need to protect the call graph itself: a callee should pass control back to the caller only by transferring control to the instruction following the call; it should not be allowed to jump back into the caller's code at an arbitrary point, either directly (via a jump instruction) or indirectly (by corrupting stack data and

2021. 2475-1421/2021/1-ART1 $15.00
https://doi.org/

returning). Well-bracketed control flow establishes safety between functions: each instruction is uniquely owned by some function, and all jumps between functions are either calls that enter at a marked entry point or returns that re-enter just after their associated call in the standard nested call structure.

To phrase local state encapsulation as a security property requires an explicit attacker model: a clear understanding of who is being protected and from what. Most existing stack protection mechanisms envisage scenarios where a callee is influenced by external inputs to corrupt the stack in some way. To avoid getting into the details of exactly how this happens when formalizing our properties, we use an even stronger model: when we state the protections that a function can rely on when it makes a call, we assume that the callee can execute arbitrary instructions until it returns.

A key challenge in adapting the concepts of language-based security to this setting is that integrity and confidentiality are "nested" properties: not only does a given caller need to be protected from its callee, but the callee also needs to be protected from *its* callees, and so on. That is, the notion of "attacker" changes at many points throughout the execution of the program. We address this issue by introducing a dynamically changing notion of *accessibility* for the different components of the machine state and adjusting the integrity and confidentiality levels of different stack locations and registers at every transition (call or return) between functions.

After formally defining stack safety, we show that the definition is enforceable using an existing tag-based enforcement mechanism [Roessler and DeHon 2018]. We first consider Roessler and DeHon's conservative Depth Isolation policy, which not only enforces our basic stack-safety property, but also an alternative lockstep property that checks for integrity and confidentiality violations at every step of an execution trace. The advantage of this property is that it "fails faster" and is therefore more suitable for efficient testing. We prove (in Coq) that this "lockstep" property implies our original definition of stack safety.

The downside of the Depth Isolation policy is that it is rather expensive to implement. In the same work, Roessler and DeHon propose a different mechanism that is more efficient by virtue of being lazy in its enforcement. Instead of enforcing the integrity and confidentiality of the stack at the exact moment when a callee returns, this policy signals violations when the caller actually accesses data that the callee has written into its stack frame. The lazy policy does not enforce our original integrity and confidentiality properties, and to our knowledge there is no formal statement of the properties that it ought to enforce. We propose "observable" versions of integrity and confidentiality property, which capture the desired behavior of a policy that detects violations between when they occur and when they make a visible impact. Unfortunately Roessler and DeHon's existing "Lazy Tagging and Clearing" policy does not quite enforce these properties, but would if enhanced (at some performance cost).

In summary, we offer the following contributions:

- A novel *stack-safety property* abstractly capturing the intuitive concepts of strict local state encapsulation and well-bracketed control flow (Section 5). This property can be enforced by the Depth Isolation policy of Roessler and DeHon [2018], which is based on hardware tagging (Section 6.1).
- A *lockstep* variant of the stack-safety property that is also enforced by the Depth Isolation policy but is more efficient to test, plus a proof (in Coq) that this variant is stronger than the original (Section 6.2).
- A *lazy* variant of the stack-safety property that can be enforced if the Lazy Tagging and Clearing policy of Roessler and DeHon [2018] is enhanced with unique tags on each activation, plus a proof (in Coq) that this variant is weaker than the original (Section 7).

Section 8 discusses related work; Section 9 sketches directions for future work.

We begin with a description of our threat model (Section 2), a running example that we refer to throughout the paper (Section 3), and some technical preliminaries (Section 4).

2 THREAT MODEL AND ASSUMPTIONS

Stack safety is meant to protect a caller from a confused or malicious callee's attempts to access private data or hijack its control flow. Our attacker model reflects this intention by assuming that the attacker can run arbitrary machine code to try to access or overwrite its caller's stack data or disrupt its control state. Hardware and timing attacks are out of scope.

We assume we are given a machine language program, together with some auxiliary information about its structure: the code footprint of each function and the locations of instructions representing calls, returns, and function entry points. Although this information would typically come from a compiler together with the machine code, we do not assume anything about their provenance. We assume that the machine has a program counter PC and stack pointer SP; other details of the machine's ISA are unimportant.

To make our stack safety definitions precise, we make some straightforward assumptions about memory layout; our definitions could easily be changed to be parametric over these assumptions, at the cost of some complexity in their statements. Memory consists of an instruction region and a stack that grows upwards. (For simplicity, we do not consider the heap. Integrating a memory-safety property for the heap in the style of de Amorim et al. [2018] is a natural extension that we discuss in Section 8.) The stack has a conventional structure, with contiguously allocated per-function frames. The boundary between the frames of a caller and callee is indicated by the SP register at the time of the call. The stack can be used to pass arguments or return results in the usual way, by placing them at the top of the caller's frame. Callers and callees can also communicate via registers if they wish; indeed, our definitions say nothing at all about registers (except PC and SP).

Our stack-safety concepts depend critically on the notions of "call" and "return," which are not fully explicit in machine code (the same instruction opcode may be used in some places as part of a call or return sequence and in other places for other purposes). We define a call to be any instruction declared as such in the compiler-provided call structure. If a call occurs at PC $= a$ and SP $= s$, then the *corresponding return* has occurred when the PC points to the next instruction after a and SP $= s$. (Our definition of well-bracketed control flow relates this definition of return to the program's declared set of return points.) Our property definitions make no other detailed assumptions about call or return sequences, although some of the enforcement mechanisms we consider do.

Enforcement mechanisms are modeled as *policies* (borrowing the terminology from tag-based mechanisms) that add some auxiliary policy state to the ordinary machine state and a partial policy step function that refines the machine step function by allowing or disallowing a step to be taken based on its auxiliary state. Enforcement works by halting the program before it can perform an action that would violate the security property; our properties are therefore naturally *termination insensitive*. This model will be made more precise in Section 4.

We model ordinary program state and policy-enforcement state separately to make it easier to use well-established concepts from language-based security. In particular, confidentiality has an elegant extensional characterization in terms of *noninterference*: a program preserves the confidentiality of certain data if *varying* that data would not change the program's observable behavior [Goguen and Meseguer 1982]. In applying this idea to a system that incorporates policy enforcement, only ordinary program data should be varied, while any associated policy state should be left alone; our model makes this distinction obvious. We will also introduce a notion of *observational integrity* that relies on the idea of *rolling back* program state; again, our model makes it easy to describe rollback of ordinary program data while leaving policy state alone.

This enforcement model directly describes micropolicies [Azevedo de Amorim et al. 2015], and we believe that it is flexible enough to capture code-altering approaches (such as code rewriting or capability-based techniques).

Micropolicies. Micropolicies are a flexible tag-based, hardware-accelerated reference monitoring mechanism; they been applied to stack safety enforcement by Roessler and DeHon [2018]. Here the policy state consists of metadata tags (e.g., identifying stack frames) attached to each value in memory or registers, and the policy step function checks that each machine operation obeys a set of rules on tags (e.g., that the current PC tag matches the tag on the stack location being accessed), and halts the machine if not. Distinguishing ordinary machine state and policy state is very natural in this setting.

Capability machines. Capability machines such as CHERI [Woodruff et al. 2014] extend conventional architectures to support efficient and fine-grained control over memory accesses, which can be used to implement security policies, including the stack protection policies of Skorstengaard et al. [2019a] and (with an extension to support linear capabilities) Skorstengaard et al. [2019b]. Capabilities package up pointers (ordinary data) with base and bounds information (policy data), and they require rewriting source code (e.g., to use capabilities in place of ordinary pointers), so separating ordinary machine state from policy state is delicate; we conjecture it can be done with some effort.

Software-only policies. Some policies are implemented in software, by modifying code in the compiler, for example to perform bounds checking [Nagarakatte et al. 2009] or to insert stack canaries [Cowan et al. 1998]. Such approaches can, in principle, be evaluated in terms of our formal notion of stack safety with a trivial null policy. But any state maintained by the enforcement mechanism must be in normal memory and subject to the same confidentiality and integrity concerns (to be varied, rolled back, etc.), a heavy constraint. Alternately it can be stripped out as policy state provided that it can be proven to protect itself.

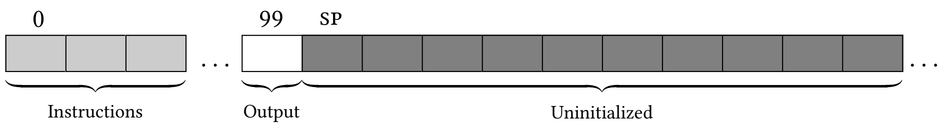
3 RUNNING EXAMPLE

Figure 1 shows a high-level view of a running example, written in C-like pseudo-code. It consists of a `main` function that performs a call to some function `f`, which in turn makes a nested call to some function `g` before returning, while performing some simple arithmetic operations in the process. Figure 2 shows an execution trace of the corresponding machine instructions. These might have been generated from the high-level code by a (rather simple-minded) compiler, but it is only the behavior of the machine instructions that we care about here—Figure 2 is our actual example.

To streamline the example, we use a small instruction set reminiscent of RISC-V and make a few simplifying assumptions about the organization of memory. First, as noted above, since our goal is to formalize stack safety, we ignore the heap and assume that all variables are stack allocated. We also assume that the machine is word- (not byte-) addressed, that program instructions start

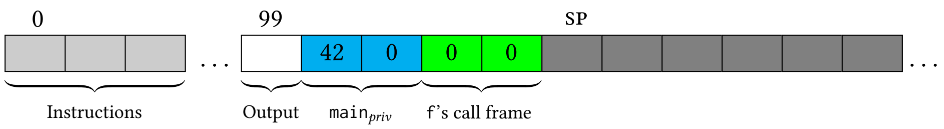
```
main = {                  f(z) = {                 g(v) = {
  int x = 42, y = 0;        int w = g(17);           return (v + 1);
  y = f(0);                 print z;               }
  print (x + y);            return (z + w);
}                         }
```

Fig. 1. A High-level Program With Nested Calls



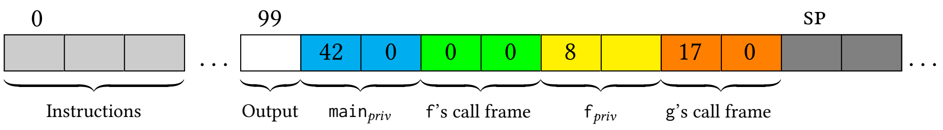
PC $= 0$ $\quad r_0 = 0$ $\quad$ RA $=?$ $\quad$ SP $= 100$ $\quad r_4 =?$ $\quad r_5 =?$

