Enforcing Formal Security Properties

Andrew Bernard[1] Peter Lee

April 27, 2001

CMU-CS-01-121

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We define the formal semantics of expressive security-property language. The language distinguishes safe from unsafe programs and can be enforced systematically using proof-carrying code. The soundness of an enforcement algorithm is shown with respect to the language semantics.

[1]This material is based on work supported under under a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

This research was sponsored by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

Keywords: Security-Policy Specification, Security-Policy Enforcement, First-Order Logic, Temporal Logic, Verification-Condition Generation, Proof-Carrying Code

1 Introduction

A system for self-certified code[1] establishes that executing an untrusted, certified program will satisfy a security property. The *security property* characterizes which actions the host system permits the untrusted program to take; for example, a host might require that a program execute less than one hundred instructions. A *certified program* is a program packaged with a certification that, once checked, guarantees that the program will satisfy the security property.

Unfortunately, many systems for self-certified code are implemented without a security-property specification; in such cases, it is difficult to determine whether the implementation enforces the correct security property. For example, checking whether a customized Java virtual machine enforces a resource-bound security property amounts to verifying the entire virtual machine implementation. But even when systems for self-certified code interpret explicit security-property specifications, it is common for specification languages to be relatively inexpressive.

In this report, we present an algorithm that enforces a security property according to its encoding in a formal language. Given a certified program and a security-property specification, the algorithm determines whether program will satisfy the security property. We prove the soundness of this algorithm with respect to the formal semantics of the security-property language. The security-property language is expressive enough to encode specialized security properties (*e.g.*, resource bounds, confidentiality, access control). It is also defined mathematically so that characteristics of specific security properties can be demonstrated, in addition to general properties of the language itself.

The enforcement algorithm is based on proof-carrying code (PCC) [NL96, Nec97], a particularly flexible form of self-certified code. In particular, we extend the PCC symbolic evaluator to evaluate a security-property specification for each instruction of the untrusted program; a standard proof checker tests the agent's certification against the resulting verification condition. In this report, we do not address the important related question of how to automatically instrument a program so that it satisfies a security-property specification, nor do we provide an algorithm for automatically certifying programs—we intend to address these problems in the remainder of the first author's dissertation research.

1.1 Security Properties

Let an *execution* be a state sequence of some system: for example, the trace of a single program run. Following Alpern and Schneider [Sch99, AS85, AS86], a *security policy* is a predicate on sets of executions. Each program has a set of possible executions. A program *satisfies* a security policy if the security policy holds for its execution set; a program *violates* a security policy if it does not

[1] We intend the term "self certified" to exclude technologies based on a trusted third party, such as "signed applets".

satisfy the security policy. Security policies allow us to characterize prescribed and proscribed behavior for an untrusted program.

A *security property*, on the other hand, is a predicate on executions—all security properties can be regarded as security policies. A program satisfies a security property if the property holds for each of its possible executions; equivalently, a program satisfies a security property if its execution set is a subset of the executions for which the property holds.

Some security properties are safety properties, others are liveness properties, and some are neither. Informally, a *safety property* asserts that a specific "bad thing" does not occur in an execution: if a safety property holds for an execution, then it holds for each prefix of the execution. A *liveness property* asserts that a specific "good thing" will occur: if a liveness property does not hold for an execution, then the execution is a prefix of an execution for which the property holds. For example, "each lock can be acquired only once" is a safety property, while "all acquired locks must be released" is a liveness property. Alpern and Schneider [AS85, Sch87] showed that all security properties are the conjunction of a safety property and a liveness property. After this point, we will encounter only security properties in this report.

1.2 Self-Certified Code

Let an *agent* be a program that performs an operation on behalf of a system. In this report we will focus on untrusted agents that a trusted system executes; the user of the combined system determines what is trusted. For example, a web browser is a trusted system that downloads an "applet" to extend its own functionality; the applet is an untrusted agent[2]. Agents should not be trusted because they can do harmful things to the system.

A system for *self-certified code* establishes that an untrusted, certified agent will not violate a security property. The agent is packaged with annotations that relate the program to the security property. The annotations comprise a *certification* that the agent satisfies the security property. The host checks the agent and certification without external trust relationships: in particular, the host does not trust the provider of the agent. The certification is encoded in a formal language that demonstrates the acceptability of the agent after a program analysis. Because precise program analysis is not decidable, the checker is conservative in that it will only accept certified programs, even if an uncertified program would not violate the security property during an actual execution.

