Formalizing Stack Safety as a Security Property

Sean Noble Anderson
Portland State University
ander28@pdx.edu

Roberto Blanco
Max Planck Institute for Security and Privacy
roberto.blanco@mpi-sp.org

Leonidas Lampropoulos
University of Maryland, College Park
leonidas@umd.edu

Benjamin C. Pierce
University of Pennsylvania
bcpierce@cis.upenn.edu

Andrew Tolmach
Portland State University
tolmach@pdx.edu

***Abstract*—The term *stack safety* is used to describe a variety of compiler, run-time, and hardware mechanisms for protecting stack memory. Unlike "the heap," the ISA-level stack does not correspond to a single high-level language concept: different compilers use it in different ways to support procedural and functional abstraction mechanisms from a wide range of languages. This protean nature makes it difficult to nail down what it means to correctly enforce stack safety.**

We propose a formal characterization of stack safety using concepts from language-based security. Rather than treating stack safety as a monolithic property, we decompose it into an integrity property and a confidentiality property for each of the caller and the callee, plus a control-flow property—five properties in all. This formulation is motivated by a particular class of enforcement mechanisms, the "lazy" stack safety micro-policies studied by Roessler and DeHon [1], which permit functions to write into one another's frames, but which taint the changed locations so that the frame's owner cannot access them. No existing characterization of stack safety captures this style of safety. We capture it here by stating our properties in terms of the observable behavior of the system.

Our properties go further than previous formal definitions of stack safety, supporting caller- and callee-saved registers, arguments passed on the stack, and tail-call elimination. We validate our properties by using them to distinguish between correct and incorrect implementations of Roessler and DeHon's micro-policies using property-based random testing. Our test harness successfully identifies several broken variants, including Roessler and DeHon's lazy policy; a repaired version of their policy does pass our tests.

I. Introduction

Functions in high-level languages, and related abstractions such as procedures, methods, etc., are units of computation that call one another to define larger computations in a modular way. At a low level, each function activation manages its own data—local variables, spilled temporaries, etc.—as well as information about the *caller* to which it must return. The *call stack* is the fundamental data structure used to implement functions, aided by an Application Binary Interface (ABI) that defines how registers are shared between activations.

From a security perspective, attacks on the call stack are attacks on the function abstraction itself. Indeed, the stack is an ancient [2] and perennial [3]–[8] target for low-level attacks, sometimes involving control-flow hijacking via corrupting the return address, sometimes memory corruption more generally. The variety in attacks on the stack is mirrored in the range of software and hardware protections that aim to prevent them, including stack canaries [9], bounds checking [10]–[12], split stacks [13], shadow stacks [14], [15], capabilities [16]–[20], and hardware tagging [1], [21].

Enforcement mechanisms can be brittle, successfully eliminating one attack while leaving room for others. To avoid an endless game of whack-a-mole, we seek formal properties of safe behavior that can be proven, or at least rigorously tested. Such properties can be used as the specification against which enforcement can be validated; even enforcement mechanisms that do not fulfill a property benefit from the ability to articulate why and when they may fail.

Of the mechanisms listed above, many are fundamentally ill-suited for offering formal guarantees: they may impede attackers, but do not provide universal protection. Shadow stacks, for instance, aim to "restrict the flexibility available in creating gadget chains" [15], not to categorically rule out attacks. Other mechanisms, such as SoftBound [10] and code-pointer integrity [13], do aim for stronger guarantees, but not typically formal ones. To our knowledge, the sole line of work making a formal claim to protect stack safety is the study of secure calling conventions by Skorstengaard et al. [19] and Georges et al. [20].

Mechanisms besides this line of work should also be amenable to strong formal guarantees. In particular, Roessler and DeHon [1] present an array of tag-based micro-policies [22] for stack safety that offer universal protection, but not yet tied to a particular formal guarantee. Their most realistic micro-policy, called Lazy Tagging and Clearing (LTC), makes an interesting performance trade-off: it allows function activations to write improperly into one another's stack frames, but ensures that the owner of the corrupted memory cannot access it afterward. Under this policy, one function activation *can* corrupt another's memory—just not in ways that affect observable behavior. Therefore, LTC would not fulfill Georges et al.'s property (adapted to the tagged setting). But LTC does arguably enforce stack safety. A looser, more observational definition of stack safety is needed to fit this situation.

We propose here a formal characterization of stack safety, based on the intuition of protecting function activations (both registers and stack frames) from each other. We use the tools

of language-based security [23], treating function activations as security principals. We decompose stack safety into a family of properties describing the *integrity* and *confidentiality* of the caller's local state and the callee's behavior during (and after) the callee's execution, plus the *well-bracketed control flow* (WBCF) property articulated by Skorstengaard et al. [19]. While our properties are motivated by the desire to specify LTC precisely, they are stated abstractly in the hope that they can also be applied to other enforcement mechanisms.

We are not attempting to present a universal definition of stack safety for all systems. While many security properties can be described at the level of a high-level programming language and translated to a target machine by a secure compiler, stack safety cannot be defined in this way, since "the stack" is not explicitly present in the definitions of most source languages. It is implied by the semantics of features such as calls and returns, but is not itself a feature.[1] But neither can stack safety be an entirely satisfying low-level property; indeed, at the lowest level, the specification of a "well-behaved stack" is almost vacuous. The ISA is not concerned with such questions as whether a caller's frame should be readable or writable to its callee. Those are the purview of high-level languages built atop the hardware stack.

Therefore, any low-level treatment of stack safety must begin by asking: which high-level features does our system support, and what are their security requirements? Different answers will beget different properties. We identify and define five security requirements that a system might try to enforce upon a function call: well-bracketed control flow, and the integrity and confidentiality of each of the caller and the callee. We apply these first to a simple system with very few features, then to a more realistic one supporting tail-call elimination, argument passing on the stack, and callee-save registers.

We show that our properties are useful for distinguishing between correct and incorrect enforcement using QuickChick [25], [26], a property-based random testing tool for Coq. We find that LTC is flawed in a way that undermines both integrity and confidentiality; after correcting this flaw, the repaired LTC satisfies all our properties. Moreover, we modify LTC to protect the features of our more realistic system, and apply random testing to validate this extended protection mechanism against the extended properties.

In sum, we offer the following contributions:

- We give a novel characterization of stack safety as a collection of properties: confidentiality and integrity for callee and caller, plus well-bracketed control-flow. The properties are parameterized over a notion of external observation, allowing them to characterize lazy enforcement mechanisms.
- We extend these core definitions to describe a realistic setting with argument passing on the stack, callee-saves registers, and tail-call elimination. Our model is modular enough that handling these features is straightforward.
- We validate a tag-based enforcement mechanism, *Lazy Tagging and Clearing*, via property-based random testing, find that it falls short, and propose and validate a fix.

In the next section, we give a brief overview of our framework and assumptions. In Section III, we walk through a function call in a simple example machine and discuss informally how each of our properties applies to it. In the process we motivate the properties from a security perspective. In Section IV we formalize the machine model, its *security semantics*, and the stack safety properties built on these. Section V describes how to support an extended set of features. In Section VI we describe the micro-policies that we test, in Section VII the testing framework itself, and in Sections VIII and IX related and future work.

The accompanying artifact [APT: what is the state of that? Depending on conference preferences, should give a URL or other pointer.] contains formal definitions (in Coq) of our properties, plus our testing framework. It does not include proofs: we use Coq primarily for the QuickChick testing library and to ensure that our definitions are unambiguous. Formal proofs are left as future work.

II. Framework and Assumptions

Stack safety properties need to describe the behavior of machine code, but they naturally talk about function activations and stack contents—abstractions that are typically not visible at machine level. To bridge this gap, our properties are defined in terms of a *security semantics* layered on top of the standard execution semantics of the machine. The security semantics identifies certain state transitions of the machine as *security-relevant operations*, which update a notional *security context*. This context consists of an (abstract) stack of function activations, each associated with a *view* that maps each machine *state element* (memory location or register) to a *security class* (active, sealed, etc.) specifying how the activation can access the element. The action of a security-relevant operation on the context is defined by a set of rules that describe how the machine code implements the function abstractoin using the stack and registers.

Given the security classes of the machine's state elements, we define high-level security properties—integrity, confidentiality, and well-bracketed control flow—as predicates that must hold on each call. These predicates draw on the idea of *variant* states from the theory of non-interference, and a notion of *observable events*, which might include specific function calls (e.g., system calls that perform I/O), writes to special addresses representing memory-mapped regions, etc. For example, to show that certain locations are kept secret, it suffices to compare the execution of states which vary at those locations and check that their traces of observable events are the same. This structure allows us to talk about the eventual implact of leaks or memory corruption without reference to internal implementation details, and to support lazy enforcement by flagging corruption of values only when it can actually impact visible behavior.

[1] Contrast Azevedo de Amorim et al.'s work on heap safety [24]: the concept of the heap figures directly in high-level language semantics and its security is therefore amenable to a high-level treatment.

We introduce these properties by example in Section III and formally in Section IV. In the remainder of this section we describe the underlying framework in more detail.

Machine Model: We assume a conventional ISA (e.g. RISC-V, x86-64, etc.), with registers including a program counter and stack pointer. We make no particular assumptions about the provenance of the machine code; in particular, we do not assume the use of any particular compiler. The machine may possibly be enhanced with enforcement mechanisms such as hardware tags [21], [27] or capabilities [16]. If so, we assume that the behavior of these mechanisms is incorporated into the basic step semantics of the machine, but we keep the enforcement state unchanged when constructing variant states. Failstop behavior by enforcement mechanisms is modeled as stepping to the same state (and thus silently diverging).