PC	*Instruction*	*Effect*	
0	SP $\leftarrow$ SP $+ 2$	SP $\leftarrow 102$	Allocate private data for `main`
1	$r_4 \leftarrow r_0 + 42$	$r_4 \leftarrow 42$	Load 42 into r_4
2	[SP $- 2$] $\leftarrow r_4$	$[100] \leftarrow 42$	Set `x` to 42
3	[SP $- 1$] $\leftarrow r_0$	$[101] \leftarrow 0$	Set `y` to 0
4	SP $\leftarrow$ SP $+ 2$	SP $\leftarrow 104$	Allocate `f`'s call frame
5	[SP $- 2$] $\leftarrow r_0$	$[102] \leftarrow 0$	Set argument to 0
6	[SP $- 1$] $\leftarrow r_0$	$[103] \leftarrow 0$	Zero out result
7	RA $\leftarrow$ PC $+ 1$; PC $\leftarrow 20$	RA $\leftarrow 8$; PC $\leftarrow 20$	Call (JAL to) `f`



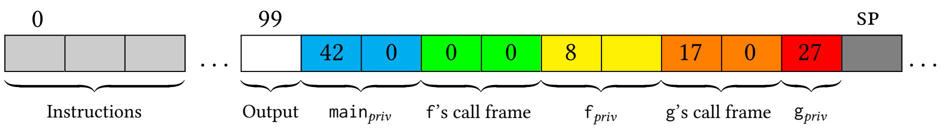
PC $= 20$ $\quad r_0 = 0$ $\quad$ RA $= 8$ $\quad$ SP $= 104$ $\quad r_4 = 42$ $\quad r_5 =?$

20	SP $\leftarrow$ SP $+ 2$	SP $\leftarrow 106$	Allocate stack space for `w`
21	[SP $- 2$] $\leftarrow$ RA	$[104] \leftarrow 8$	Save return address
22	SP $\leftarrow$ SP $+ 2$	SP $\leftarrow 108$	Allocate `g`'s call frame
23	$r_4 \leftarrow r_0 + 17$	$r_4 \leftarrow 17$	Store 17 into r_4
24	[SP $- 2$] $\leftarrow r_4$	$[106] \leftarrow 17$	Store 17 as the argument `v`
25	[SP $- 1$] $\leftarrow r_0$	$[107] \leftarrow 0$	Zero out result
26	RA $\leftarrow$ PC $+ 1$; PC $\leftarrow 40$	RA $\leftarrow 27$; PC $\leftarrow 40$	Call `g`



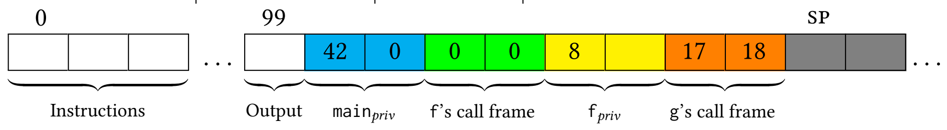
PC $= 40$ $\quad r_0 = 0$ $\quad$ RA $= 27$ $\quad$ SP $= 108$ $\quad r_4 = 17$ $\quad r_5 =?$

40	SP $\leftarrow$ SP $+ 1$	SP $\leftarrow 109$	Allocate space for return address
41	[SP $- 1$] $\leftarrow$ RA	$[108] \leftarrow 27$	Save return address
42	$r_4 \leftarrow$ [SP $- 3$]	$r_4 \leftarrow 17$	Load the argument (`v`) into r_4
43	$r_4 \leftarrow r_4 + 1$	$r_4 \leftarrow 18$	Increment r_4



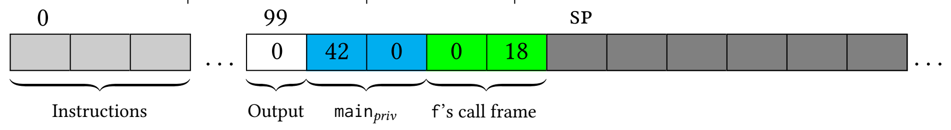
PC $= 40$ $\quad r_0 = 0$ $\quad$ RA $= 44$ $\quad$ SP $= 109$ $\quad r_4 = 18$ $\quad r_5 =?$

44	$[\textsc{sp} - 2] \leftarrow r_4$	$[107] \leftarrow 18$	Store r_4 as the return value
45	$\textsc{ra} \leftarrow [\textsc{sp} - 1]$	$\textsc{ra} \leftarrow 27$	Load return address
46	$\textsc{sp} \leftarrow \textsc{sp} - 1$	$\textsc{sp} \leftarrow 108$	Deallocate g's local state
47	$\textsc{pc} \leftarrow \textsc{ra}$	$\textsc{pc} \leftarrow 27$	Return to f (JALR)



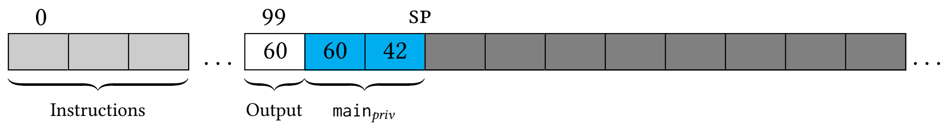
$\textsc{pc} = 27 \quad r_0 = 0 \quad \textsc{ra} = 48 \quad \textsc{sp} = 108 \quad r_4 = 18 \quad r_5 = ?$

27	$r_4 \leftarrow [\textsc{sp}]$	$r_4 \leftarrow 18$	Load return value into r_4
28	$[\textsc{sp} - 3] \leftarrow r_4$	$[105] \leftarrow 18$	Set w to return value
29	$\textsc{sp} \leftarrow \textsc{sp} - 2$	$\textsc{sp} \leftarrow 106$	Deallocate g's call frame
30	$r_5 \leftarrow [\textsc{sp} - 4]$	$r_5 \leftarrow 0$	Load argument z to r_5
31	$[\textsc{out}] \leftarrow r_5$	$[99] \leftarrow 0$	Print z
32	$r_4 \leftarrow r_4 + r_5$	$r_4 \leftarrow 18$	Add r_4 and r_5
33	$[\textsc{sp} - 3] \leftarrow r_4$	$[103] \leftarrow 18$	Store r_4 as the return value
34	$\textsc{ra} \leftarrow [\textsc{sp} - 2]$	$\textsc{ra} \leftarrow 8$	Load return address
35	$\textsc{sp} \leftarrow \textsc{sp} - 2$	$\textsc{sp} \leftarrow 104$	Deallocate f's local state
36	$\textsc{pc} \leftarrow \textsc{ra}$	$\textsc{pc} \leftarrow 8$	Return to main (JALR)



$\textsc{pc} = 8 \quad r_0 = 0 \quad \textsc{ra} = 37 \quad \textsc{sp} = 104 \quad r_4 = 18 \quad r_5 = 0$

8	$r_4 \leftarrow [\textsc{sp} - 1]$	$r_4 \leftarrow 18$	Load return value to r_4
9	$[\textsc{sp} - 3] \leftarrow r_4$	$[101] \leftarrow 18$	Set y to the result
10	$\textsc{sp} \leftarrow \textsc{sp} - 2$	$\textsc{sp} \leftarrow 102$	Deallocate f's call frame
11	$r_5 \leftarrow [\textsc{sp} - 2]$	$r_5 \leftarrow 42$	Load x to r_5
12	$r_4 \leftarrow r_4 + r_5$	$r_4 \leftarrow 60$	Add x and y
13	$[\textsc{out}] \leftarrow r_4$	$[99] \leftarrow 60$	Print x + y



$\textsc{pc} = 14 \quad r_0 = 0 \quad \textsc{ra} = 37 \quad \textsc{sp} = 102 \quad r_4 = 60 \quad r_5 = 42$

Fig. 2. Detailed Execution Trace

from location 0, that the machine communicates with the outside world through a single memory-mapped output port at address 99 (so a `print` statement will be compiled to a store to this location), and that the stack begins at location 100 and grows upwards.

A sketch of the initial memory layout appears at the top of Figure 2: the instructions for `main` begin at 0, and the (uninitialized) stack begins at 100. Directly below the memory sketch, we summarize the parts of the register file that are relevant at this point in the example. The rest of

Figure 2 traces the execution of the program, showing the machine state at call and return points. Instructions are intended to be similar to RISC-V, but are described in a (hopefully more readable) ad-hoc notation rather than using assembler syntax. The JAL (jump-and-link) instruction used for calls atomically simultaneously the return address in a register and jumps to a specified new PC; the JALR (jump-and-link-via-register) instruction used for returns jumps to the address stored in a register. Register r_0 is hardwired to zero, so that, e.g., instruction 3 stores 0 into address 101.

The code itself is straightforward, and should be largely self-explanatory, but a few points deserve clarification. Functions return their result value on the stack rather than in a register. Stack frames are built in two parts: space for local variables (which are all stack-allocated) and the return address is allocated and initialized at function entry, and deallocated at function exit; space for arguments and return values is allocated and initialized immediately prior to each function call and deallocated on return. The return value slot is always initialized because some of our enforcement mechanisms will rely on this. The return address is spilled to the stack even by leaf functions such as g that could just leave it in RA.

4 MACHINES, TRACES, AND OBSERVATIONS

We next describe our machine model. To make our definition of stack safety as generic as possible, this section proposes an abstract interface to a machine model plus some kind of (software- or hardware-enforced) policy monitor. In Section 6, we will instantiate this model with a concrete dynamic policy enforcement mechanism based on Roessler and DeHon [2018].

4.1 Values, States, and Observations

The basic building blocks of the machine are *values* and *addresses*. Both are drawn from some set of *words* $\mathcal{W}$, ranged over by w. Our machine states are composed of *components* k, addressed either by words (memory addresses) or by register names (r) drawn from some set $\mathcal{R}$.

$$k \in \mathcal{K} = \mathcal{W} + \mathcal{R}$$

The register names are assumed to include two special purpose registers: the program counter PC and the stack pointer SP.

A *machine state* is a map from components to values:

$$m \in \mathcal{M} = \mathcal{K} \rightarrow \mathcal{W}$$

The step function of the machine takes a machine state and produces a pair of a machine state and an *observation*, which may be either silent (written τ) or a word of datathat the machine communicates to the outside world:

$$o \in Obs = \{\tau\} \uplus \mathcal{W}$$

$$m \xrightarrow{o} m' \in \mathcal{M} \rightarrow \mathcal{M} \times Obs$$

This step function is total, so the machine's execution traces (defined in Section 4.3) are infinite.

We choose to model observations explicitly, rather than simply treating a subset of registers or memory as being observable. This choice gives us the flexibility to treat programs as observationally equivalent even when their internal states may differ.

In the running example of Section 3, $\mathcal{W}$ is the set of 64-bit integers and $\mathcal{R}$ is the set $\{r_0, \cdots, r_{31}\} \cup \{\text{PC}, \text{SP}\}$. Observations are writes to the special OUT location, which can be thought of as a memory-mapped output port: if during an execution step a word w is written to OUT then $o = w$, otherwise $o = \tau$.