An *enforcement mechanism* prevents an agent from violating a security property. Enforcement can be accomplished by certification or by other means, such as run-time monitoring or program instrumentation. For self-certified code, the enforcement mechanism checks the agent and its certification. An enforcement mechanism is *sound* with respect to a security property if it rejects all agents that violate the security property. An enforcement mechanism is *complete* with

[2] We presume a cautious user who does not trust downloaded applets.

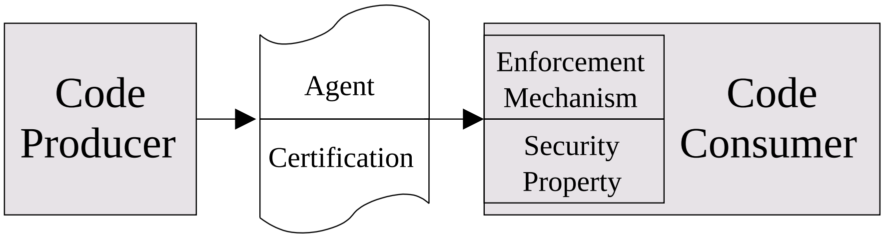

Figure 1: Self-Certified Code

respect to a security property if it accepts all agents that satisfy the security property. An enforcement mechanism *enforces* the security property for which it is both sound and complete.

Following the usual terminology [NL98b], a *code producer* transmits a certified agent to a *code consumer*. The code consumer wants to execute the agent, but does not trust that it satisfies a security property: a trusted enforcement mechanism therefore checks the certified agent. The code producer is typically a software developer who arranges that the agent will satisfy the security property and that the enforcement mechanism will accept it. The certification is typically constructed for the agent by an automatic *certification mechanism*, such as a certifying compiler. For example, a code producer might publish an applet on a web site that users download into their web browsers. Figure 1 illustrates this process; in this figure, the security property is coupled to the enforcement mechanism.

Note that self-certified code does not presume a trust relationship between code producer and code consumer, or with any third party—this scheme contrasts with cryptographic code certification (*e.g.*, signed applets [GMPS97]), which presumes that the code consumer trusts the code producer and only doubts the *authenticity* of the agent. Self-certified code is compatible with cryptographic certification (*e.g.*, the signature might attest that the agent is correct as well as harmless), but self-certified code protects the code consumer from defective agents in addition to malicious ones. Cryptographic certification requires the code consumer to trust the ability of the code producer, in addition to his or her intentions.

Kozen [Koz99] contains further discussion of self-certified code.

1.3 Approaches to Security-Property Specification

1.3.1 Proof-Carrying Code

Proof-carrying code (PCC) [Nec97, Nec98, NL96, NL98b, NL98a] is form of self-certified code in which the code producer packages the agent with annotations and a formal proof[3] that together demonstrate that it satisfies a specific security property. The code producer constructs the proof and encodes it in a formal logic: the enforcement mechanism checks that the proof is valid, and that it matches the agent and the security property. PCC deliberately places a heavier burden on the code producer than on the code consumer (proof checking is usually easier than proof discovery). PCC enables the code consumer to enforce security properties that are not decidable: the code producer must combine agents and proof generators such that valid proofs are possible. The code producer can even construct proofs manually, but this approach is feasible only for small programs.

Typical PCC implementations invoke a verification-condition (VC) generator [Kin71] to derive a proposition from the agent and a built-in security property. The code producer is obliged to show that the VC is valid under a set of axioms and inference rules. The code consumer need not trust the code producer, because the enforcement mechanism independently validates the proof.

Necula [Nec98] defines a security property according to the valid transitions of a *safe interpreter* for an abstract assembly language. The VC generator is based on the operational semantics of the safe interpreter, and he proves its soundness in his dissertation. This approach anticipates limited parameterization, because the preconditions of dangerous instructions are uninterpreted predicates on states. The corresponding predicate symbols are axiomatized to encode security properties for instruction safety and memory safety. Necula developed sample safety properties for proscribing memory access, for sand-boxing memory, and for abstract types.