Security Semantics and Property Structure: A security semantics extends a machine with additional context about the identities of current and pending functions (which act as security principals) and about the registers and memory they require to be secure. This added context is purely notional; it does not affect the behavior of the real machine. The security context evolves dynamically through the execution of security-relevant operations, which include calls, returns, and frame manipulation. Our security properties are phrased in terms of this context, often as predicates on future states ("when control returns to the current function...") or as relations on traces of future execution (hyper-properties).

Security-relevant operations abstract over the implementation details of the actions they take. In this paper, each operation corresponds to a single underlying machine instruction. One instruction may perform multiple operations. We assume that a compiler or other trusted source has provided labels to disambiguate instructions that have multiple purposes. For instance, in the tagged RISC-V architecture that we use in our examples and tests, calls and returns are conventionally performed using the `jal` ("jump-and-link") and `jalr` ("jump-and-link-register") instructions, but these instructions might also be used for other things.

We end up with an annotated version of the machine transition function written $m \xrightarrow{\bar{\psi},e} m'$, where m and m are machine states, e is an (optional) external event, and $\bar{\psi}$ is a list of security-relevant operations. We then lift this into a transition between pairs of machine states and contexts, by applying a set of operation-specific rules to transform each state and context to a new context, in parallel with the ordinary transition on states. The most important rules describe call and return operations. A call pushes a new view onto the context stack and changes the class of the caller's data to protect it from the new callee; a return reverses these steps. Other operations can serve to signal how parts of the stack frame are being used to store or share data, and their corresponding rules alter the classes of different state elements accordingly.

Exactly which operations and rules are needed depends on what code features we wish to support. The set of security-relevant operations (Ψ) covered in this paper is given in

Operation $\psi \in \Psi$	Parameters	Sections
call	target address, argument registers	III,IV
	stack arguments (base, offset & size)	V,VII
return		III,IV
alloc	offset & size	III,IV
	public flag	V,VII
dealloc	offset & size	III,IV
tailcall	(same as for **call**)	V,VII
promote	register, offset & size	V-C
propagate	source register/address	V-C
	destination register/address	V-C
clear	target register/address	V-C

TABLE I: Security-relevant operations and their parameters, with the Sections in which they are first used in the paper.

Table I. A core set of operations covering calls, returns, and local memory is introduced in the example in Section III and formalized in Section IV. An extended set covering simple memory sharing and tail-call elimination is described in Section V and tested in Section VII. The remaining operations are needed for the capability-based model in Section V-C.

Views and Security Classes: The security context consists of a stack of views: functions that map each state element to a *security class*: $public$, $free$, $active$, or $sealed$.

State elements that are outside of the stack—general-purpose memory used for globals and the heap, as well as the code region and globally shared registers—are always labeled $public$. We place security requirements on some $public$ elements for purposes of the WBCF property, and a given enforcement mechanism might restrict their access (e.g., by rendering code immutable), but for integrity and confidentiality purposes they are considered accessible at all times.

For a newly active function, every stack location that is available for use but not yet initialized is seen as $free$. From the perspective of the caller, the callee has no obligations regarding its use of free elements.

Arguments are marked $active$, meaning that their contents may be safely used. When a function allocates memory for its own stack frame, that memory will also be $active$. Then, on a call, $active$ elements that are not being used to communicate with the new callee will become $sealed$—reserved for an inactive principal, and expected to be unchanged when it becomes active again.

Instantiating the Framework: Conceptually, the following steps are needed to instantiate the framework to a specific machine and code style: (i) define the base machine semantics, including any hardware security enforcement features; (ii) identify the set of security-relevant operations and rules required by the code style; (iii) determine how to label machine instructions with security-relevant operations as appropriate; (iv) specify the form of observable events.

Threat Model and Limitations: When our properties are used to evaluate a system, the threat model will depend on the details of that system. However, there are some constraints that our design puts on any system. In particular, we must trust that the security-relevant operations have been correctly labeled. If a compiled function call is not marked as such,

then the caller's data might not be protected from the callee; conversely, marking too many operations as calls will simply cause otherwise safe programs to be rejected.

We do not assume that low-level code adheres to any fixed calling convention or implements any particular source-language constructs. Indeed, if the source language is C, then high-level programs might contain undefined behavior, in which case they might be compiled to arbitrary machine code. A given enforcement mechanism or target architecture might place additional constraints, particularly on the behavior of call and return sequences. For instance, extant implementations tend to assume implicitly that callee-saved registers have their values maintained by whichever compiler generated their code. [APT: I do not understand the last sentence. Implementations of what?] Our properties explicitly state this as a requirement, which could be enforced by a micro-policy, a well-behaved compiler, or other enforcement technique.

In general, it is impossible to distinguish buggy machine code from an attacker. In our examples we will identify one function or another as an attacker, but we do not require any static division between trusted and untrusted code, and we aim to protect even buggy code.

This is a strong threat model, but it does omit some important aspects of stack safety in real systems: in particular, it does not address concurrency. Hardware and timing attacks are also out of scope.

III. Properties by Example

In this section we introduce our security property definitions by means of small code examples, using a simplified set of security-relevant operations for calls, returns, and private allocations. Figure 1 gives C code and possible corresponding compiled 64-bit RISC-V code for a function `main`, which takes an argument `secret` and initializes a local variable `sensitive` to contain potentially sensitive data. Then `main` calls another function `f`, and afterward performs a test on `sensitive` to decide whether to output `secret`. Since `sensitive` is initialized to 0, the test should fail, and `main` should instead output the return value of `f`. Output is performed by writing to the special global `out`, and we assume that such writes are the only observable events in the system.

The C code is compiled using the standard RISC-V calling conventions [28]. In particular, the first function argument and the function return value are both passed in `a0`. Memory is byte-addressed and the stack grows towards lower addresses. We assume that `main` begins at address 0 and its callee `f` at address 100.

We now consider how `f` might misbehave and violate desirable stack safety properties associated with `main`. To put the violations in a security framework, suppose that `f` is actually an attacker seeking to leak `secret`. It might do so in a number of ways, shown as snippets of assembly code in Fig. 2. Leakage is most obviously viewed as a violation of `main`'s *confidentiality*. In Fig. 2a, `f` takes an offset from the stack pointer, accesses `secret`, and directly outputs it. But more

```
volatile int out;
void main(int secret) {
    int sensitive = 0;
    int res = f();
    if (sensitive == 42)
        out = secret;
    else
        out = res;
}
```

0:	`addi sp,sp,-20`	**alloc** $(-20, 20)$
4:	`sd ra,12(sp)`	
8:	`sw a0,8(sp)`	
12:	`sw zero,4(sp)`	
16:	`jal f,ra`	**call** ε
20:	`sw a0,0(sp)`	
24:	`lw a4,4(sp)`	
28:	`li a5,42`	
32:	`bne a4,a5,L1`	
36:	`lw a0,8(sp)`	
40:	`sw a0,out`	
44:	`j L2`	
L1, 48:	`lw a0,0(sp)`	
52:	`sw a0,out`	
L2, 56:	`ld ra,12(sp)`	
60:	`addi sp,sp,20`	**dealloc** $(0, 20)$
64:	`jalr ra`	**return**

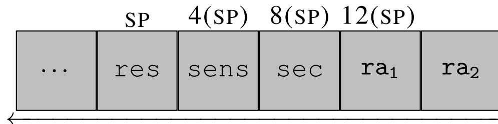


Fig. 1: Example: C and assembly code for `main`, and layout of its stack frame; the stack grows to the left.[APT: In this and later figures, can we lighten the blue backgrounds?]

subtly, even if somehow prevented from outputting `secret` directly, `f` can instead return that value so that `main` stores it to `out`, as in Fig. 2b. Beyond simply reading `secret`, the attacker might overwrite `sensitive` with 42, guaranteeing that `main` publishes its own secret unintentionally (Fig. 2c). Attacks of this kind do not violate `main`'s confidentiality, but rather its *integrity*. In Fig. 2d, the attacker arranges to return to the wrong instruction, thereby bypassing the check and publishing `secret` regardless, violating the program's *well-bracketed control flow* (WBCF). In Fig. 2e, a different attack violates WBCF, this time by returning to the correct program counter but with the wrong stack pointer.[2]

The security semantics for this program is based on the security-relevant events noted in the right columns of Figs. 1 and 2, namely execution of instructions that allocate or deallocate space (specified by an SP-relative offset and size), make a call (with a specified list of argument registers), or make a return.

[2] We pad some of the variants with `nop`s just so that all the snippets have the same length, which keeps the step numbering uniform in Fig. 3.

```
100:  lw a4,8(sp)
104:  sw a4,out
108:  li a0,1
112:  jalr ra          return
```

(a) Leaking secret directly

```
100:  lw a4,8(sp)
104:  mov a0,a4
108:  nop
112:  jalr ra          return
```

(b) Leaking secret indirectly

```
100:  li a5,42
104:  sw a5,4(sp)
108:  li a0,1
112:  jalr ra          return
```

(c) Attacking sensitive

```
100:  addi ra,ra,16
104:  nop
108:  nop
112:  jalr ra          return
```

(d) Attacking control flow

```
100:  addi sp,sp,8
104:  nop
108:  nop
112:  jalr ra          return
```

(e) Attacking stack pointer integrity

Fig. 2: Example: assembly code alternatives for f as an attacker.

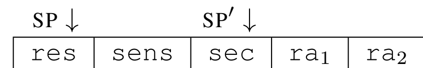
Our security semantics attaches a security context to the machine state, which consists of a view V and a stack σ of pending activations' views. Figure 3 shows how the security context evolves over the first few steps of the program. (The formal details of the security semantics are described in Section IV, and the context evolution rules are formalized in Fig. 7.) Execution begins at the start of main, where the program counter (PC) is zero, and with the stack pointer (SP) at address 1000. State transitions are numbered and labeled with a list of security operations, written $\downarrow \overline{\psi}$ between steps. We write ε for empty lists.