4.2 Policies

We now present our model of enforcement, which we introduced informally in Section 2. A policy is a general model of an enforcement mechanism, consisting of a set of policy states $p \in \mathcal{P}$, a policy step function $(m, p) \rightharpoonup p' \in \mathcal{M} \times \mathcal{P} \rightharpoonup \mathcal{P}$. This step function is partial; it is undefined on input configurations that correspond to a policy fault. We assume that the compiler provides an initial policy state p_0 to accompany the initial machine state. A concrete policy based on Roessler and DeHon [2018] will be described in Section 6.

We write *MPState* for the set of all pairs of machine states and policy states.

$$mp \in \mathit{MPState} = \mathcal{M} \times \mathcal{P}$$

We lift the policy step function to operate on such pairs by combining it with the regular step function for machine states.

$$\frac{m_1 \xrightarrow{o} m_2 \quad (m_1, p_1) \rightharpoonup p_2}{(m_1, p_1) \overset{o}{\rightharpoonup} (m_2, p_2)}$$

This combined step function is also partial, and traces of the machine-with-policy will be finite iff a policy fault occurs.

4.3 Traces

A trace over a type A is a nonempty, finite or infinite sequence of As; we write *Traces* (A) for the set of traces over A. We will mostly be concerned with sequences of pairs of machine states and observations, called *machine traces* and denoted $M \in \mathit{Traces}\,(\mathcal{M} \times \mathit{Obs})$, and with sequences of triples of a machine state, a policy state, and an observation, called *MP-traces* and denoted $MP \in \mathit{Traces}\,(\mathcal{M} \times \mathcal{P} \times \mathit{Obs})$.

Trace-Of. The "trace-of" operator, written $m \hookrightarrow M$, coinductively relates a machine state with the trace of machine states and observations produced by repeated application of step:

$$\frac{m_0 \xrightarrow{o} m_1 \quad m_1 \hookrightarrow M}{m_0 \hookrightarrow (m_0, o)M}$$

(We use juxtaposition of an element and a sequence to represent "cons.") Since the step function is total, if $m \hookrightarrow M$, then M must be infinite.

In our running example in Figure 2 we can see a depiction of (a prefix of) such a trace: starting from an initial machine state at the very top, each instruction leads to a new machine state (the diff of which, compared to the previous one, appears in the "*Effect*" column), and the majority of observations are τ, except for instructions 31 and 13, which are writes to OUT where the observations are 0 and 60, respectively.

In a similar fashion, we define $mp \hookrightarrow MP$ to relate a machine-policy state pair to the MP-trace induced by the step function on *MPState*. Unlike the *Traces* $(\mathcal{M} \times \mathit{Obs})$ above, this trace can be finite as the policy can cause a fault, in which case the final observation is a τ:

$$\frac{\neg \exists p_1 . (m_0, p_0) \rightharpoonup p_1}{(m_0, p_0) \hookrightarrow (m_0, p_0, \tau)} \qquad \frac{(m_0, p_o) \overset{o}{\rightharpoonup} (m_1, p_1) \quad (m_1, p_1) \hookrightarrow MP}{(m_0, p_0) \hookrightarrow (m_0, p_0, o)MP}$$

We will also use π_m to project the trace of machine states out of a trace, π_p to project the trace of policy states, and π_o to project the observations, with π_m and π_o overloaded for machine traces and MP-traces.

Head and Last. We take the first element of a trace with $\mathit{head}(T)$, which is a total function since traces are non-empty, and the final element (if one exists) with $\mathit{last}(T)$, which is partial.

Until. The operation $Until\,(f,T)$ takes a trace $T \in Traces\,(A) \times Obs$ and a predicate on elements $f \subset A$ and gives the prefix of T ending with the first element on which f holds. One subtlety is that, because the observation associated with some element in a trace represents what can be observed when this element is executed (i.e., the observation on that step), when we take a prefix from a trace containing observations we must replace the final observation with a τ.

$$\frac{f\ a}{Until\,(f,(a,o)T) = (a,\tau)} \qquad \frac{}{Until\,(f,(a,o)) = (a,\tau)} \qquad \frac{\neg f\ a \quad Until\,(f,T) = T'}{Until\,(f,(a,o)T) = (a,o)T'}$$

If the resulting trace is a strict prefix of the parameter, then the predicate must hold on its final element. If the predicate never holds, then *Until* is the identity function.

We will frequently take a prefix of the trace from an initial state up through the first state where some condition holds on the machine state. This can easily be implemented using *Until*, and we provide special notation for convenience. If f be a predicate on machine states, we define $m \hookrightarrow M|f$ (read "M is the prefix of running m up to f") and similarly for traces with policies:

$$\frac{m \hookrightarrow M' \quad Until\,(f,M') = M}{m \hookrightarrow M|f} \qquad \frac{mp \hookrightarrow MP' \quad Until\,(\lambda(m,p).f\ m, MP') = MP}{mp \hookrightarrow MP|f}$$

Observational Similarity. We say that a trace of observations O_1 is a prefix of O_2 as far as an external observer is concerned, written $O_1 \lesssim O_2$, if the sequence of non-silent observations of O_1 is a prefix of those of O_2; that is, we operate up to deletion of τ observations, coinductively:

$$\frac{}{O \lesssim O} \qquad \frac{}{\tau \lesssim O} \qquad \frac{}{w \lesssim wO}$$

$$\frac{O_1 \lesssim O_2}{\tau O_1 \lesssim O_2} \qquad \frac{O_1 \lesssim O_2}{O_1 \lesssim \tau O_2} \qquad \frac{O_1 \lesssim O_2}{wO_1 \lesssim wO_2}$$

We then define similarity of observation traces as traces prefixing each other:

$$O_1 \simeq O_2 \triangleq O_1 \lesssim O_2 \wedge O_2 \lesssim O_1$$

Note that an infinite silent trace is a prefix of (and similar to) any other trace. While this might seem surprising at first, it makes sense in a timing-insensitive context: an external observer looking at two machine runs cannot (computably) distinguish between a machine that steps forever and a machine that steps for a long time before producing some output.

5 STACK SAFETY, FORMALLY

We are finally ready for our definition of stack safety. We begin by describing how we model knowledge about the program structure, then dive into the formal definitions of local state encapsulation and well-bracketed control flow, which together constitute stack safety.

5.1 Program Structure

Our properties and policies assume that the program is annotated with various information about its structure.

Function Map. A *function map* identifies some addresses as code and assigns each to a function. Formally, a function map is a partial function from addresses to a function identifier drawn from the set $\mathcal{F}$.

$$fm \in FuncMap = \mathcal{W} \rightharpoonup \mathcal{F}$$

In our running example, instructions 1-13 correspond to `main`, instructions 20-36 correspond to `f`, and instructions 40-47 correspond to g, which means that the function map would be:

$$fm(w) = \begin{cases} \texttt{main} & \text{if } 1 \leq w \leq 13 \\ \texttt{f} & \text{if } 20 \leq w \leq 36 \\ \texttt{g} & \text{if } 40 \leq w \leq 47 \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

Call Map. A *call map* identifies addresses that complete a call sequence, abstracting away from any concrete calling convention. Formally, it is a partial function from addresses to a natural number, the number of arguments and return values of the call.

$$cm \in \mathit{CallMap} = \mathcal{W} \rightharpoonup \mathbb{N}$$

In our running example, the callmap contains two entries: the word 7 corresponding to the call to `f` which has one argument and one return value; and the word 26 corresponding to the call to g, also with a single argument and return value:

$$cm = \{7 \mapsto 2, 26 \mapsto 2\}$$

For concreteness, in a typical calling convention, the call map will identify the locations of the JAL instructions representing calls. A state in which a call map is defined on the program counter is referred to as a *call state.*

Return Map. Similarly, a *return map* identifies addresses that complete a return sequence. Formally, it is a set of words; each one is the address of the final jump corresponding to a function return:

$$rm \in \mathit{RetMap} \subseteq \mathcal{W}$$

In our running example, the return map identifies the JALR instructions 36 and 47:

$$rm = \{36, 47\}$$

Entry Map. Finally, the entry map identifies valid entry points to a function and is also a set:

$$em \in \mathit{EntryMap} \subseteq \mathcal{W}$$

In our running example, the entry map contains the entry points of `f` and g:

$$em = \{20, 40\}$$

What is a return? We separately identify, for any call state, what it means to have correctly returned from that call. We consider the call to have returned when control returns to the instruction following the JAL with the stack restored. Formally, this is a predicate on states that says when the machine has just returned from a specified call state (where *next* gives the address of the next instruction after a given address):

$$\mathit{justret}\ m_c \triangleq \lambda m_r . m_r(\textsc{pc}) = \mathit{next}(m_c(\textsc{pc})) \wedge m_r(\textsc{sp}) = m_c(\textsc{sp})$$

There are two such return points in our running example, both of which are depicted in the figure: the state before the execution of instruction 27 corresponding to the return from `f` and the state before instruction 8 corresponding to the return from g.

In the rest of this section we implicitly parameterize all definitions by a function map *fm*, a call map *cm*, a return map *rm* and an entry map *em*.

5.2 Local State Encapsulation

The first part of stack safety is *local state encapsulation*. Informally, it captures the notion that a caller's data is protected from a callee's actions. We formalize this notion as a security property with an integrity and a confidentiality component, where the former protects the caller's data from being written and the latter from being read.

As is standard with security properties, we associate with each component of the machine a *label*: a pair of security levels denoting high or low confidentiality and high or low integrity.

$$\mathit{label} ::= \{\mathrm{HC}, \mathrm{LC}\} \times \{\mathrm{HI}, \mathrm{LI}\}$$

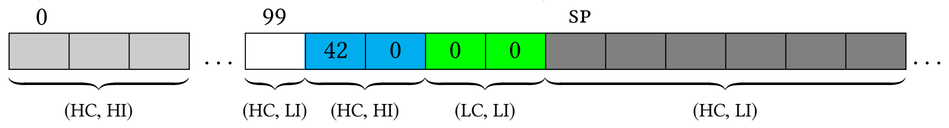
In a traditional setting, integrity would enforce that the contents of HI locations remain unchanged, while confidentiality would enforce that the contents of HC locations do not influence the observable behavior of the machine. However, when dealing with nested calls and returns, stack safety needs to enforce integrity and confidentiality for *every* function call in a trace; that is, we must treat the security level of data dynamically. To that end, we introduce the notion of a *contour*, a map from components to labels:

$$C \in \mathcal{C} ::= \mathcal{K} \rightarrow \mathit{label}$$

When m_c is a call state ($cm(m_c(\textsc{pc})) = n$ for some n) and $m_c \xrightarrow{o} m_e$ for some m_e and o, we can construct a contour based on the bounds of the callee's stack frame. The private state of the callee is the entire area of the memory from stack base to stack pointer, with the exception of the n locations corresponding to the arguments and result locations of the call. Private components are marked high confidentiality and high integrity: the callee should be able to neither read nor write them. Everything above the stack pointer at the point of the call is "uninitialized" memory that can be written into but should not be read. To protect the callee itself from such reads, these components are marked high confidentiality and low integrity. The same is true for the designated output location: it can be freely written into but should not be read. Code is always high integrity and low confidentiality. Finally, registers and the frame of arguments and results are free for the callee to use, and are therefore tagged with low integrity and low confidentiality.