1.3.2 Typed Assembly Language

The enforcement mechanism for *typed assembly language* (TAL) [MWCG98, MCG+99, CM99] is a type checker that does not accept programs that violate a security property; type annotations accompany agent instructions. A TAL compiler translates a well-typed source program into a well-typed assembly program. TAL has a potential performance advantage over PCC because safety proofs are not present; however, a TAL type checker must reconstruct type derivations from type annotations. Because PCC transmits complete type derivations, it is practical even when reconstruction is prohibitively expensive.

The original TAL [MWCG98] defines a security property implicitly by the type system of an assembly language. Walker [Wal99] developed a TAL type system based on an arbitrary security automaton; Alpern and Schneider [AS86, Sch99] invented security automata to encode security properties as formal automata. This version of TAL is novel because it separates the security property from the enforcement mechanism, and because security automata are not tailored to the type checker.

[3] Under the most general definition of PCC, the proof is any term of a formal system: thus, typed assembly language could be considered PCC, if we regard type annotations as summary type derivations. We prefer to identify this broader category as *self-certified code* and reserve PCC for systems based on proof terms of formal logics.

Crary and Weirich [CW00] developed a TAL type system that enforces resource bounds. The compiler for this type system is automatic, but it must be given a resource-bound annotation for each function. Crary, Walker, and Morrisett [CWM99] developed a TAL type system to enforce security properties based on a capability [DvH66] calculus. This calculus can ensure the safety of explicit deallocation. This enables an enforcement mechanism without a trusted garbage collector—the inclusion of a garbage collector in the trusted computing base is a drawback of many current enforcement mechanisms.

1.3.3 Safe Interpreters

For a *safe interpreter*, the agent language ensures that no security-property violations can occur. On the one hand, the enforcement mechanism can check the agent dynamically, in which case each operation is tested during execution; on the other hand, it can check the agent statically, in which case all tests occur before execution. It is common to see a hybrid of both techniques, because static enforcement is typically both more efficient and less precise than dynamic enforcement.

The Java virtual machine (JVM) [LY99] is a well-known enforcement mechanism for Java class files, which we consider to be self-certified code. Java class files contain instructions in the Java byte code language; the *byte code verifier* checks Java byte code statically. The JVM specification [LY99] documents the type-safety property of the byte-code verifier, in addition to other run-time security checks; this specification thus fixes the security property for Java byte code. Determining precisely what security property the JVM enforces is a challenge, because of its informal prose specification. Original implementations of the JVM interpreted byte codes directly, but modern implementations translate byte code into machine code.

The Java Language Specification [GJS96] documents the Java Security Manager, a trusted system that enforces access control. Permissions determine the operations that a process can perform. The JVM manages permissions transparently; although this approach is convenient in some respects, it restricts optimizing compilers because of stack introspection [WBDF97].

The security property of the Java Development Kit (JDK) 1.2 Security Model [GMPS97] is partially specified through configuration files. A *policy file* specifies which permissions an agent receives based on predefined attributes (*e.g.*, its origin or digital signature). Other researchers (*e.g.*, PoET [ES00], J-Kernel [HCC$^+$97], Naccio [ET99]) have developed extensions for more expressive security properties.

1.3.4 Software Fault Isolation

Software fault isolation (SFI) [WLAG93, ALLW96] instruments the agent so that it cannot violate a built-in security property. We do not consider SFI self-certified code, because it does not use a certification. SFI enforces a memory safety property that the instrumentation tool implicitly defines. SFI is fully automatic because it can instrument any agent, regardless of code producer. Unfortunately, SFI relies on run-time checks that entail run-time overhead and preclude fine-grain confidentiality properties [ML97].

Security automata SFI implementation (SASI) is an SFI-based tool developed by Erlingsson and Schneider [ES99, ES00] for enforcing security properties encoded in a security-automata language. Like security automata for TAL, we consider SASI an important contribution, because it disentangles the security property from the enforcement mechanism. The dissimilarities between these two mechanisms suggest that security automata are a universal security-property representation.

1.4 Limitations of Current Approaches

Unfortunately, enforcement mechanisms often determine security properties, rather than vice-versa. Such security properties (*e.g.*, SFI [WLAG93]) are difficult to document independently; witness attempts to formalize the Java byte code verifier [SA99, FM98, O'C99]. In the absence of precise definitions, it is impossible to establish rigorously that a security property prohibits malicious behavior.