The initial view V_0 maps all stack addresses below SP to *free* and the remainder of memory to *public*. The sole used argument register, a0, is mapped to *active*; other caller-save registers are mapped to *free* and callee-save registers to *sealed*. Step 1 allocates a word each for secret, sensitive, and res, as well as two words for the return address. This has the effect of marking those bytes *active*. (We use $V[\![\cdot]\!]$ to denote updates to V.)

At step 5, the current principal's record is pushed onto the inactive list. The callee's view is updated from the caller's such that all *active* memory locations become *sealed*. (For now we assume no sharing of memory between activations; data is passed only through argument registers, which remain active. In the presence of memory sharing, some memory would remain active, too.) Function f does not take any arguments; if it did, any registers containing them would be mapped to *active*, while any non-argument, caller-saved registers are mapped to *free*. In the current example, only register a0 has a change in security class. All callee-save registers remain *sealed* for all calls, so if, in the example, we varied the assembly code for main so that sensitive were stored in a callee-save register (e.g. s0) rather than in memory, its security class would still be *sealed* at the entry to f. At step 9, f returns, and the topmost inactive view, that of main, is restored.

We now show how this security semantics can be used to define notions of confidentiality, integrity, and correct control flow in such a way that many classes of bad behavior, including the attacks in Fig. 2, are detected as security violations.

Well-bracketed Control Flow: To begin with, what if f returns to an unexpected place (i.e. PC $\neq 20$ or SP $\neq 980$)? We consider this to violate WBCF. WBCF is a relationship between call steps and their corresponding return steps: just after the return, the program counter should be at the next instruction following the call, and the stack pointer should be the same as it was before the call. Both of these are essential for security. In Fig. 2d, the attacker adds 16 to the return address and then returns; this bypasses the if-test in the code and outputs secret. In Fig. 2e, the attacker returns with SP$' = 988$ instead of the correct SP $= 980$. In this scenario, given the layout of main's frame,



main's attempt to read sensitive will instead read part of the return address, and its attempt to output res will instead output secret.

Before the call, the program counter is 16 and the stack pointer is 980. So we define a predicate on states that should hold just after the return: $\mathit{Ret}\ m \triangleq m[\text{PC}] = 20 \wedge m[\text{SP}] = 980$. We can identify the point just after the return (if a return occurs) as the first state in which the pending call stack is smaller than it was just after the call. WBCF requires that if m is the state at that point, then $\mathit{Ret}\ m$ holds. (This property is formalized in Table II, line 1).

Stack Integrity: Like WBCF, stack integrity defines a condition at the call that must hold upon return. This time the condition applies to all of the memory in the caller's frame. In Fig. 3 we see the lifecycle of an allocated frame: upon allocation, the view labels it *active*, and when a call is made, it instead becomes *sealed*. Intuitively, the integrity of main is preserved if, when control returns to it, any *sealed* elements are identical to when it made the call. Again, we need to know when a caller has been returned to, and we use the same mechanism of checking the depth of the call stack. In the case of the call from main to f, the *sealed* elements

PC	SP	Context
0	1000	V_0, ε
$1 \downarrow$ [**alloc** $(-20, 20)$]		
4	980	$V_1 = V_0[\![980..999 \mapsto \mathit{active}]\!], \varepsilon$
2-4 $\downarrow \varepsilon$		
16	980	V_1, ε
$5 \downarrow$ [**call** $100\ \varepsilon$]		
100	980	$V_2 = V_1[\![980..999 \mapsto \mathit{sealed}, \mathtt{a0} \mapsto \mathit{free}]\!], [V_1]$
6-8 $\downarrow \varepsilon$		
112	980	$V_2, [V_1]$
$9 \downarrow$ [**return**]		
20	980	V_1, ε

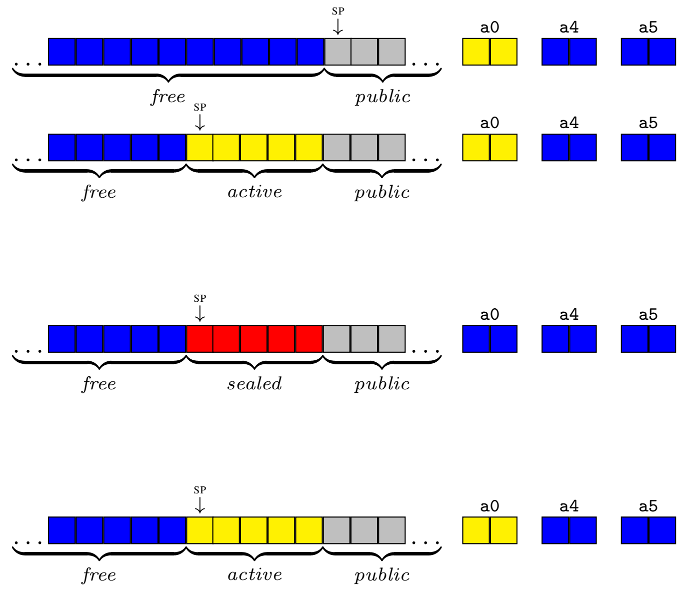


Fig. 3: Execution of example up through the return from `f`. In stack diagrams, addresses increase to the right, stack grows to the left, and boxes represent 4-byte words.

are the addresses 980 through 999 and callee-saved registers such as the stack pointer. Note that callee-saved registers often change during the call—but if the caller accesses them after the call, it should find them restored to their prior value.

While it would be simple to define integrity as "all sealed elements retain their values after the call," this would be stricter than necessary. Suppose that a callee overwrites some data of its caller, but the caller never accesses that data (or only does so after re-initializing it). This would be harmless, with the callee essentially using the caller's memory as scratch space, but the caller never seeing any change.

For a set of elements K, a pair of states m and n are K-*variants* if their values an only disagree on elements in K. We say that the elements of K are *irrelevant* in m if they can be replaced by arbitrary other values without changing the observable behavior of the machine. All other elements are *relevant*.[3]

We define *caller integrity* (CLRI) as the property that every relevant element that is *sealed* under the callee's view is restored to its original value at the return point. (This property is formalized in Table II, line 2).

In our example setting, the observation trace consists of the sequence of values written to `out`. The example in Fig. 2c modifies the value of `sensitive`, which is *sealed*. Figure 4 shows the state just after the call at step 5, assuming that `sec` is 5. [APT: But the figure seems to show several states. Unclear!] Similar to WBCF, we define *Int* as a predicate on states that holds if all relevant sealed addresses in m are the

[3]This story is slightly over-simplified. If an enforcement mechanism maintains additional state associated with elements, such as tags, we don't want that state to vary. Formal definitions of variants and relevance that incorporate this wrinkle are given in Section IV-D. [SNA: Might need to make this concept a touch more explicit in Section IV-D, or else promise a little less here.]

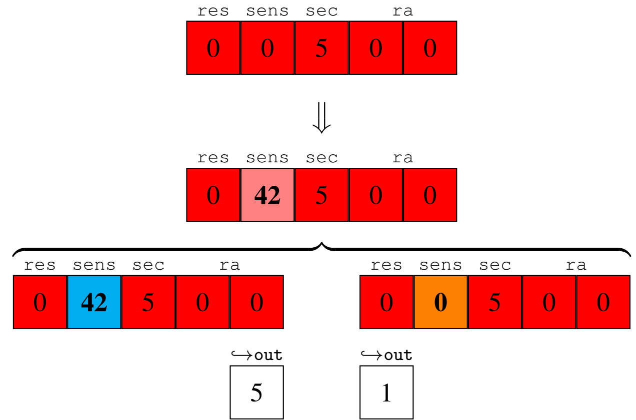


Fig. 4: Integrity Violation[APT: Unclear how these states are related. Extend caption? And what do the colors means?]

same as after step 5. We require that *Int* hold on the state following the matching return, which is reached by step 9. Here `sensitive` has obviously changed, but is it relevant? [APT: Rest of this paragraph is still not very clear, especially in (non-?)relation to the figure.] Consider a variant state in which `sensitive` has any other value, say 43. As execution continues after the return from the original state, it passes the `if`-test on `sensitive`, whereas the execution from the variant does not, resulting in differing outputs. Therefore `sensitive` *is* relevant, so *Int* does not hold, and integrity has indeed been violated.

Caller Confidentiality: We treat confidentiality as a form of non-interference as well: the confidentiality of a caller means that its callee's behavior is dependent only on publicly visible data, not the caller's private state. This also requires

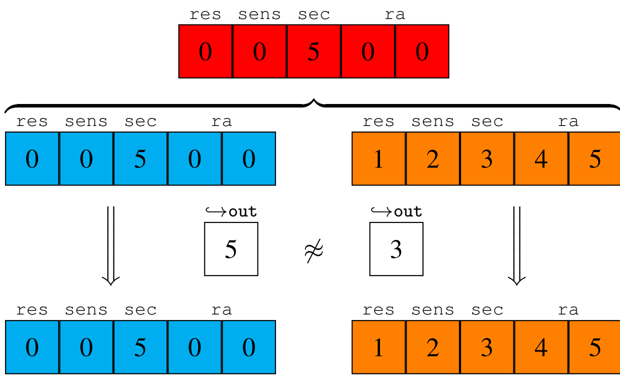


Fig. 5: Internal Confidentiality Violation

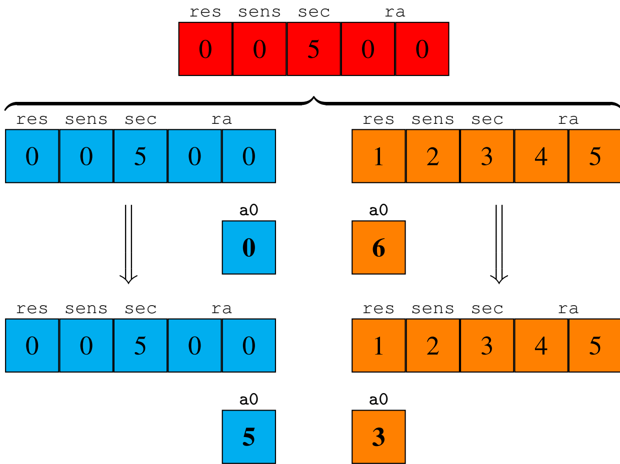


Fig. 6: Return-time Confidentiality Violation

that the callee initialize memory before reading it. As we saw in the examples, we must consider both the observable events that the callee produces during the call and the changes that the callee makes to the state that might affect the caller after the callee returns.