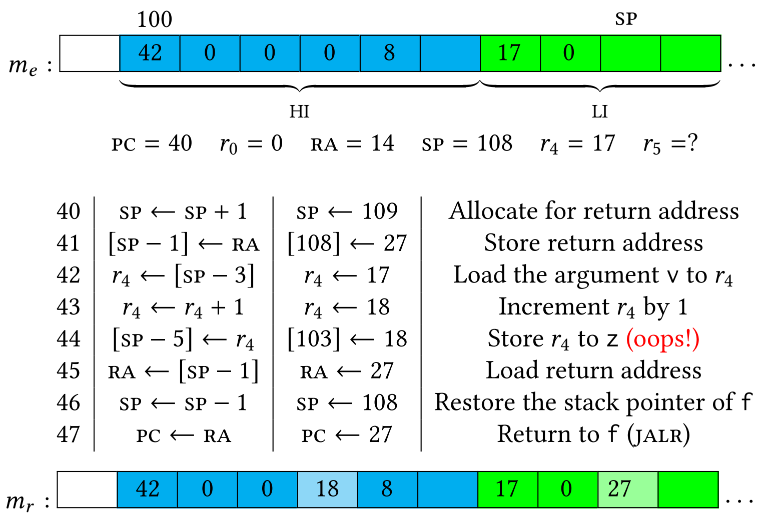
$$\mathit{Cof}(m,n)(k) = \begin{cases} (\mathrm{LC}, \mathrm{HI}) & \text{if } k \in \mathcal{W} \text{ and } fm(k) = f \text{ for some } f \\ (\mathrm{HC}, \mathrm{LI}) & \text{else if } k \in \mathcal{W} \text{ and } k \geq m(\textsc{sp}) \text{ or } k = \textsc{out} \\ (\mathrm{HC}, \mathrm{HI}) & \text{else if } k \in \mathcal{W} \text{ and } k \leq m(\textsc{sp})\text{-}n \\ (\mathrm{LC}, \mathrm{LI}) & \text{else} \end{cases}$$

Going back to our running example, here is the state of the machine at the call from `main` to `f`, showing addresses annotated with their corresponding labels:



5.2.1 Stack Integrity. The first component of local state encapsulation is an integrity property: it ensures that the caller's data is never overwritten by a callee. We will formally define integrity in two steps: first, we will define integrity as a trace property over the trace corresponding to a single call (from its entry point to its corresponding return), and then we will lift that to a property for every such subtrace of a callee.

Definition 5.1. Let C be a contour and MP be a (potentially infinite) MP-trace. say MP satisfies *trace integrity* with respect to C if its final state (if any) agrees with its initial one on all components



PC = 40 $\quad r_0 = 0 \quad$ RA = 14 $\quad$ SP = 108 $\quad r_4 = 17 \quad r_5 = ?$

40	SP ← SP + 1	SP ← 109	Allocate for return address
41	[SP − 1] ← RA	[108] ← 27	Store return address
42	r_4 ← [SP − 3]	r_4 ← 17	Load the argument v to r_4
43	r_4 ← r_4 + 1	r_4 ← 18	Increment r_4 by 1
44	[SP − 5] ← r_4	[103] ← 18	Store r_4 to z (oops!)
45	RA ← [SP − 1]	RA ← 27	Load return address
46	SP ← SP − 1	SP ← 108	Restore the stack pointer of f
47	PC ← RA	PC ← 27	Return to f (JALR)

Fig. 3. Failing Trace Integrity

marked HC in C:

$$\textsc{TraceInt}\ C\ MP \triangleq head(MP) = (m_e, \_, \_) \land last(MP) = (m_r, \_, \_) \Rightarrow$$
$$\forall k.\ C(k) = \mathrm{HC} \Rightarrow m_e(k) = m_r(k)$$

Intuitively, the caller expects to find their private data untouched after the callee returns. That means that if the callee never returns, integrity is trivially satisfied. Similarly, the callee is allowed to temporarily overwrite the caller's private locations, as long as their values are restored by the time the callee returns. We will see variations of this later on: a stronger, inductive integrity property that ensures that the caller's private data is never overwritten, even during the callee's execution (which leads to easier reasoning and more effective testing); and a weaker, lazy integrity property that only ensures the preservation of locations that subsequently affect the program's observable actions (which allows for a more efficient enforcement mechanism).

Trace integrity characterizes the trace corresponding to a single callee. Going back to our running example, such a trace would be, for instance, the entire trace of g from instruction 40 to instruction 47. As an example of an integrity violation affecting this trace, Fig. 3 shows what happens if we change line 44 of the example program to [SP − 5] ← r_4. Instead of storing data to the return value slot, the changed version overwrites f's local variable z. The figure shows the modified trace, with snapshots of the initial and final states colored to show the contour; high integrity memory is in cyan, while low integrity is in green. At the end of the trace high integrity address 103 should match the initial state, and does not. So trace integrity is violated.

We now lift this definition to an integrity property for a complete execution trace. We do this by requiring that trace integrity holds for every subtrace delimited by a call and its corresponding return.

Definition 5.2. Given a trace MP, an *n-transition* is a pair of machine-policy state pairs (m_c, p_c) and (m_e, p_e) in MP such that m_c is a valid call with n arguments, and (m_c, p_c) steps to (m_e, p_e).

$$Tr_n\ MP\ (m_c, p_c)\ (m_e, p_e) \triangleq \exists o.\ (m_c, p_c) \xrightarrow{o} (m_e, p_e)$$
$$\land\ cm(m_c(\textsc{pc})) = n$$

Definition 5.3. We say that a system enjoys *stack integrity* if for every initial state (m_0, p_0) and its corresponding induced trace MP (with $(m_0, p_0) \hookrightarrow MP$), and for every n-transition from (m_c, p_c) to (m_e, p_e) in MP, if we take the prefix of the call until its corresponding return $(m_e, p_e) \hookrightarrow MP_{call} \mid justret\ m_c$, then MP_{call} must satisfy trace integrity:

$$\textsc{TraceInt}\ Cof(m_c, n)\ MP_{call}$$

5.2.2 *Stack Confidentiality.* The second component of local state encapsulation is a confidentiality property: just like integrity ensures that a caller's data is never overwritten by a callee, confidentiality ensures that the caller's data never influences the callee's observable behavior. Again, we will formally define confidentiality in two steps: a trace version and a system version.

Unlike integrity, confidentiality is a noninterference-style property, comparing the behavior of a machine with an arbitrary variation that preserves low confidentiality data.

Definition 5.4. Two machine states m and m' are *variations* of one another given a contour C if they agree on low confidentiality data:

$$m \approx_C m' \triangleq \forall k. C(k) = \mathrm{LC} \Rightarrow m(k) = m'(k)$$

Then, the trace version of confidentiality operates on two traces: the "original" trace $MP \in Traces(\mathcal{M} \times \mathcal{P} \times Obs)$ that corresponds to a callee f and a "variant" trace $M \in Traces(\mathcal{M} \times Obs)$ that corresponds to how f would behave if it were started from a variant initial machine state. As an example of variations, consider the top of Figure 4 which shows a snapshot of the state as in the running example (top memory layout) and a variation of it (bottom). The "Effect" column shows differences between the two runs by writing values in the original run in cyan and values in the variant run in magenta separated by a slash. This example will end up with a violation of confidentiality, as a variation in a confidential location (42 vs −5 in location 100) leads to a visible difference in return value.

Definition 5.5. *Trace confidentiality* will be a property of two traces, MP and M, with respect to a return predicate $R \subset \mathcal{M}$ that reflects the states that are considered to have returned. We abstract over R because what it means to return depends on the state from which f was called. There are three cases in which MP and M enjoy trace confidentiality with respect to R.

- If MP has a final state $last(MP) = (m_r, \_, \_)$ and $R\ m_r$, then M is also finite with some final state m'_r and any changes from the initial states are matched between m_r and m'_r. In addition their observations are similar.

$$\begin{aligned} ReturnCase\ R\ MP\ M \triangleq\ & last(MP) = (m_r, \_, \_)\ \wedge\ R\ m_r \Rightarrow \\ & \exists m'_r. last(M) = (m'_r, \_) \wedge \pi_o(MP) \eqsim \pi_o(M) \wedge \\ & \forall\ k.(m_e(k) \neq m_r(k) \vee m'_e(k) \neq m'_r(k)) \Rightarrow m_r(k) = m'_r(k) \end{aligned}$$

- If MP is infinite, then M is also infinite and their observations are similar.

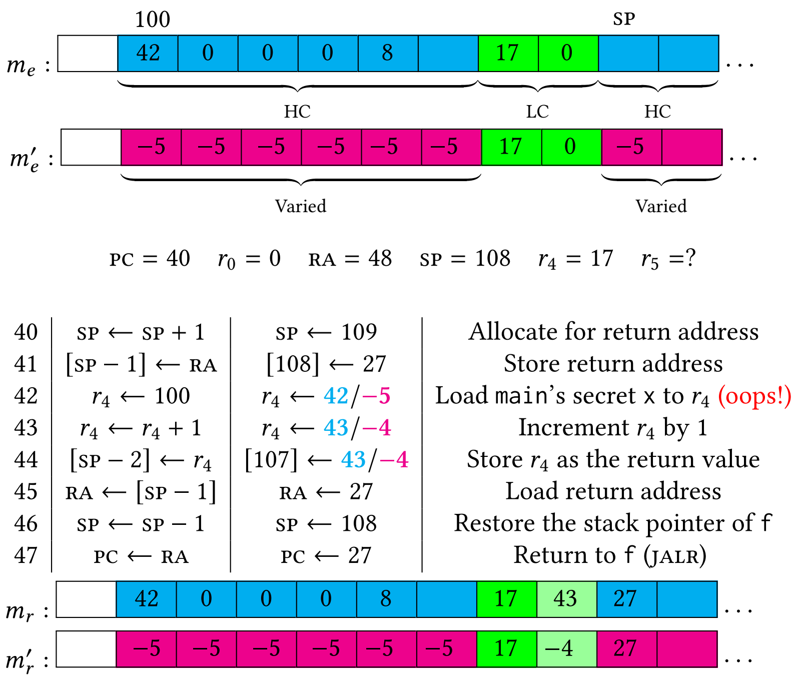
$$\begin{aligned} InfCase\ MP\ M \triangleq\ & \neg(\exists mpo. last(MP) = mpo) \Rightarrow \\ & \neg(\exists mo. last(M) = mo) \wedge \pi_o(MP) \eqsim \pi_o(M) \end{aligned}$$

- If MP ends prematurely without fulfilling R, then its observations are a prefix of those of M.

$$\begin{aligned} FailCase\ R\ MP\ M \triangleq\ & \forall m_r. last(MP) = (m_r, \_, \_)\ \wedge\ \neg\ R\ m_r \Rightarrow \\ & \pi_o(MP) \lesssim \pi_o(M) \end{aligned}$$

Putting it all together:

$$\textsc{TraceConf}\ R\ MP\ M \triangleq ReturnCase\ R\ MP\ M\ \wedge InfCase\ MP\ M \wedge FailCase\ R\ MP\ M$$



$\text{PC} = 40 \quad r_0 = 0 \quad \text{RA} = 48 \quad \text{SP} = 108 \quad r_4 = 17 \quad r_5 = ?$

40	$\text{SP} \leftarrow \text{SP} + 1$	$\text{SP} \leftarrow 109$	Allocate for return address
41	$[\text{SP} - 1] \leftarrow \text{RA}$	$[108] \leftarrow 27$	Store return address
42	$r_4 \leftarrow 100$	$r_4 \leftarrow 42/-5$	Load `main`'s secret `x` to r_4 (oops!)
43	$r_4 \leftarrow r_4 + 1$	$r_4 \leftarrow 43/-4$	Increment r_4 by 1
44	$[\text{SP} - 2] \leftarrow r_4$	$[107] \leftarrow 43/-4$	Store r_4 as the return value
45	$\text{RA} \leftarrow [\text{SP} - 1]$	$\text{RA} \leftarrow 27$	Load return address
46	$\text{SP} \leftarrow \text{SP} - 1$	$\text{SP} \leftarrow 108$	Restore the stack pointer of `f`
47	$\text{PC} \leftarrow \text{RA}$	$\text{PC} \leftarrow 27$	Return to `f` (JALR)

Fig. 4. Failing Trace Confidentiality

In the non-return cases, we only care that the traces exhibit the same observable behavior (or a prefix if the trace halts prematurely), as changes in the state will not affect the caller absent a return. In the return case we must additionally guarantee that no changes were made based on secret data that could corrupt the caller upon return.

Consider the result of modifying line 42 of our running example to $r_4 \leftarrow [\text{SP} - 9]$. Instead of reading the proper argument `v`, `g` now reads directly from the secret value of `x` that belongs to `main`. Figure 4 shows m_e, the entry state of `g`, with high confidentiality memory in cyan and low confidentiality in green. Below it, the variant state m'_e agrees on low confidentiality memory, but all other addresses hold -5, in magenta. After the last step, the return states m_r and m'_r should agree on every component that changed since m_e or m'_e. Addresses 108 and 109 have changed, shown in light green, and while 109 had changed in a consistent way (independent of the variation), 108 holds either 43 or -4, so the trace does not obey trace confidentiality.

As another example, changing instruction 30 of the running example to $r_5 \leftarrow [\text{SP} - 6]$ would cause `f` to read the value of `main`'s local variable `x` and then immediately output it, violating the internal observation clause of the confidentiality definition.

To lift trace confidentiality to a system property, we follow the same pattern as in stack integrity:

Definition 5.6. For every initial state (m_0, p_0), we take the induced trace $(m_0, p_0) \hookrightarrow MP$. Then for any n-transition from (m_c, p_c) to (m_e, p_e) in MP, we take the prefix of the trace from the entry point until its corresponding return:

$$(m_e, p_e) \hookrightarrow MP_{call} \mid justret\ m_c$$

We also take the corresponding trace from any variant $m'_e \approx_{Cof(m_c,n)} m_e$:

$$m'_e \hookrightarrow M_{call} \mid justret\ m_c$$

If trace confidentiality holds for all such traces, $\textsc{TraceConf}$ *justret* m_c MP_{call} M_{call}, and it also holds for the toplevel execution trace to account for unallocated memory, $\textsc{TraceConf}$ $\emptyset$ MP M , then the system as a whole enjoys *stack confidentiality* with respect to *cm*.

5.3 Well-bracketed Control Flow

Both the integrity and confidentiality components of local state encapsulation concern themselves only with the accessibility of data, and may still hold on a wide range of programs that violate expected control flow. For instance, if `f` managed to jump in the middle of the code of g with its call, confidentiality and integrity would still be satisfied. However, that is still unwanted behavior: for example, a caller (or a callee) should not be able to bypass a password check by jumping (or returning) in the middle of a block of code — even if such an execution would ostensibly satisfy local state encapsulation. Stack safety enforcement mechanisms like Skorstengaard et al. [2019b] usually aim to prevent such unwanted flows: every jump between functions should either be a call or a return, and each return should be to the instruction after its corresponding call.

To formally capture this aspect of stack safety, we introduce three security properties that rely on the underlying knowledge of the program structure (call map *cm*, function map *fm*, return map *rm*, and entry map *em*): *control separation* ensures the validity of transition sources between functions, *entry integrity* ensures the validity of call targets, and *return integrity* ensures that of return targets.

Definition 5.7. A system enjoys *control separation* if, for any initial state (m_0, p_0) and any adjacent pair of states (m_1, p_1, o_1) and (m_2, p_2, o_2) in the induced trace from (m_0, p_0), if $fm(m_1(\textsc{pc})) \neq fm(m_2(\textsc{pc}))$, then (m_1, p_1) is either a call $cm(m_1(\textsc{pc}))$ or a return $rm(m_1(\textsc{pc}))$.

For example, consider changing line 5 of the running example to a $\textsc{jalr}$, $\textsc{pc} \leftarrow r_4$, with effect $\textsc{pc} \leftarrow 42$, causing `main` to jump into the middle of g. This violates control separation because the instruction that causes a function switch is not a valid call (or a return) according to the call map. On the other hand we do permit functions to jump freely internally.

Definition 5.8. A system enjoys *entry integrity* if, for any initial state (m_0, p_0) and any n-transition from (m_c, p_c) to (m_e, p_e) in the induced trace from (m_0, p_0), m_e is a valid entry:

$$cm(m_c(\textsc{pc})) = n \Rightarrow em(m_e)$$

The changed program described just above also violates entry integrity, because the target (instruction 42) of the new $\textsc{jalr}$ is not an established entry point according to the entry map. Our properties do not enforce which call points jump to which entry points, however; that would require a stronger notion of control flow integrity.

Return integrity captures the expected behavior of marked returns with respect to their call points. Intuitively, it says that the marked return matching a call m_c should satisfy *justret* m_c. Observe that, given an MP-trace $(m_c, p)MP$, if m_c is a call state then the matching return will be the first *unmatched* return in *MP*. So we begin by defining an auxiliary "unmatched" relation between two traces.

Definition 5.9. We write *um MP MP′* if *MP′* is obtained from *MP* by dropping elements until we reach the first unmatched return (according to the return map). Formally, *um* is defined coinductively as follows:

- A trace that starts with a return is related to itself:

$$\frac{cm(m(\textsc{pc})) = \bot \quad rm(m(\textsc{pc}))}{um\ (m,p)\ (m,p)} \qquad \frac{cm(m(\textsc{pc})) = \bot \quad rm(m(\textsc{pc}))}{um\ (m,p)MP\ (m,p)MP}$$

- If a trace MP starts with a non-call, non-return state, its unmatched return must come later:

$$\frac{cm(m(\textsc{pc})) = \bot \qquad \neg rm(m(\textsc{pc})) \qquad um\ MP\ MP'}{um\ (m,p)MP\ MP'}$$

- Finally, if a trace starts with a call, then we must discard the first unmatched return from its tail to get to the real one:

$$\frac{\begin{array}{c} cm(m(\textsc{pc})) = n \qquad \neg rm(m(\textsc{pc})) \\ MP = (m,p)MP_{call} \qquad um\ MP_{call}\ MP_{matched} \\ MP_{matched} = (m',p')MP_{tail} \qquad um\ MP_{tail}\ MP_{unmatched} \end{array}}{um\ MP\ MP_{unmatched}}$$

Definition 5.10. A system enjoys *return integrity* if all of its returns match prior calls and return to the appropriate location. For any initial state and its induced trace $(m_0\, p_0) \hookrightarrow MP$, two conditions must hold. *(i)* MP must not have an unmatched return, i.e., $um\ MP\ MP_{tail}$ must not hold for any MP_{tail}. *(ii)* For any state pair (m_c, p_c) corresponding to a call in MP, if MP_{call} is the suffix of the trace starting at the call, and $mp = (m_r, \_, \_)$ is the first unmatched return such that $um\ MP_{call}\ MP_{tail}$ and $head(MP_{tail}) = mp$, then m_r must correspond to a return state for m_c, i.e., $justret\ m_c\ m_r$ holds.

For example, if we modified the running example by adding an instruction $\textsc{ra} \rightarrow \textsc{ra} - 19$ between 45 and 46, this would cause the return from `g` to go to instruction 8 and return to `main` instead of `f`. While the instruction that causes the return would still be a valid return (satisfying control separation), the state after the return would not correspond to an unmatched return for `f`'s call to `g`, violating return integrity.

Definition 5.11. The above properties are somewhat orthogonal, but together they capture a standard notion of control flow for a stack: functions keep their code separate outside of calls and returns, calls respect explicit entry points, and returns respect implicit entry points corresponding to their call. A system with all three properties (with respect to a function map, call map, return map, and entry map) is said to enjoy *well-bracketed control flow* (with respect to those maps).

6 ENFORCEMENT

In this section we examine how an existing enforcement mechanism, Depth Isolation from Roessler and DeHon [2018], can implement the formal stack-safety property described in the previous section: it in fact enforces a stronger, inductive version of that property which strengthens the local state encapsulation component by checking for integrity and confidentiality violations at every step of the callee's execution trace.

The enforcement policy of Roessler and DeHon relies on a programmable, tag-based reference monitor that runs alongside the program. To control this monitor, they use a programming model that allows fine-grained manipulation of metadata tags to encode so-called *micropolicies* [Azevedo de Amorim et al. 2015]. In such a system, all values in memory addresses and registers (including the $\textsc{pc}$) are enriched with an abstract metadata tag, which can represent arbitrary information about the value. A micropolicy is defined as a set of tags and a collection of software-defined rules, indexed by machine opcode. At each step of the machine, the relevant rule is applied to the tags on the instruction's inputs (PC, registers, memory) and on the instruction itself, and produces one of two outcomes: either the instruction is allowed to execute (and generates tags for the result of the operation and the new PC), or the machine fail-stops with a policy violation. Azevedo de Amorim et al. have shown that a wide range of micropolicies can be defined using this scheme.

Efficient execution of these micropolicies relies on hardware implementations, such as the PUMP architecture [Dhawan et al. 2015]. Tags are represented as word-size bit vectors, stored separately from the regular memory and registers. The hardware incorporates a rule cache to allow quick retrieval of rule outputs for mapped inputs. If the cache misses, the hardware traps to a software handler (running in a privileged context or on a co-processor) to compute the rule result. To obtain adequate performance, it is important to design micropolicies so that they hit in the cache as much as possible. Thus, practical policies maintain a small working set of distinct tags.