Consider the state after step 5, 5with the attacker code from Fig. 2a. We take a variant state over the set of elements that are *sealed* in V_2 (see Fig. 5). If we take a trace of execution from each state until it returns, the traces may differ, in this case outputting 5 (the original value of `secret`) and 4 [APT: 3???] (its value in the variant) respectively. This is a violation of *internal confidentiality* (formalized in Table II, line 3a).

But, in Fig. 2b, we also saw an attacker that exfiltrated the secret by reading it and then returning it, in a context where the caller would output the returned value. Figure 6 shows the behavior of the same variants under this attacker, but in this case, there is no output during the call. Instead the value of `secret` is extracted and placed in `a0`, the return value register. In this case, `a0` has changed during the call, *and* the return states do not agree on its value. We can therefore deduce that it carries some information derived from the original varied elements. By contrast, the sealed stack frame has not changed during the call, so the fact that it varies between the return states represents that the initial states disagreed, not a leak. [APT: Last sentence still quite mysterious.] Unless `a0` happens to be irrelevant to the caller, this example is a violation of what we term *return-time confidentiality* (formalized in Table II, line 3b). [SNA: Consider connections to robust declassification. Low priority.]

Structurally, return-time confidentiality resembles integrity, but now dealing with variants. We begin with a state immediately following a call, m. We consider an arbitrary variant state, n, which may vary any element that is *sealed* or *free*, i.e., any element that is not used legitimately to pass arguments. Caller confidentiality therefore can be thought of as the callee's insensitivity to elements in its initial state that are not part of the caller-callee interface.

We define a binary relation *Conf* on pairs of states, which holds on eventual return states m' and n' if all relevant elements are *uncorrupted* relative to m and n. An element is *corrupted* if it differs between m' and n', and it either changed between m and m' or between n and n'.

Finally, we define *caller confidentiality* (CLRC) as the combination of internal and return-time confidentiality (Table II, line 3).

The Callee's Perspective: We presented our initial example from the perspective of the caller, but a callee may also have privilege that its caller lacks, and which must be protected from the caller. Consider a function that makes a privileged system call to obtain a secret key, and uses that key to perform a specific task. An untrustworthy or erroneous caller might attempt to read the key out of the callee's memory after return, or to influence the callee to cause it to misuse the key itself!

Where the caller's confidentiality and integrity are concerned with protecting specific, identifiable state—the caller's stack frame—their callee equivalents are concerned with enforcing the expected interface between caller and callee. Communication between the principals should occur only through the state elements that are designated for the purpose: those labeled *public* and *active*.

Applying this intuition using our framework, *callee confidentiality* (CLEC) turns out to resemble CLRI, extended to every element that is not marked *active* or *public* at call-time. The callee's internal behavior is represented by those elements that change over the course of its execution, and which are not part of the interface with the caller. At return, those elements should become irrelevant to the subsequent behavior of the caller.

Similarly, in *callee integrity* (CLEI), only elements marked *active* or *public* at the call should influence the behavior of the callee. It may seem odd to call this integrity, as the callee does not have a private state. But an erroneous callee that performs a read-before-write within its stack frame, or which uses a non-argument register without initializing it, is vulnerable to its caller seeding those elements with values that will change

its behavior. The fact that well-behaved callees have integrity by definition is probably why callee integrity is not typically discussed.

IV. Formalization

We now give a formal description of our machine model, security semantics, and properties. Our definitions abstract over: (i) the details of the target machine architecture and ABI, (ii) the set of security-relevant operations and their effects on the security context, (iii) the set of observable events, and (iv) a notion of value compatibility.

A. Machine

The building blocks of a machine are *words* and *registers*. Words are ranged over by w and, when used as addresses, a, and are drawn from the set $\mathcal{W}$. Registers in the set $\mathcal{R}$ are ranged over by r, with the stack pointer given the special name SP; some registers may be classified as caller-saved (CLR) or callee-saved (CLE). Along with the program counter, PC, these are referred to as *state elements* k in the set $\mathcal{K} ::= \text{PC}|\mathcal{W}|\mathcal{R}$.

A *machine state* $m \in \mathcal{M}$ is a map from state elements to a set $\mathcal{V}$ of *values*. Each value v contains a *payload* word, written $|v|$. We write $m[k]$ to denote the value of m at k and $m[v]$ as shorthand for $m[|v|]$. Depending on the specific machine being modeled, values may also contain other information relevant to hardware enforcement (such as a tag). When constructing variants (see Section IV-D, this additional information should not be varied. To capture this idea, we assume a given *compatibility* equivalence relation $\sim$ on values, and lift it element-wise to states. Two values should be compatible if their non-payload information (e.g. their tag) is identical.

The machine has a step function $m \xrightarrow{\bar{\psi},e} m'$. Except for the annotations over the arrow, this function just encodes the usual ISA description of the machine's instruction set. The annotations serve to connect the machine's operation to our security setting: $\bar{\psi}$ is a list of security-relevant operations drawn from an assumed given set Ψ, and e is an (potentially silent) observable event; these are described further below.

B. Security semantics

The security semantics operates in parallel with the machine. Each state element (memory word or register) is given a *security class* $l \in \{public, active, sealed, free\}$. A *view* $V \in VIEW$ maps elements to security classes. For any security class l, we write $l(V)$ to denote the set of elements k such that $V\ k = l$.[APT: Did you ever use this? I've edited some places to do do.] The *initial view* V_0 maps all stack locations to *free*, all other locations to *public*, and registers based on which set they belong to: *sealed* for callee-saved, *free* for caller-saved except for those that contain arguments at the start of execution, which are *active*, and *public* otherwise.

A (security) *context* is a pair of the current activation's view and a list of views representing the call stack (pending inactive principals), ranged over by σ.

$$c \in C ::= VIEW \times list\ \ VIEW$$

$$range\ r\ \ off\ \ sz\ \ m \triangleq \{m[r] + i | off \leq i < off + sz\}$$

$$\frac{K = range\ \text{SP}\ \ off\ \ sz\ \ m \cap free(V) \qquad V' = V[\![a \mapsto active \mid a \in K]\!]}{Op\ m\ (\mathbf{alloc}\ off, sz)\ (V, \sigma) = (V', \sigma)}$$

$$\frac{K = range\ \text{SP}\ \ off\ \ sz\ \ m \cap active(V) \qquad V' = V[\![a \mapsto free \mid a \in K]\!]}{Op\ m\ (\mathbf{dealloc}\ off, sz)\ (V, \sigma) = (V', \sigma)}$$

$$\frac{V' = V[\![r \mapsto free | r \in CLR]\!][\![r \mapsto public | r \in \overline{r_{args}}]\!] \qquad V'' = V'[\![a \mapsto sealed | a \in active(V')]\!]}{Op\ m\ (\mathbf{call}\ a_{target}\ \overline{r_{args}})\ (V, \sigma) = (V'', V :: \sigma)}$$

$$\frac{}{Op\ m\ \mathbf{return}\ (_, (V, \sigma')) = (V, \sigma')}$$

Fig. 7: Basic Operations[APT: This was absolutely full of problems. Please triple-check my changes. And do we really need two-stage calculation in the call rule, or was that just to avoid type-setting issues?]

The initial context is $c_0 = (V_0, \varepsilon)$.

Section III describes informally how the security context evolves as the system performs security-relevant operations. Formally, we combine each machine state with a context to create a *combined state* $s = (m, c)$ and lift the transition to $\Longrightarrow$ on combined states. At each step, the context updates based on an assumed given function $Op : \mathcal{M} \to C \to \Psi \to C$. Since a single step might correspond to multiple operations, we apply Op as many times as needed, using *foldl*.

$$\frac{m \xrightarrow{\bar{\psi},e} m' \qquad foldl\ (Op\ m)\ c\ \bar{\psi} = c'}{(m, c) \xRightarrow{\bar{\psi},e} (m', c')}$$

A definition of Op is most convenient to present decomposed into rules for each operation. We have already seen the intuition behind the rules for **alloc**, **call**, and **ret**. For the machine described in the example, the Op rules would be those found in Fig. 7. Note that Op takes as its first argument the state *before* the step.

C. Events and Traces

We abstract over the events that can be observed in the system, assuming just a given set $EVENTS$ that contains at least the element τ, the silent event. Other events might represent certain function calls (i.e., system calls) or writes to special addresses representing memory-mapped regions. A *trace* is a nonempty, finite or infinite sequence of events, ranged over by $\mathcal{E}$. We use "$\cdot$" to represent "cons" for traces, reserving "::" for list-cons.

We are particularly interested in traces that end just after a function returns. We define these in terms of the depth d of the security context's call stack σ. We write $d \hookrightarrow s$ for the trace of execution from a state s up to the first point where the stack depth is smaller than d, defined coinductively by these rules:

$$\frac{|\sigma| < d}{d \hookrightarrow (m,(V,\sigma)) = \tau}$$

$$\frac{(m,(V,\sigma)) \xRightarrow{\bar{\psi},e} (m',c') \quad |\sigma| \geq d \quad d \hookrightarrow (m',c') = \mathcal{E}}{d \hookrightarrow (m,(V,\sigma)) = e \cdot \mathcal{E}}$$

When $d = 0$, the trace will always be infinite[APT: why? remind us about machine not halting (if that is the reason)?]; in this case we omit d and just write $\hookrightarrow s$.