6.1 A Conservative Policy

The first micropolicy we present to enforce the definition of stack safety developed in Section 5 is a small variation on the Depth Isolation policy presented by Roessler and DeHon [2018]. Their policy tags the stack memory with ownership information associated to each stack frame (and to each separate object inside that frame, a more finely grained access control that we do not need to consider here), and tags registers containing stack pointers with access permission information. The policy also uses tags on instructions to identify the code sequences that have permission to manage the stack, say during calls and returns.

Our micropolicy works as follows. (We assume for ease of exposition that no arguments are passed on the stack.) The micropolicy maintains tags on values in memory of the form STACK n, indicating locations that belong to the stack frame at activation depth n, or UNUSED; it tags the PC with tags of the form PC n. During normal execution, the micropolicy rules only permit load and store operations when the target memory is tagged with the same depth as the current PC tag. Initially, the entire stack is tagged UNUSED, and the PC has tag PC 0. These tags are altered at exactly those points in the program where the contour of the stack-safety property changes:

- From caller to callee, when the machine executes an instruction marked as a call in the call map. At this point, the current PC n tag is incremented to PC $(n + 1)$, and the function entry sequence initializes all locations of the new function frame with tag STACK $(n + 1)$.
- From callee back to caller, when execution is about to reach the return point *justret c* corresponding to the call at c. The exit sequence retags all locations in the function frame with UNUSED. At the actual return instruction, the PC tag is decremented.

This discipline suffices to enforce local stack encapsulation. To implement it, we define "blessed" instruction sequences intended to appear at the entry and exit of each function, which manipulate tags as just described in addition to performing the usual calling convention tasks of saving/restoring the return address to/from the stack and adjusting the stack pointer. With the aid of an additional tag on the SP register, these sequences also serve to enforce well-bracketed control flow. The micropolicy guarantees atomic execution of these sequences using a combination of tags on the instructions and an additional tag on the PC; we omit the details here.

There remains the question of how to ensure that the sequences are invoked at the right places. For a program to enjoy the stack-safety property (with respect to a particular call map), all we need require is that the entry sequence be initiated at any instruction marked as a call in the call map. This is achieved by giving these instructions (another) special tag, and it is easy to check statically that this has been done correctly. If the code fails to initiate an exit sequence at a point where the stack-safety property expects a return, the micropolicy will incorrectly behave as if execution is continuing in the callee, but since the callee never has stronger access rights than the caller, this is harmless.

Note that in this policy both the entry and exit sequences must write to each element of the frame, which can be quite expensive, especially for programs that allocate large but sparsely populated frames; we return to this point in Section 7. The running example in Section 3 manages the stack in this way. For example, `f`'s "blessed" sequences allocate space to save its return address (and private variables) in instructions 20–21, and later restores that return address and deallocates its private frame space before returning in 34–36. Additionally, unlike in the simplified presentation of the policy, the caller passes information to the callee through the stack, and for this creates and initializes (22 and 24–25) and destructs (29) the argument part of the callee's stack frame.

6.2 A Stronger Lockstep Stack Safety Property

This conservative policy satisfies not only the stack safety policy of Section 5, but also a provably (in Coq) stronger one that enforces local state encapsulation at every point during execution. To encode that property, we not only keep track of the execution of one machine (whose trace we are trying to characterize as safe), but of an entire stack of machine variations corresponding to nested calls of the original machine. For each step of the original machine, we take a step in every variant machine (thus, keeping them in lockstep execution) and enforce a single-step variant of integrity and confidentiality tests. In addition, every time a call is made we push a variation of the original machine state at the time of that call on top of the variant stack (just like the stack-confidentiality property enforced trace confidentiality for every call transition), and every time the machine returns from a call we pop the top variation (to account for stack confidentiality taking the prefix of the transition trace until its return).

We start by defining the integrity and confidentiality tests, as well as the variant stack, before giving the full definition of lockstep stack safety.

Definition 6.1. Given two machine states m and m' and a contour C, these satisfy the *stack integrity test* if their high-integrity components match:

$$int_T\ C\ m\ m' \triangleq \forall k.\ C(k) = \mathrm{HI} \rightarrow m(k) = m'(k)$$

Definition 6.2. Two machine states m_1 and m_2 satisfy the *stack confidentiality test* if the machines they stepped to have their changed components coincide and the observations during these steps are equal:

$$\begin{aligned} conf_T\ m_1\ m_2 \ \triangleq\ & \exists o_1\ o_2\ m_1'\ m_2'.\ m_1 \xrightarrow{o_1} m_1' \wedge m_2 \xrightarrow{o_2} m_2' \Rightarrow \\ & o_1 = o_2 \wedge \forall k.\ (m_1(k) \neq m_1'(k) \vee m_2(k) \neq m_2'(k)) \rightarrow m_1'(k) = m_2'(k) \end{aligned}$$

Definition 6.3. The testing property keeps track of a variation machine for every nested call performed so far on a *variant stack*: a list of machine states, each annotated with its own contour (written as vm_C), and each being a variation of the actual machine according to C.[1]

We also need two auxiliary notations that relate to variant stacks. First, lockstep stack safety will check that the basic tests above hold for every machine in the variant stack. Given a machine that steps $(m, p) \overset{o}{\rightharpoonup} (m', p')$, say that a variant stack vs satisfies the *variant stack test* if for each variant element vm_C (which is a variation of m for C), vm_C can take a step, denoted $vm_C \xrightarrow{o} m'$, while satisfying the stack integrity test $int_T\ C\ m\ m'$ and stack confidentiality test $conf_T\ m\ m'$ with regard to the original machine state. We will write $vs_T\ (m, p)\ vs$ for this compound check. Second, we will write $vs \Longrightarrow vs'$ if every variant machine in vs steps to the corresponding machine in vs'.

We are now ready to define lockstep stack safety coinductively.

Definition 6.4. Given a state $(m, p) \in MPState$ and a variant stack vs in which every element vm_C is a variation of m with respect to its annotated contour C, we say that an MP-trace induced from (m, p) satisfies the *trace lockstep stack safety*, written $stack_T\ MP\ vs$, if the following conditions are satisfied:

[1]The full formal definition in Coq also records additional information (the initial machine state at the call, its initial variation, the contour, and a return predicate) which we elide here for simplicity of exposition.

- If the machine takes a *non-call, non-return step*
$$(m, p) \hookrightarrow_P (m', p', o)MP,$$
then the variant stack test must be satisfied
$$vs_T\ (m, p)\ vs,$$
and trace lockstep stack safety must be coinductively satisfied for the remainder of the trace and the variant stack vs' that vs steps to ($vs \Longrightarrow vs'$):
$$stack_T\ ((m', p', o)MP)\ vs'$$
- If the machine takes a *call step* and C is the contour at the call
$$(m, p) \overset{o}{\rightharpoonup} (m', p') \wedge cm(m) = n,$$
then the variant stack test must be satisfied
$$vs_T\ (m, p)\ vs,$$
and trace lockstep stack safety must be coinductively satisfied for the remainder of the trace and for every variant stack that is the result of adding a variation of m' with respect to C at the top of the stack vs' that vs steps to:
$$stack_T\ ((m', p', o)MP)\ (vm_C : vs')$$
- If the machine m takes a *return step*,
$$(m, p) \overset{o}{\rightharpoonup} (m', p') \wedge rm(m),$$
then the top of the variant stack also returns, the tail of vs satisfies the stack safety test
$$vs_T\ (m, p)\ tail(vs),$$
and trace lockstep stack safety must be coinductively satisfied for the remainder of the trace and for the tail of the variant stack that vs steps to:
$$stack_T\ ((m', p', o)MP)\ (tail(vs'))$$
- If the machine *fail-stops*, i.e., $(m, p) \hookrightarrow_P (m, p, \tau)$, the property is satisfied for the singleton trace: $stack_T\ [(m, p)]\ vs$.

Definition 6.5. A system satisfies *lockstep stack safety* if, for every initial machine (m, p), the trace induced from (m, p) satisfies trace lockstep stack safety with respect to the empty stack.

Theorem. Lockstep stack safety implies stack safety.

The proof is a fairly involved coinductive argument (around 3K LoC in Coq) on the definition of trace lockstep stack safety, using an auxiliary depth-index to ensure the proper alignment of calls and returns.

7 STACK SAFETY PROPERTIES FOR LAZY ENFORCEMENT

The conservative policy described in the previous section, while testable and enforceable, is rather slow to be of practical use. What we want, instead, is a somewhat more permissive policy that can also be efficiently implemented. Most of the performance overhead incurred stems from the need to set stack activation tags as a frame is created and reset them as it is destructed, as Roessler and DeHon [2018] confirm in their evaluation. To mitigate those costs, they propose optimizations called Lazy Tagging and Lazy Clearing that relax and defer some of the checks of the Depth Isolation policy. Under these optimizations, the policy does not initialize stack frames on entry or clear them on exit (thus speeding up execution), and it permits all writes to the stack, even when the pc tag does not match the memory tag. However, reads from the stack do still require PC tag and memory tag to match. Thus, even if a callee illicitly writes to a private location in its caller, the caller will eventually detect this if it ever tries to read from that location. These *lazy policies* admit more efficient implementations, but they deliberately allow violations of stack integrity temporarily, with the checks deferred until the point a violation truly becomes harmful. The natural question then is, how do we characterize the protections provided by these policies once stack safety is broken? How does a harmful violation that must be caught later differ from a harmless one? The properties in this section rely on our notion of observations to draw this distinction.

7.1 Observable Integrity

Rolling Back to Idealized State. Observable integrity weakens eager integrity by allowing a returned state to differ from its call state at high integrity components, provided that the differences do not change the observable behavior of the rest of the program. The challenge here is to define what constitutes a change in behavior. We construct an idealized return state that contains only the changes that the caller was allowed to make, and use the trace from that state as a baseline against which the real trace will be compared.

Definition 7.1. Given a contour, a call state, and a returned state, we define a *rollback* function $roll \in \mathcal{C} \times \mathcal{M} \times \mathcal{M} \rightarrow \mathcal{M}$. The rollback returns a state that matches the call state on those components that are high integrity in its contour, and matches the return state on low integrity components:

$$roll(C, m_e, m_r)(k) = \begin{cases} m_e(k) & \text{if } C(k) = (\mathrm{HI}, \_) \\ m_r(k) & \text{if } C(k) = (\mathrm{LI}, \_) \end{cases}$$

If m_e is an entry to a call with contour C and m_r its return, the result of $roll(C, m_e, m_r)$ agrees with the entry m_e on every component the callee should not change, and with the actual return m_r on those components that it is permitted to change. So the observable behavior that follows is the idealized behavior to which we compare the trace from m_r.

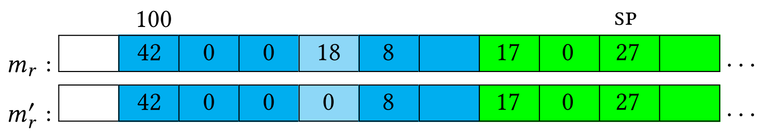
Now we can define a version of trace integrity that compares the trace from a returned state to that from its idealized counterpart.

Definition 7.2. A trace enjoys *observable trace integrity* with respect to some contour C if the trace induced by its final state (if any) has observable behavior that prefixes that of its idealized version (to account for premature halts due to policy violations):

$$\begin{aligned} \textsc{ObsTraceInt}\ C\ MP \ \triangleq\ & head(MP) = (m_e, \_, \_) \ \wedge\ last(MP) = (m_r, p_r, \_) \wedge \\ & (m_r, p_r) \hookrightarrow MP' \ \wedge\ roll(C, m_e, m_r) \hookrightarrow M' \Rightarrow \\ & \pi_o(MP') \lesssim_O \pi_o(M') \end{aligned}$$

For our example, we return to the violation of trace integrity in Fig. 3, in which g has written to z and returned. Whether observable integrity holds depends on the continuing trace, shown in Fig. 5. Above, we have the final state of MP, m_r, and the rolled-back state, $m_r' = roll(C, m_e, m_r)$, in which high integrity components are restored to their values as of m_e. Specifically, address 103 is restored to 0. When instruction 31 prints z in the trace from m_r, it prints 18, but in the trace from m_r' it prints 0. Only then is observable trace integrity violated.

However, consider instead if we modify the original code of Fig. 2 to insert after instruction 44 a new instruction $44.5 : [\mathrm{SP} - 4] \leftarrow 50$, overwriting w in f's frame. So m_r' maps address 105 to 50, while the rolled-back state m_r' maps it to 0. But in the trace from either state, instructions 27 and 28



$\text{PC} = 27 \quad r_0 = 0 \quad \text{RA} = 48 \quad \text{SP} = 108 \quad r_4 = 18 \quad r_5 = ?$

27	$r_4 \leftarrow [\text{SP}]$	$r_4 \leftarrow 0$	Load return value to r_4
28	$[\text{SP} - 3] \leftarrow r_4$	$[105] \leftarrow 0$	Set w to return value
29	$\text{SP} \leftarrow \text{SP} - 2$	$\text{SP} \leftarrow 106$	Deallocate g's call frame
30	$r_5 \leftarrow [\text{SP} - 4]$	$r_5 \leftarrow 18/0$	Load argument z r_5
31	$[\text{OUT}] \leftarrow r_5$	$[99] \leftarrow 18/0$	Print z

$$\pi_o(MP) \not\approx \pi_o(M')$$

Fig. 5. Rolling back from Figure 3.

load the return value (18) and store it to 105, eliminating the only difference. The remainder of the observation traces must be identical, so this trace does obey observable trace integrity.

Definition 7.3. A system enjoys *observable stack integrity*, like stack integrity, if, for any initial state and each n-transition from (m_c, p_c) to (m_e, p_e) in its induced trace, we take the prefix of the call $(m_e, p_e) \hookrightarrow MP_{call} \mid justret\ m_c$ and observable trace integrity holds on MP_{call} with contour $Cof(m_c, n)$.

7.2 Observable Confidentiality

Two of the cases of trace confidentiality (Definition 5.5), *InfCase* and *FailCase*, are defined solely in terms of observation traces. If the traces are infinite, they should be similar; if one fail-stops, it should be a prefix. Only the *ReturnCase* checks machine states, because in that case the traces could continue, but have been cut off by a return. The key to making confidentiality lazy is not to perform that check but rather to consider the traces after that return, checking only whether they are observationally similar.

This requires restoring varied components to their original values in the trace from the variant state, since after return they will once again be legitimately accessible. To return to the example in Fig. 4, the varied final state m'_r still has -5 as a return address for f, so the trace after that return will behave oddly in a way that does not reflect a confidentiality violation. That means we cannot immediately compare traces from m_r and m'_r, but rather that we must restore the varied values while preserving differences that have occurred during execution.

Definition 7.4. The *restore* function takes four states: the original entry state, a variant entry state, and corresponding return states. It returns another state. The resulting state restores those components that varied between entries to match the original entry, unless they were changed during execution:

$$restore(m_e, m'_e, m_r, m'_r)(k) = \begin{cases} m_e(k) & \text{if } m_e(k) \neq m'_e(k) \text{ and} \\ & m_e(k) = m_r(k) \text{ and } m'_e(k) = m'_r(k) \\ m'_r(k) & \text{else} \end{cases}$$

Now we can describe a trace property akin to trace confidentiality, but continuing execution after the end of the callee trace.

Definition 7.5. Observable trace confidentiality is the lazy analog to trace confidentiality, and reuses the cases in which the traces do not return. For a return predicate R and a pair of traces MP and M, we consider the return case, in which MP has a last element $(m_r, \_, \_)$ and that machine state is a return, i.e. $R\ m_r$ holds. Observable trace confidentiality then requires that M also has a final state $(m'_r, \_)$ that is a return, i.e. $R\ m'_r$ holds. In addition, just like in the eager case, observations of MP are similar to those of M.

The only thing that changes is the requirement that m_r and m'_r agree on components. Instead, let m_e be the machine state of $head(MP)$, and m'_e be that of $head(M)$. We restore m'_r based on m_e and m'_e to get $m''_r = restore(m_e, m'_e, m_r, m'_r)$. Then observable trace confidentiality requires that the induced trace from (m_r, p_r) should be an observational prefix of the induced trace from m'_r, reflecting the possibility of a fail-stop after return.

$$\begin{aligned} ObsReturnCase\ R\ ((m_e,\_,\_)MP)\ ((m'_e,\_)M) \ \triangleq\ & last(MP) = (m_r, p_r, \_) \ \wedge\ last(M) = (m'_r, \_) \wedge \\ & m''_r = restore(C, m_e, m'_e, m_r, m'_r) \wedge \\ & (m_r, p_r) \hookrightarrow MP' \ \wedge\ m''_r \hookrightarrow M' \Rightarrow \\ & \pi_o(MP) \eqsim \pi_o(M) \ \wedge\ \pi_o(MP') \lesssim \pi_o(M') \end{aligned}$$

We combine this case with the other two cases from ordinary trace confidentiality:

$$\textsc{ObsTraceConf}\ R\ MP\ M \ \triangleq\ ObsReturnCase\ R\ MP\ M\ \wedge InfCase\ MP\ M \wedge FailCase\ R\ MP\ M$$

In our example, Fig. 6 shows how execution continues from m'_r and the restored

$$m''_r = restore(C, m_e, m'_e, m_r, m'_r).$$

Addresses 107 and 108 are not restored; every address that was varied and not changed in one of the executions is. So 108 contains 43 in the trace from m_r, and after returning from `f`, `main` prints 85. But in the trace from m''_r, 108 contains -4, and ultimately 38 is printed. So observable trace confidentiality does not hold. This example illustrates the weakness of observable confidentiality as a testing property: it takes until the very end of the program to determine whether a tainted value actually influences observable behavior.

On the other hand, suppose that we inserted an instruction at 43: $r_6 \leftarrow [\textsc{sp} - 8]$. r_6 would hold 42 in m_r and -5 in both m'_r and m''_r. But since r_6 is never used otherwise, the traces of `f` will enjoy observable trace confidentiality.

Definition 7.6. Observable stack confidentiality is structured like stack confidentiality. For each initial state (m_0, p_0) and n-transition from (m_c, p_c) to (m_e, p_e) in the induced trace $(m_0, p_0) \hookrightarrow MP$, take any variation $m'_e \approx_{Cof(m_c,n)} m_e$. Let $(m_e, p_e) \hookrightarrow MP_{call} \mid justret\ m_c$ and $m'_e \hookrightarrow M_{call} \mid justret\ m_c$. We must have $\textsc{ObsTraceConf}\ justret\ m_c\ MP_{call}\ M_{call}$.

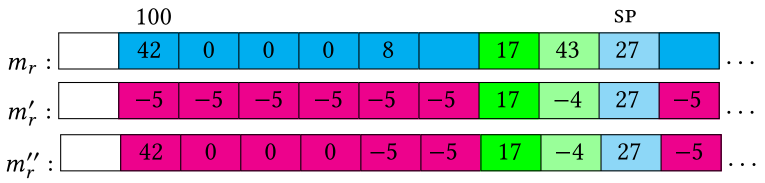
Additionally, for any variation $m'_0 \approx_{Cof(m_0),0} m_0$, taking the induced trace $m'_0 \hookrightarrow M$, we must have $\textsc{TraceConf}\ \emptyset\ MP\ M$.

A system enjoys observable stack confidentiality if observable confidentiality holds in all such cases.

Definition 7.7. A system enjoys *observable stack safety* if it enjoys both observable stack integrity and observable stack confidentiality with respect to them.

Theorem. Stack safety implies observable stack safety.

This is more straightforward than the implication in the previous section (1,5K LoC in Coq) as the recursive structure of the two properties is similar.



$\text{PC} = 27 \quad r_0 = 0 \quad \text{RA} = 32 \quad \text{SP} = 108 \quad r_4 = 18 \quad r_5 =?$

27	$r_4 \leftarrow [\text{SP}]$	$r_4 \leftarrow 43/-4$	Load return value to r_4
28	$[\text{SP} - 3] \leftarrow r_4$	$[105] \leftarrow 43/-4$	Set `w` to returned value
29	$\text{SP} \leftarrow \text{SP} - 2$	$\text{SP} \leftarrow 106$	Deallocate `g`'s call frame
30	$r_5 \leftarrow [\text{SP} - 4]$	$r_5 \leftarrow 0$	Load argument `z` r_5
31	$[\text{OUT}] \leftarrow r_5$	$[99] \leftarrow 0$	Print `z`
32	$r_4 \leftarrow r_4 + r_5$	$r_4 \leftarrow 43/-4$	Add r_4 and r_5
33	$[\text{SP} - 3] \leftarrow r_4$	$[103] \leftarrow 43/-4$	Store r_4 as ther return of `f`
34	$\text{RA} \leftarrow [\text{SP} - 2]$	$\text{RA} \leftarrow 8$	Load return address
35	$\text{SP} \leftarrow \text{SP} - 2$	$\text{SP} \leftarrow 104$	Deallocate `f`'s local state
36	$\text{PC} \leftarrow \text{RA}$	$\text{PC} \leftarrow 8$	Return to `main` (JALR)

8	$r_4 \leftarrow [\text{SP} - 1]$	$r_4 \leftarrow 43/-4$	Load return value to r_4
9	$[\text{SP} - 3] \leftarrow r_4$	$[101] \leftarrow 43/-4$	Set `y` to the result
10	$\text{SP} \leftarrow \text{SP} - 2$	$\text{SP} \leftarrow 102$	Deallocate `f`'s call frame
11	$r_5 \leftarrow [\text{SP} - 2]$	$r_5 \leftarrow 42$	Load `x` to r_5
12	$r_4 \leftarrow r_4 + r_5$	$r_4 \leftarrow 85/38$	Add `x` and `y`
13	$[\text{OUT}] \leftarrow r_4$	$[99] \leftarrow 85/38$	Print `x + y`

$$\pi_o(MP) \not\approx \pi_o(M)$$

Fig. 6. Continuing from Fig. 4.