Two event traces $\mathcal{E}_1$ and $\mathcal{E}_2$ are *similar*, written $\mathcal{E}_1 \eqsim \mathcal{E}_2$, if the sequence of non-silent events is the same. That is, we compare up to deletion of τ events. Note that this results in an infinite silent trace being similar to any trace. So, a trace that silently diverges due to a failstop will fulfill this property [APT: what property?] vacuously.

$$\frac{}{\mathcal{E} \eqsim \mathcal{E}} \qquad \frac{\mathcal{E}_1 \eqsim \mathcal{E}_2}{e \cdot \mathcal{E}_1 \eqsim e \cdot \mathcal{E}_2}$$

$$\frac{\mathcal{E}_1 \eqsim \mathcal{E}_2}{\tau \cdot \mathcal{E}_1 \eqsim \mathcal{E}_2} \qquad \frac{\mathcal{E}_1 \eqsim \mathcal{E}_2}{\mathcal{E}_1 \eqsim \tau \cdot \mathcal{E}_2}$$

D. Variants, corrupted sets, and "on-return" assertions

Two (compatible) states are variants with respect to a set of elements K if they agree on the value of every element not in K. Our notion of non-interference involves comparing the traces of such K-variants. We use this to define sets of irrelevant elements. Recall that $\sim$ is a policy-specific compatibility relation.

Definition 1. The *difference set* of two machine states m and m', written $\Delta(m,m')$, is the set of elements k such that $m[k] \neq m'[k]$.

Definition 2. Machine states m and n are *K-variants*, written $m \approx_K n$, if $m \sim n$ and $\Delta(m,n) \subseteq K$.

Definition 3. An element set K is *irrelevant* to state (m,c), written $(m,c) \parallel K$, if for all n such that $m \approx_K n$, $\hookrightarrow (m,c) \eqsim \hookrightarrow (n,c)$.

When comparing the behavior of variant states, we need a notion of how their differences have influenced them.

Definition 4. The *corrupted set* $\bar{\Diamond}(m,m',n,n')$ is the set $(\Delta(m,m') \cup \Delta(n,n')) \cap \Delta(m',n')$.

If we consider two execution sequences, one from m to m' and the other from n to n', then $\bar{\Diamond}(m,m',n,n')$ is the set of elements that change in one or both executions and end up with different values. Intuitively, this captures the effect of any differences between m and n, i.e., the set of values that are "corrupted" by those differences.

Our "on-return" assertions are defined using a second-order logical operator $d \uparrow P$, pronounced "P holds on return from depth d," where P is a predicate on machine states. This is a coinductive relation similar to "weak until" in temporal logic—it also holds if the program never returns from depth d.

$$\frac{|\sigma| < d \qquad P\ m}{(d \uparrow P)\ (m,(V,\sigma))}\ \textsc{Returned}$$

$$\frac{\begin{array}{c}|\sigma| \geq d \quad (d \uparrow P)\ (m',c') \\ (m,(V,\sigma)) \xRightarrow{\bar{\psi},e} (m',c')\end{array}}{(d \uparrow P)\ (m,(V,\sigma))}\ \textsc{Step}$$

Similarly, we give a analogous binary relation for use in confidentiality. We define $\Uparrow$ so that $(m,c)\ (d \Uparrow R)\ (m',c')$ holds if R holds on the first states that return from depth d after (m,c) and (m',c'), respectively. Once again, $\Uparrow$ is coinductive.

$$\frac{|\sigma_1| < d \qquad |\sigma_2| < d \qquad m_1\ R\ m_2}{(m_1,(V_1,\sigma_1))\ (d \Uparrow R)\ (m_2,(V_2,\sigma_2))}\ \textsc{Returned}$$

$$\frac{\begin{array}{c}|\sigma_1| \geq d \quad (m_1,(V_1,\sigma_1)) \xRightarrow{\bar{\psi},e} (m_1',c_1') \\ (m_1',c_1')\ (d \Uparrow R)\ (m_2,(V_2,\sigma_2))\end{array}}{(m_1,(V_1,\sigma_1))\ (d \Uparrow R)\ (m_2,(V_2,\sigma_2))}\ \textsc{Left}$$

$$\frac{\begin{array}{c}|\sigma_2| \geq d \quad (m_2,(V_2,\sigma_2)) \xRightarrow{\bar{\psi},e} (m_2',c_2') \\ (m_1,(V_1,\sigma_1))\ (d \Uparrow R)\ (m_2',c_2')\end{array}}{(m_1,(V_1,\sigma_1))\ (d \Uparrow R)\ (m_2,(V_2,\sigma_2))}\ \textsc{Right}$$

E. Properties

Finally, the core property definitions are given in Table II, arranged to show their commonalities and distinctions. Each definition gives a criterion quantified over states s that immediately follow call steps. If an execution includes a transition $s' \xRightarrow{\bar{\psi}} s$ where **call** $a\ \bar{r} \in \bar{\psi}$, then s is the target of a call. As a shorthand, we write that each property is defined by a criterion that must hold "for all call targets s," or, in the case of WBCF, "for all call steps $s \Longrightarrow s'$."

1. WBCF: Given a call step $(m,(V,\sigma)) \Longrightarrow (m',(V',\sigma'))$, we define the predicate *Ret* to hold on states m'' whose stack pointer matches that of m and whose program counter is at the next instruction. A system enjoys WBCF if, for every call transition, *Ret* holds just after the callee returns (i.e., the call stack shrinks).

2. CLRI: When the call target is $(m,(V,\sigma))$, we define the predicate *Int* to hold on states m' if any elements that are both sealed in V and in the difference set between m and m' are irrelevant. A system enjoys CLRI if, for every call, *Int* holds just after the corresponding return.

3. CLRC: When the call target is $(m,(V,\sigma))$, we begin by taking an arbitrary n that is a K-variant of m, where K is the set of sealed elements in V. We require that two clauses hold. On line 3a, the behavior of a trace from $(m,(V,\sigma))$ up to its return must match that of $(n,(V,\sigma))$. On line 3b, we define a relation *Conf* that relates states m' and n' if their corrupted set (relative to m and n) is irrelevant, and require that it hold just after the returns from the callees that start at $(m,(V,\sigma))$ and $(n,(V,\sigma))$. A system enjoys CLRC if both clauses hold for every call.

4. CLEC: We consider the callee's private behavior to be any changes that it makes to the state outside of legitimate channels—elements marked *active* or *public*. The remainder should be kept secret, which is to say, irrelevant to future execution. Similar to CLRI, given a call target $(m,(V,\sigma))$, we define a predicate *CConf* to hold on states m' if the difference set between m and m', excluding *active* or *public* locations,

#	Property	Definition	Scope
1	WBCF $\triangleq (\lvert\sigma'\rvert \uparrow \mathit{Ret})\ (m', (V', \sigma'))$	where $\mathit{Ret}\ m'' \triangleq m''[\textsc{sp}] = m[\textsc{sp}]$ $\wedge\, m''[\textsc{pc}] = m[\textsc{pc}] + 4$	for all calls $(m, (V, \sigma)) \Longrightarrow (m', (V', \sigma'))$
2	CLRI $\triangleq (\lvert\sigma\rvert \uparrow \mathit{Int})\ (m, (V, \sigma))$	where $\mathit{Int}\ m' \triangleq m' \parallel (\mathit{sealed}(V) \cap \Delta(m, m'))$	for all call targets $(m, (V, \sigma))$
3	CLRC $\triangleq \forall n$ s.t. $m \approx_K n$,	where $K = \mathit{sealed}(V)$	for all call targets $(m, (V, \sigma))$
3a	$\lvert\sigma\rvert \hookrightarrow (m, (V, \sigma)) \simeq \lvert\sigma\rvert \hookrightarrow (n, (V, \sigma))$		
3b	and $(m, (V, \sigma))\ (\lvert\sigma\rvert \Uparrow \mathit{Conf})\ (n, (V, \sigma))$	where $(m'\ \mathit{Conf}\ n') \triangleq m' \parallel \bar{\diamond}(m, n, m', n')$	
4	CLEC $\triangleq (\lvert\sigma\rvert \uparrow \mathit{CConf})\ (m, (V, \sigma))$	where $\mathit{CConf}\ m' \triangleq m' \parallel (\Delta(m, m') - K)$ where $K = \mathit{public}(V) \cup \mathit{active}(V)$	for all call targets $(m, (V, \sigma))$
5	CLEI $\triangleq \forall n$ s.t. $m \approx_K n$,	where $K = \mathcal{K} - (\mathit{public}(V) \cup \mathit{active}(V))$	for all call targets $(m, (V, \sigma))$
5a	$\lvert\sigma\rvert \hookrightarrow (m, (V, \sigma)) \simeq \lvert\sigma\rvert \hookrightarrow (n, (V, \sigma))$		
5b	and $(m, (V, \sigma))\ (\lvert\sigma\rvert \Uparrow \mathit{CInt})\ (n, (V, \sigma))$	where $(m'\ \mathit{CInt}\ n') \triangleq m' \parallel \bar{\diamond}(m, n, m', n')$	

TABLE II: Properties[APT: The hardwired +4 is a little sad, but oh well.]

is irrelevant. A system enjoys CLEC if, for every call, CConf holds just after the corresponding return.

5. CLEI: Callee integrity means that the caller does not influence the callee outside of legitimate channels. The caller's influence can be seen internally, or in corrupted data on return, just like the caller's secrets would be under CLRC. So, for a call target $(m, (V, \sigma))$, we take an arbitrary n that is a K-variant of m, where K is the set of elements that are not active or public. The remainder of the property is identical to CLRC.

V. Extended Code Features

The system we model in Sections III and IV is very simple, but our framework is designed to make it easy to add support for additional code features. To support argument passing on the stack, we just add new parameters to the existing security-relevant operations, and refine how they update the security context. The remainder of the properties do not change at all. To add tail-calls, we add and define a new operation, and since it is a kind of call, we add it to the definition of call targets. The rules for the extended security semantics are given in Fig. 8; the rules in Fig. 7 can be recaptured by instantiating **call** with $\overline{sa}$ as the empty set, and **alloc** with flag $\mathbf{f}$.

A. Sharing Stack Memory

In our examples, we have presented a vision of stack safety in which the interface between caller and callee is in the registers that pass arguments and return values. This is frequently not the case in a realistic setting. Arguments may be passed on the stack because there are too many to pass in registers, as variadic arguments, or because they are composite types that inherently have pass-by-reference semantics. The caller may also pass a stack-allocated object by reference in the C++ style, or take its address and pass it as a pointer.

We refine our call operation to make use of the information that we have about which stack memory locations contain arguments. The new annotation $\overline{sa}$ is a set of triples of a register, an offset from the value of that register, and a size. We first define the helpful set $\mathit{passed}\ \overline{sa}\ m$, then extend the call operation to keep all objects in passed marked as active and seal everything else (Fig. 8b).

Using this mechanism, a call-by-value argument passed on the stack at an SP-relative offset is specified by the triple $(\textsc{sp}, \mathit{off}, \mathit{sz})$. In this case, only the immediate callee gains access to the argument location. A C++-style call-by-reference argument passed in r is instead specified by $(r, 0, \mathit{sz})$, which allows the reference to be passed further down the stack.[APT: Last sentence is completely mysterious.]

[APT: This needs light revision to reflect fact that Capabilities section is now on same page, not much later!] If the address of an object is taken directly and passed as a pointer, we simply classify the object as "public" and give it no protection against access by other functions. We extend the **alloc** operation with a boolean flag, where $\mathbf{t}$ indicates that the allocation is public, and $\mathbf{f}$ that it is private. If space for multiple objects is allocated in a single step, that step can make multiple allocation operations, each labeled appropriately. Public objects are labeled public rather than active, so they are never sealed at a call (Fig. 8a). Providing more fine-grained control over sharing is desirable, but seems to require a considerably more complex model; Section V-C describes one approach, based on capabilities.

B. Tail Calls

The rule for a tail call is similar to that for a normal call. We do not push the caller's view onto the stack, but replace it outright. This means that a tail call does not increase the size of the call stack, and therefore for purposes of our properties, all tail calls will be considered to return simultaneously when the eventual **return** operation pops the top of the stack.

Since the caller will not be returned to, it does not need integrity, but it should still enjoy confidentiality. We set its frame to *free* rather than *sealed* to express this. In Table II, we replace "call targets" with "call or tail call targets" in CLRC, CLEC, and CLEI.

C. Provenance, Capabilities, and Protecting Objects

Lastly, what if we want to express a finer-grained notion of safety, in which stack objects are protected unless the function that owns them intentionally passes a pointer to them? This can be thought of as a *capability*-based notion of security. Capabilities are unforgeable tokens that grant access to a region of memory, typically corresponding to valid pointers to that region. As such, this capability safety relies on some preexisting notion of pointer validity, i.e. *pointer provenance*. Memarian et al.'s PVI [29] (provenance via integer) memory model is a good option: it annotates pointers with the identity of the object they first pointed to, and propagates the

$$range\ r\ off\ sz\ m \triangleq \{m[r]+i | off \leq i < off + sz\}$$

$$\frac{K = range\ \text{SP}\ off\ \ sz\ \ m \cap \{a \mid V\ \ a = free\} \quad V' = V[\![a \mapsto sealed \mid a \in K]\!]}{Op\ m\ (\textbf{alloc f}\ off, sz)\ (V, \sigma) = (V', \sigma)}$$

$$\frac{K = range\ \text{SP}\ off\ \ sz\ \ m \cap \{a \mid V\ \ a = free\} \quad V' = V[\![a \mapsto public \mid a \in K]\!]}{Op\ m\ (\textbf{alloc t}\ off, sz)\ (V, \sigma) = (V', \sigma)}$$

$$\frac{b = m[\text{SP}] + off \quad V' = V[\![a \mapsto free | b \leq a < b + sz \wedge V\ \ a = active]\!]}{Op\ m\ (\textbf{dealloc}\ off, sz)\ (V, \sigma) = (V', \sigma)}$$

(a) Memory Allocation

$$passed\ \overline{sa}\ m = \bigcup_{(r, off, sz) \in \overline{sa}} range\ \ r\ \ off\ \ sz\ \ m$$

$$\frac{V' = V[\![r \mapsto free | r \in CLR]\!][\![r \mapsto public | r \in \overline{r_{args}}]\!] \quad V'' = V'[\![a \mapsto sealed | V'\ \ a = active \wedge a \notin (passed\ \overline{sa}\ m)]\!]}{Op\ m\ (\textbf{call}\ a_{target}\ \overline{r_{args}}\ \overline{sa})\ (V, \sigma) = (V'', V :: \sigma)}$$

$$\frac{V' = V[\![r \mapsto free | r \in CLR]\!][\![r \mapsto public | r \in \overline{r_{args}}]\!] \quad V'' = V'[\![a \mapsto free | V'\ \ a = active \wedge a \notin (passed\ \overline{sa}\ m)]\!]}{Op\ m\ (\textbf{tailcall}\ a_{target}\ \overline{r_{args}}\ \overline{sa})\ (V, \sigma) = (V', \sigma)}$$

$$\frac{\sigma = (V, a_{ret}, a_{sp}) :: \sigma'}{Op\ m\ \textbf{return}\ (_, \sigma) = (V, \sigma')}$$

(b) Calls with Argument Passing on the Stack

Fig. 8: Operations supporting tail calls and argument passing on stack.[APT: This needs similar edits to the previous operations figure. In particular, same question about the V'' in call and tailcall.]

annotation when the pointer is copied and when operations are performed on it. This constitutes a substantial addition to the security context, which is why this enhancement is more speculative than the others, and we have not tested it.

We can model the provenance model as a trio of additional security-relevant operations: one which declares a register to contain a valid pointer, one which transmits the provenance of a pointer from one element to another, and one which clears the provenance (for instance, when a pointer is modified in place in a way that makes it invalid).

In addition to the normal call stack, our security context will carry a map ρ from elements to memory regions, represented as a base and a bound $c = (V, \sigma, \rho)$. Existing operations are extended to preserve the value of ρ, and the new operations work as follows[APT: Put these and tweaked call rule into a separate figure in the same style as the existing two]:

$$\frac{\psi = \textbf{promote}\ r_{dst}\ \ (r_{base}, off, sz) \quad \rho' = \rho[r_{dst} \mapsto range\ r_{base}\ \ off\ \ sz]}{Op\ m\ \psi\ (V, \sigma, \rho) = (V, \sigma, \rho')}\ \text{PROMOTE}$$

$$\frac{\psi = \textbf{propagate}\ k_{src}\ k_{dst} \quad \rho' = \rho[k_{dst} \mapsto \rho[k_{src}]]}{Op\ m\ \psi\ (V, \sigma, \rho) = (V, \sigma, \rho')}\ \text{PROPAGATE}$$

$$\frac{\psi = \textbf{clear}\ k}{Op\ m\ \psi\ (V, \sigma, \rho) = (V, \sigma, \rho[k \mapsto \emptyset])}\ \text{CLEAR}$$

We now have a notion of provenance, and must integrate it into the definition of stack safety. We essentially generalize the above notion of passing: we will consider a caller to have intentionally passed an object if that object is reachable by a capability that has been passed to the callee. This includes capabilities passed indirectly, by being stored in an object that is in turn passed. Formally, we call this set *capped*, and define it recursively:

$$capped\ \ K\ \ \rho \triangleq \bigcup_{k \in K} \{k\} \cup capped\ \ K'\ \ \rho \text{ where}$$
$$K' = \{k' | \rho[k] = (base, bound) \wedge base \leq k' < bound\}$$

We then tweak the call operation to seal only objects that are in *capped*, or the previously defined *passed*.

$$\frac{\psi = \textbf{call}\ a_{target}\ \overline{r_{args}}\ \overline{sa} \quad V' = V[\![r \mapsto free | r \in CLR]\!][\![r \mapsto public | r \in \overline{r_{args}}]\!] \quad K = \{a | V'\ \ a = active \wedge a \notin (passed\ \overline{sa}\ \ m) \cup (capped\ \overline{r_{args}}\ \ \rho)\} \quad V'' = V'[\![K \mapsto sealed]\!]}{Op\ m\ \psi\ (V, \sigma, \rho) = (V'', V :: \sigma, \rho)}$$

In the resulting property, once an object is sealed (because its capability has not been passed to a callee), subsequent nested calls can never unseal it. On the other hand, an object that is passed via a pointer may be passed on indefinitely.

VI. Enforcement

We implement and test two micro-policies inspired by Roessler and DeHon [1]: *Depth Isolation* (DI) and *Lazy Tagging and Clearing* (LTC). Their precise connection to Roessler and DeHon's work is discussed below. They share a common structure: each function activation is assigned a "color" n representing its identity. Stack locations belonging to that activation are tagged STACK n, and while the activation is running, the tag on the program counter (PC tag) is PC n. Stack locations not part of any activation are tagged UNUSED.

In DI, n always corresponds to the depth of the stack when the function is called. A function must initialize its entire frame upon entry in order to tag it, and then clear the frame before returning. During normal execution, the micro-policy rules only permit load and store operations when the target memory is tagged *with the same depth* as the current PC tag, or, for store operations, if the target memory is tagged UNUSED.

In LTC, a function neither initializes the frame at entry nor clears it at exist; instead, it simply sets each location's tag to the PC tag when that location is written. It does not check if those writes are legal! If the PC tag is PC n, then any stack location that recieves a store will be tagged STACK n. On a load, the micro-policy failstops if the source memory location is tagged UNUSED or STACK n for some n that doesn't match the PC tag.