Connection to Policies. Observable properties allow us to defer enforcement until a property violation would become visible, as in lazy policies. But by formalizing the connection to the eager policy, we can identify when lazy policies miss violations that can become visible. The optimized lazy tag policy from Roessler and DeHon [2018] tags each stack slot with the call depth at which it was written, and enforces that it must be read from the same depth. However, this allows violations of observable stack safety! Say we have functions `foo`, `bar` and `baz`, where `foo` calls `bar`, and then after `bar` returns `foo` calls `baz`. Since `bar` and `baz` are at the same stack depth, if `bar` writes to `foo`'s frame, `baz` can then read the tainted write and use it to create an observable difference that violates both observable integrity and confidentiality without violating the policy.

The lazy policy can be repaired if, instead of tagging each function activation with its depth in the stack, we generate a fresh activation identifier on each call, which prevents a program from exploiting stale tags from previous activations. A related mechanism was explored in the Static Authorities policy of Roessler and DeHon [2018], which associates a unique activation identifier to each function, which is however shared by all activations of the same function in order to obtain good cacheability of tag rules. Static Authorities comes closer to observable stack safety, though the example above would still exist if `bar` and `baz` were merged into a single function that tracked whether it had been called previously and changed its behavior accordingly.

8 RELATED WORK

Formal Stack Safety on Capability Machines. To our knowledge, the first and only line of work that has attempted to furnish a positive and formal characterization of the meaning of stack safety is Skorstengaard et al. [2019a], who introduce a calling convention that uses local capabilities to preserve local state encapsulation and well-bracketed control flow, using a logical relation to reason about the stack safety of concrete programs. Although the required hardware support is readily available in capability machines like CHERI, this technique incurs significant costs, because it requires the entire unused part of the stack to be cleared whenever a security boundary is crossed. Their logical relation captures capability safety without "externally observable side-effects (like console output or memory access traces)." In the discussion, "while [the authors] claim that [their] calling convention enforces control-flow correctness, [they] do not prove a general theorem that shows this, because it is not clear what such a theorem should look like," noting that the correctness property enjoyed by their technique "is not made very explicit."

StkTokens [Skorstengaard et al. 2019b] is a natural continuation of that work. Like its predecessor, it aims to protect the stack by enforcing local state encapsulation and well-bracketed control flow. It does so by defining a new calling convention that makes use of linear capabilities for stack and return pointers. The convention operates on a single shared stack and requires that protected components avoid compromising their own security by following certain simple rules—like not leaking their private capability seals. More precisely but still informally, local state encapsulation is defined as restricting accesses to the range of memory allocated to the current stack frame, and well-bracketed control flow as only allowing returns from the topmost frame to the immediately adjacent frame below. Formally, it improves on Skorstengaard et al. [2019a] by building those properties into the semantics of a capability machine with a built-in call stack and call and return instructions, which is proven fully abstract with respect to a more concrete capability machine that replaces those pseudo-instructions with their calling convention. Their proof of full abstraction uses a standard notion of components, which import and export functions through their interfaces; their model of observations is limited to cotermination.

We believe that the linear capability machines introduced by [Skorstengaard et al. 2019b], in combination with the StkTokens calling convention, can be modeled in our framework and satisfy our definition of stack safety in Section 5. Further, we expect them to satisfy the even stronger property developed in Section 6. Proving these conjectures would involve materializing their definitions of local state encapsulation and well-bracketed control flow, which exist only in enmeshed form as part of the semantics of the capability machines. The main practical limitation of StkTokens is its reliance on linear capabilities, as it is unclear now they could be added to practical capability machines, especially in terms of efficiency—previous work on micropolicies [Juglaret et al. 2015] has shown how to use that framework to implement linear return capabilities.

Protecting the Stack with Micropolicies. Roessler and DeHon [2018] consider a standard attacker model where all attacks against stack data are in scope (but not side channels or hardware attacks), and study the protection of stack data through three families of micropolicies that tag stack objects with a pair of frame and object identifiers used to validate accesses to the stack: Return Address Protection (which prevents an adversary from overwriting designated return addresses), Static Authorities (which only allows the code of a function to access the stack frames of its own dynamic instances), and Depth Isolation (described in Section 6). All these policies exploit various kinds of spatial and temporal locality of stack memory and local call graphs, as well as information generated by the compilation toolchain, to strike various balances between precision and cacheability, evaluated through benchmarks that demonstrate limited performance overhead. In addition to the baseline "eager" policies, they propose a number of lazy optimizations, notably Lazy Tagging and Lazy Clearing, discussed in Section 7, with associated improvements in performance.

Heap Safety as a Security Property. Heap safety, like stack safety, may be framed as a security property in the form of noninterference [de Amorim et al. 2018]. Just like de Amorim et al. give a rigorous characterization of the meaning of (heap) memory safety, in this paper we aim to do the same for the stack. Their model describing safety of shared stack-allocated objects extends naturally to heap safety, complicated by the fact that these objects are not deallocated by returns, and therefore a function's privilege may increase or decrease after its entry point. For instance, if a callee allocates a heap object and returns the pointer to its caller, that object's addresses become accessible in contradiction to a contour computed at the caller's entry. So such an extension must enforce trace confidentiality and integrity properties separately on continuous segments of each call, with contours computed at each crossing between caller and callee.

9 FUTURE WORK

We have presented a pure notion of stack safety that omits many complicating factors common to real systems. In particular, our model as presented exploits the simplifying assumption that privilege never increases during a function call, and that therefore a contour computed once on the entry to a function describes its privilege throughout. Our future plans are to extend the model with common language features, some of which violate this assumption. Here we describe the basic principles by which our model can be extended to some of these cases. We also plan to develop a testing framework for quickly checking the validity of different enforcement mechanisms.

Stack-derived Pointers. Common programming idioms involve a caller passing a pointer to its local data into a callee. Using standard notions of pointer provenance [Memarian et al. 2019], we can extend our model to distinguish safe use of this idiom from true stack safety violations. We require additional annotation to identify which addresses within a function's stack frame correspond to distinct objects. Then a valid pointer to an object is one derived through legal arithmetic from its base address, and at each call point, an object's addresses are marked low confidentiality and integrity if a valid pointer exists in a register or in the transitive closure of the callee's accessible memory. All pointers to an object cease to be valid when it is deallocated along with its stack frame.

Non-stack control flow. Control structures beyond well-bracketed calls and returns require modifications to the model. Tail calls, for instance, reuse the caller's stack frame for its callee, and every nested tail call returns simultaneously to the top non-tail-calling function. Under normal stack integrity, the first function in a chain of tail calls returns and violates integrity, while subsequent tail calls never actually return at all. Instead we need a variant that treats the entry of a tail call as the return of its caller.

More complex is the addition of a coroutine model, in which multiple stacks respect stack safety internally, but might also yield to one another. Here we must distinguish three levels of integrity: accessible, in the same stack, and in another stack. Stack integrity requires that same-stack data be unchanged when a function returns, but other-stack data may change. A similar *yield integrity* property requires that, from a yield out of a coroutine to the next yield back into it, all accessible and same-stack data are unchanged. Confidentiality properties are split similarly.

Random Testing. Part of the motivation for the strong lockstep stack-safety property is that it uncovers errors quickly, and therefore should be quicker to test, just like the smarter testing properties of Hriţcu et al. [2013]. We have already developed a testing framework for debugging our policy enforcement for this property, and it would be interesting to explore whether a testing-amenable variant of the lazy property also exists.

REFERENCES

Arthur Azevedo de Amorim, Maxime Dénès, Nick Giannarakis, Cătălin Hriţcu, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. 2015. Micro-Policies: Formally Verified, Tag-Based Security Monitors. In *36th IEEE Symposium on Security and Privacy (Oakland S&P)*. IEEE.

David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. 2015. Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS âĂŽ15)*. Association for Computing Machinery, New York, NY, USA, 117âĂŞ130. https://doi.org/10.1145/2694344.2694367

Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7* (San Antonio, Texas) *(SSYMâĂŽ98)*. USENIX Association, USA, 5.

Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2015. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (Singapore, Republic of Singapore) *(ASIA CCS âĂŽ15)*. Association for Computing Machinery, New York, NY, USA, 555âĂŞ566. https://doi.org/10.1145/2714576.2714635

Arthur Azevedo de Amorim, Catalin Hritcu, and Benjamin C. Pierce. 2018. The Meaning of Memory Safety. In *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10804)*, Lujo Bauer and Ralf Küsters (Eds.). Springer, 79–105. https://doi.org/10.1007/978-3-319-89722-6_4

Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. HardBound: Architectural Support for Spatial Safety of the C Programming Language. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems*. 103–114. http://acg.cis.upenn.edu/papers/asplos08_hardbound.pdf

Udit Dhawan, Cătălin Hriţcu, Rafi Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Jr., Benjamin C. Pierce, and André DeHon. 2015. Architectural Support for Software-Defined Metadata Processing. http://ic.ese.upenn.edu/abstracts/sdmp_asplos2015.html

J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*. 11–11.

Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. 2013. Testing Noninterference, Quickly. In *18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. http://www.crash-safe.org/node/24 Full version in Journal of Functional Programming, special issue for ICFP 2013, 26:e4 (62 pages), April 2016. Technical Report available as arXiv:1409.0393.

Yannis Juglaret, Cătălin Hriţcu, Arthur Azevedo de Amorim, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. 2015. Towards a Fully Abstract Compiler Using Micro-Policies: Secure Compilation for Mutually Distrustful Components. Technical Report, arXiv:1510.00697. http://arxiv.org/abs/1510.00697

Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDIâĂŽ14)*. USENIX Association, USA, 147âĂŞ163.

Kayvan Memarian, Victor Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert Watson, and Peter Sewell. 2019. Exploring C semantics and pointer provenance. *Proceedings of the ACM on Programming Languages* 3 (01 2019), 1–32. https://doi.org/10.1145/3290380

Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 245–258. http://repository.upenn.edu/cgi/viewcontent.cgi?article=1941&context=cis_reports

Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *9th International Symposium on Memory Management*. ACM, 31–40. http://acg.cis.upenn.edu/papers/ismm10_cets.pdf

Nick Roessler and André DeHon. 2018. Protecting the Stack with Metadata Policies and Tagged Hardware. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 478–495. https://doi.org/10.1109/SP.2018.00066

Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. 2019. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy* (Phoenix, AZ, USA) *(HASP âĂŽ19)*. Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages. https://doi.org/10.1145/3337167.3337175

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019a. Reasoning about a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management. *ACM Trans. Program. Lang. Syst.* 42, 1, Article 5 (Dec. 2019), 53 pages. https://doi.org/10.1145/3363519

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019b. StkTokens: Enforcing Well-Bracketed Control Flow and Stack Encapsulation Using Linear Capabilities. *Proc. ACM Program. Lang.* 3, POPL, Article 19 (Jan. 2019), 28 pages. https://doi.org/10.1145/3290332

Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA) *(ISCA âĂŹ14)*. IEEE Press, 457âĂŞ468.