To implement this discipline, *blessed instruction sequences* appear at the entry and exit of each function, which manipulate tags as just described while performing the usual tasks of saving/restoring the return address to/from the stack and adjusting the stack pointer. A blessed sequence uses further tags to guarantee that the full sequence executes from the beginning—no jumping into the middle.

Applicability to Roessler & DeHon [1]:: For Roessler and DeHon (henceforward *R&D*), Lazy Tagging and Lazy Clearing are both optimizations that can be applied to their Depth Isolation policy. Our version of LTC corresponds to Depth Isolation with both optimizations applied. How closely do our properties correspond to the specifications of their policies?

R&D differentiate between memory safety policies (without lazy optimization) and *data-flow integrity* policies (with lazy optimization). Our properties are phrased in terms of data flow, and we apply them to both optimized and non-optimized Depth Isolation. R&D do not attempt to define explicit formal properties, but they do list the behaviors that they expect their data-flow integrity policies to prevent, namely: reads from sealed objects (our CLRC), writes to sealed objects if they are later read (our CLRI), and reads from deallocated objects (our CLEC). They also note that Lazy Clearing prevents uninitialized reads, which corresponds roughly to our CLEI.

R&D note a flaw in Depth Isolation: because function activations are identified by depth, a dangling pointer into a stack frame might be usable when a new frame is allocated at the same depth. Our testing does not discover this flaw, because we do not test address-taken objects, but it discovers a related flaw under Lazy Tagging and Clearing that does not require an object's address to be taken. If an activation reads a location that was previously written by an earlier activation at the same depth, it will violate callee confidentiality. If that location was in a caller's frame, it also violates caller integrity and confidentiality.

R&D propose addressing the dangling-pointer issue by tracking both the depth of the current activation and the static identity of the active function. This would not eliminate all instances of this issue, but it would require the confidentiality-violating activation to be of the same function that wrote the data in the first place, which is a significantly higher bar. We propose instead tracking every activation uniquely, which should eliminate the issue entirely—and does in our tests.

Protecting Registers: R&D do not need to protect registers, since they include the compiler in their trusted computing base, but we target threat models that do not. In particular, CLRI requires callee-saved registers to be saved and restored properly. We extend DI and LTC so that callee-saved registers are also tagged with the color of the function that is using them. In DI they are tagged as part of the entry sequence, while in LTC they are tagged when a value is placed in them.

VII. Validation through Random Testing

There are several ways to evaluate whether an enforcement mechanism enforces the above stack safety properties. Ideally such validation would be done through formal proof over the semantics of the enforcement-augmented machine. However, while there are no fundamental barriers to producing such a proof, it would be considerable work to carry out for a full ISA like RISC-V and complex enforcement mechanisms like Roessler and DeHon's micro-policies. We therefore choose to systematically *test* their micro-policies. Our primary testing targets are the eager *Depth Isolation* and the *Lazy Per-Activation Tagging and Clearing* micro-policies.[APT: Reads awkwardly coming just after last section.]

We use a Coq specification of the RISC-V architecture [30], extend it with a runtime monitor implementing a stack safety micro-policy, and test it using QuickChick [26], a randomized property-based testing framework. QuickChick works by generating random programs, executing them, and checking that they fulfill our criteria.

Such testing is sound—it will not produce false positives—but necessarily incomplete. We might test a flawed policy but fail to generate a program that exploits the flaw. Additionally, detecting violations of noninterference-style properties is dependent on choosing appropriate variant states, so it is possible to generate a dangerous program but have it pass the test due to variant selection. We increase our confidence in our test coverage by *mutation testing*, in which we intentionally inject flaws into the policies and demonstrate that testing can find them.

A. Test Generation

To use QuickChick, we develop random test-case generators that produce an initial RISC-V machine state tagged appropriately for the micro-policy (see Section VI), including a code region containing a low-level program. They also produce the meta-information about how instructions in that program map to security-relevant operations, which would normally be provided by the compiler.

Our generators build on the work of Hriţcu et al. [31], [32], which introduced *generation by execution*, a technique that produces programs that lead to longer executions—and hopefully towards more interesting behaviors as a result. Each step of generation by execution takes a partially instantiated machine state and attempts to generate an instruction that makes sense locally (e.g., jumps go to a potentially valid code location, loads read from a potentially valid stack location). The generator repeats this process for an arbitrary number of steps, or until it reaches a point where the machine cannot step any more. Each time it generates a call or return, it places the appropriate policy tags on the relevant instruction(s) and records the operation.

We extend Hriţcu et al.'s technique with additional statefulness to avoid early failstops. For example, immediately after a call, we increase the probability of generating code that initializes any stack-allocated variables. To allow for potential attack vectors to manifest, the generator periodically relaxes those constraints and generates potentially ill-formed code, such as failing to initialize variables, writing outside

of the current stack frame, or attempting an ill-formed return sequence,

B. Property-based Testing

Once a test program is generated, QuickChick tests it against a property. A typical hyperproperty testing scheme might do this by generating a pair of initial variant states, executing them to completion, and comparing the results. We extend this procedure to handle the nested nature of confidentiality.

For our setup to naïvely test the confidentiality of every call, it would need to create a variant state at each call point, execute it until return, then generate a post-call variant based on any tainted values. The post-call variant would execute alongside the "primary" execution until the test is finished. This results in tracking a number of variant executions that is linear in the total number of calls!

For better performance, we instead maintain a single "post-call taint" execution that executes in parallel with the original. Everytime a function returns with elements containing tainted data[APT: I don't understand what that is], the test process sets those elements to random values in the tainted execution. So, at any given time, we need only simulate (1) the original execution, (2) the tainted execution, and (3) one variant execution for each call on the call stack. This approach makes testing longer executions substantially faster, at the cost of making it harder to identify which call is the source of a failure.

C. Mutation Testing

To ensure the effectiveness of testing against our formal properties, we use *mutation testing* [33] to inject errors (mutations) in a program that should cause the property of interest (here, stack safety) to fail, and ensure that the testing framework can find them. The bugs we use for our evaluation are either artificially generated by us (deliberately weakening the micro-policy in ways that we expect should break its guarantees), or actual bugs that we discovered through testing our implementation. We elaborate on some such bugs below.

For example, when loading from a stack location, *Depth Isolation* needs to enforce that the tag on the location being read is STACK n for some number n and that the tag of the current PC is PC n for the same depth n. We can relax that restriction by omitting the check (bug *LOAD_NO_CHECK_DI*). Similarly, when storing to a stack location, the correct micro-policy needs to ensure that the tag on the memory location is either UNUSED or has again the same depth as the current PC tag. Relaxing that constraint causes violations to the integrity property (bug *STORE_NO_CHECK*).

In additional intentional mutations, our testing catches errors in our own implementation of the enforcement mechanism, including one interesting bug where the initial function's frame included space allocated for its return address, but this uninitialized (and therefore UNUSED-tagged) space was treated as private data but left unprotected. We added this to our set of mutations as *HEADER_NO_INIT*.

Bug	Property Violated	MTTF (s)	Tests
LOAD_NO_CHECK_DI	Confidentiality	24.2	13.3
STORE_NO_CHECK	Integrity	26.9	26
HEADER_NO_INIT	Integrity	69.5	76.3
PER_DEPTH_TAG	Integrity	189.7	8342.5
LOAD_NO_CHECK_LT	Integrity	23.5	12.0
LOAD_NO_CHECK_LT	Confidentiality	19.2	695.5
STORE_NO_UPDATE	Integrity	70	80.6
STORE_NO_UPDATE	Confidentiality	4.9	88.5

TABLE III: MTTF for finding bugs in erroneous policy enforcement mechanisms[APT: what does the horizontal line mean?]

D. Results

The mean-time-to-failure (MTTF) and average number of tests for various bugs can be found in Table III, along with the average number of tests it took to find the failure. Experiments were run in a desktop machine equipped with i7-4790K CPU @ 4.0GHz with 32GB RAM.

For LTC, the original micro-policy, implemented as *PER_DEPTH_TAG*, fails in testing, in cases where data is leaked between sequential calls. To round out our mutation testing we also check *LOAD_NO_CHECK_LT*, equivalent to its counterpart in depth isolation, and a version where stores succeed but fails to propagate the PC tag, *STORE_NO_UPDATE*. It turns out that *PER_DEPTH_TAG* is a comparatively subtle bug, taking longer to catch.

Our properties allow us to identify an enforcement mechanism as not really stack safe, and to validate a possible fix.[APT: This paragraph comes as a bit of a non-sequitur here.]

VIII. Related Work

The centrality of the function abstraction and its security are behind the many software and hardware mechanisms proposed for its protection [1], [9]–[21]. Many enforcement techniques focus purely on WBCF; others combine this with some degree of memory protection, chiefly focusing on integrity. Roessler and DeHon's *Depth Isolation* and *Lazy Tagging and Clearing* [1] both offer protections corresponding to WBCF, CLRI, and CLRC, though they do not give a formal description of this. They are generally not concerned with protecting callees.

To our knowledge, the only other line of work that aims to rigorously characterize the security of the stack is the StkTokens-Cerise family of CHERI-enforced secure calling conventions [18]–[20]. The authors define stack safety as overlay semantics and related stack safety properties, phrased in terms of logical relations instead of trace properties. Originally, they describe "local state encapsulation" [19] in terms of integrity only (but it has confidentiality, equivalent to CLRI *and* CLRC). Their latest paper [20] was inspired by the properties presented in this paper to extend their formalism to include confidentiality. When checking if our properties applied to their old calling convention, they noted that it did not enforce CLEC, and made sure that their new version would in addition to building it into their formalism [34]. This demonstrates the benefit of our choice to explicitly state properties in security

terms: specifying security is hard, and when the spec takes the form of a "correct by construction" machine, it is easy to neglect a non-obvious security requirement.

In terms of direct feature comparison with Georges et al. [20] (the most recent work in the line), with the addition of confidentiality to their formalism, we are roughly at parity in terms of the expressiveness of our properties. We have additionally proposed callee-integrity, but it is probably the least practical of our properties. We extend our model to tailcalls, which they do not, and to the passing of pointers to stack objects. They discuss stack objects and the interaction between stack and heap, but their calling convention does not guarantee safety in the presence of pointer passing without additional checks. We test a limited degree of pointer passing, which does not guarantee memory safety for the passed pointer but which does not undermine the security of its frame, and we offer an untested formalism for memory-safe passing of pointers. On the other hand, their properties are validated by proof, while ours are only tested.

IX. Future Work

[SNA: Look into wasm compilation per reviewer 1. Low priority.]

We plan to test our properties against multiple enforcement mechanisms. The top priority is capability machines, namely CHERI [35], a modern architecture designed to provide efficient fine-grained memory protection and compartmentalization. We want to test the most recent work by Georges et al. [20], which is designed to enforce analogues of all of our properties except for CLEI.

It would also be interesting to test a software enforcement approach. Under a bounds checking discipline [10], all the pointers in a program are extended with some disjoint metadata used to gate memory accesses. These approaches enforce a form of *memory safety*, and we would therefore expect them to enforce CLRI and CLRC. They aim to enforce WBCF by cutting off attacks that involve memory-safety violations, but that may not be sufficient. Bounds checking approaches require substantial compiler cooperation. This is not a problem for our properties in general, but it is not very compatible with generation-by-execution of low-level code. A better choice might be to generate high-level code using a tool like CSmith [36], or prove the properties instead.

Several popular enforcement mechanisms are not designed to provide absolute guarantees of security. For example, stack canaries [9] and shadow stacks [14], [15] are chiefly hardening techniques: they increase the difficulty of some control-flow attacks on the stack, but cannot provide absolute guarantees on WBCF under a normal attacker model. Interestingly, these are lazy enforcement mechanisms, in that the attack may occur and be detected some time later, as long as it is detected before it can become dangerous. That would make our observation-based formalism a good fit for defining their security, if we could find a formal characterization of what they do acheive (perhaps in terms of a base machine with restricted addressing power).

We have preliminary work on extending our model to handle C++-style exceptions, which, like tailcalls, obey only a weakened version of WBCF. We are also exploring extensions to concurrency, starting with a model of statically allocated coroutines. These extensions will also require non-trivial testing effort. We also plan to test the model in Section V-C for arbitrary memory-safe pointer sharing.

Acknowledgements: We thank the reviewers for their comments, CHR Chhak and Allison Naaktgeboren for feedback during the writing process, and Aïna Linn Georges for her enthusiastic reception of our early work.[APT: would consider rephrasing that last.]

This work was supported by the National Science Foundation under Grant No. 2048499, Specifying and Verifying Secure Compilation of C Code to Tagged Hardware; and by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Excellence Strategy of the German Federal and State Governments, EXC 2092 CASA – 390781972.[APT: Benjamin?]Leo?

References

[1] N. Roessler and A. DeHon, "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, 2018, pp. 478–495. [Online]. Available: https://doi.org/10.1109/SP.2018.00066

[2] A. One, "Smashing the stack for fun and profit," *Phrack*, vol. 7, no. 49, November 1996. [Online]. Available: http://www.phrack.com/issues.html?issue=49&id=14

[3] MITRE Corporation, "Common weakness enumeration:2022 top 25 most dangerous software weaknesses," https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html, 2022.

[4] V. van der Veen, N. dutt-Sharma, L. Cavallaro, and H. Bos, "Memory errors: The past, the present, and the future," in *Research in Attacks, Intrusions, and Defenses - 15th International Symposium, RAID 2012, Amsterdam, The Netherlands, September 12-14, 2012. Proceedings*, ser. Lecture Notes in Computer Science, D. Balzarotti, S. J. Stolfo, and M. Cova, Eds., vol. 7462. Springer, 2012, pp. 86–106. [Online]. Available: https://doi.org/10.1007/978-3-642-33338-5_5

[5] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. IEEE Computer Society, 2013, pp. 48–62. [Online]. Available: https://doi.org/10.1109/SP.2013.13

[6] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 2016, pp. 969–986. [Online]. Available: https://doi.org/10.1109/SP.2016.62

[7] M. Miller, "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape," https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/, 2019.

[8] Chromium Projects, "Chromium security:memory safety," https://www.chromium.org/Home/chromium-security/memory-safety/.

[9] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, ser. SSYM'98. USA: USENIX Association, 1998, p. 5.

[10] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "SoftBound: highly compatible and complete spatial memory safety for C," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2009, pp. 245–258. [Online]. Available: http://repository.upenn.edu/cgi/viewcontent.cgi?article=1941&context=cis_reports

[11] ——, "CETS: compiler enforced temporal safety for C," in *9th International Symposium on Memory Management*. ACM, 2010, pp. 31–40. [Online]. Available: http://acg.cis.upenn.edu/papers/ismm10_cets.pdf

[12] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "HardBound: Architectural support for spatial safety of the C programming language," in *13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008, pp. 103–114. [Online]. Available: http://acg.cis.upenn.edu/papers/asplos08_hardbound.pdf

[13] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USA: USENIX Association, 2014, p. 147–163.

[14] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 555–566. [Online]. Available: https://doi.org/10.1145/2714576.2714635

[15] V. Shanbhogue, D. Gupta, and R. Sahita, "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*, ser. HASP '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3337167.3337175

[16] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. IEEE Press, 2014, p. 457–468.

[17] D. Chisnall, C. Rothwell, R. N. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann, "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*, ser. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 117–130. [Online]. Available: https://doi.org/10.1145/2694344.2694367

[18] L. Skorstengaard, D. Devriese, and L. Birkedal, "Reasoning about a machine with local capabilities: Provably safe stack and return pointer management," *ACM Trans. Program. Lang. Syst.*, vol. 42, no. 1, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3363519

[19] ——, "Stktokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities," *J. Funct. Program.*, vol. 31, p. e9, 2021. [Online]. Available: https://doi.org/10.1017/S095679682100006X

[20] A. L. Georges, A. Trieu, and L. Birkedal, "Le temps des cerises: Efficient temporal stack safety on capability machines using directed capabilities," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, apr 2022. [Online]. Available: https://doi.org/10.1145/3527318

[21] R. Gollapudi, G. Yuksek, D. Demicco, M. Cole, G. N. Kothari, R. H. Kulkarni, X. Zhang, K. Ghose, A. Prakash, and Z. Umrigar, "Control flow and pointer integrity enforcement in a secure tagged architecture," in *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2023, pp. 1780–1795. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00102

[22] A. Azevedo de Amorim, M. Dénès, N. Giannarakis, C. Hriţcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach, "Micro-policies: Formally verified, tag-based security monitors," in *36th IEEE Symposium on Security and Privacy (Oakland S&P)*. IEEE, May 2015.

[23] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[24] A. Azevedo de Amorim, C. Hritcu, and B. C. Pierce, "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*, ser. Lecture Notes in Computer Science, L. Bauer and R. Küsters, Eds., vol. 10804. Springer, 2018, pp. 79–105. [Online]. Available: https://doi.org/10.1007/978-3-319-89722-6_4

[25] M. Dénès, C. Hritcu, L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce, "QuickChick: Property-based testing for Coq (abstract)," in *VSL*, 2014. [Online]. Available: http://www.easychair.org/smart-program/VSL2014/index.html

[26] L. Lampropoulos and B. C. Pierce, *QuickChick: Property-Based Testing in Coq*, ser. Software Foundations series, volume 4. Electronic textbook, Aug. 2018, version 1.0. http://www.cis.upenn.edu/ bcpierce/sf.

[27] U. Dhawan, N. Vasilakis, R. Rubin, S. Chiricescu, J. M. Smith, T. F. Knight, B. C. Pierce, and A. DeHon, "PUMP – A Programmable Unit for Metadata Processing," in *Proceedings of the 3rd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://www.crash-safe.org/node/32

[28] R.-V. Consortium, "Risc-v calling conventions," https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/master/riscv-cc.adoc.

[29] K. Memarian, V. Gomes, B. Davis, S. Kell, A. Richardson, R. Watson, and P. Sewell, "Exploring c semantics and pointer provenance," *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1–32, 01 2019.

[30] T. Bourgeat, I. Clester, A. Erbsen, S. Gruetter, A. Wright, and A. Chlipala, "A multipurpose formal risc-v specification," *ArXiv*, vol. abs/2104.00762, 2021.

[31] C. Hriţcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. Azevedo de Amorim, and L. Lampropoulos, "Testing noninterference, quickly," in *18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Sep. 2013, 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. [Online]. Available: http://www.crash-safe.org/node/24

[32] C. Hriţcu, L. Lampropoulos, A. Spector-Zabusky, A. Azevedo de Amorim, M. Dénès, J. Hughes, B. C. Pierce, and D. Vytiniotis, "Testing noninterference, quickly," *J. Funct. Program.*, vol. 26, p. e4, 2016. [Online]. Available: https://doi.org/10.1017/S0956796816000058

[33] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011. [Online]. Available: http://crest.cs.ucl.ac.uk/fileadmin/crest/sebasepaper/JiaH10.pdf

[34] A. L. Georges, Personal communication.

[35] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. M. Norton, M. Roe, S. D. Son, and M. Vadera, "CHERI: A hybrid capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 20–37. [Online]. Available: https://doi.org/10.1109/SP.2015.9

[36] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, M. W. Hall and D. A. Padua, Eds. ACM, 2011, pp. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532