PCC and TAL are based on formal models, but it is often impossible to change a security property without modifying the enforcement mechanism. For example, Necula [Nec98] parameterizes his PCC by predicates for a memory safety property, but a resource bound or confidentiality property requires a change to the code of the implementation.

Because of this rigidity, any work put into verifying an enforcement mechanism is lost when the security property is changed, because changes may introduce bugs that are not in the formal model. Additionally, enforcing a set of security properties requires a corresponding set of distinct implementations, each of which must be verified. Adding enforcement mechanisms increases the size of the trusted computing base and makes the entire system harder to trust.

Consider, for example, a personal digital assistant (PDA) that downloads agents for new functionality. A trusted enforcement mechanism for self-certified code checks the agents. Two agents are of interest:

- The *alarm clock* runs continuously, but only for brief intervals. It updates a display each second and when a previously scheduled time arrives, it emits a sound.

- The *synchronizer* runs to completion when the user activates it. The synchronizer ensures that the PDA database is consistent with the user's desktop database.

Figure 2 contains a diagram for this design. Each agent must satisfy a distinct set of security properties to be safe to run. A *memory-safety* security property protects the operating system and libraries from corruption by defective or malicious agents. Additional *resource-bound* security properties limit the system resources consumed by agents. The memory-safety security property is

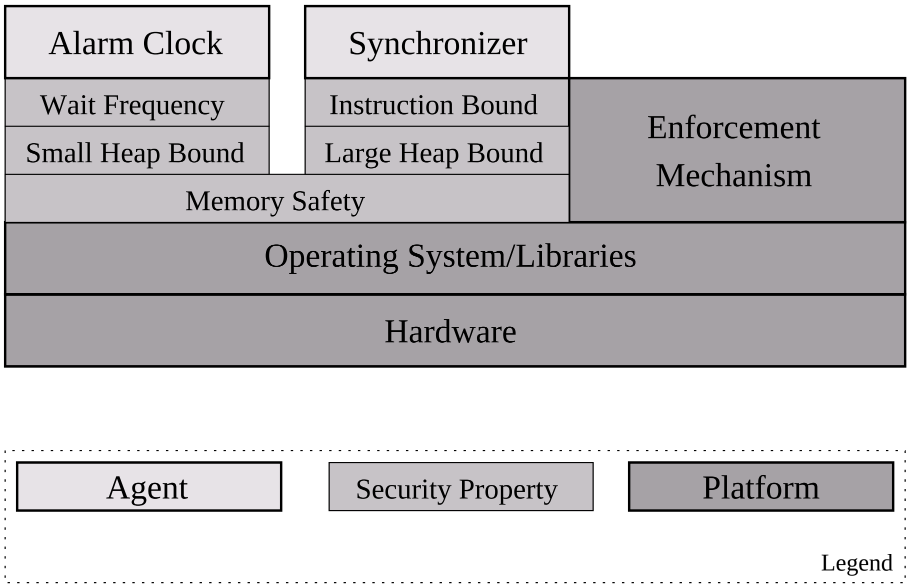

Figure 2: Safe PDA

common to all agents because it preserves the basic integrity of the system. The resource-bound security properties, however, are tailored to a specific agent.

For our example, the alarm clock needs little memory to run, but runs continuously for an unlimited period of time. However, it is usually in a waiting state between clock ticks. We thus assign the alarm clock the *wait-frequency* security property that limits it to a small number of instructions before invoking the `wait` system call. The *small-heap-bound* security property limits the alarm clock to only a small amount of dynamic memory.

However, we assign the synchronizer a different set of resource-bound security properties. The *instruction-bound* security property requires the agent to terminate after executing a number of instructions proportional to the size of the PDA database. The *large-heap-bound* security property limits the synchronizer to a large amount of dynamic memory. Because the synchronizer will terminate after a fixed time, we know that its dynamic memory will be released soon.

The usual realization of this design would include an enforcement mechanism for each security property. This approach has drawbacks, however, because all enforcement mechanisms are in the trusted computing base. In addition, implementations are relatively difficult to understand and change, and also tend to depart from a specification over time. The system designer is thus interested in developing *security properties* as opposed to *enforcement mechanisms*; ideally, the enforcement mechanism is derived automatically from the security property.

1.5 Goals for a Security-Property Language

To address the problems uncovered in Section 1.4, we have developed a *security-property language*. A security-property language encodes security properties in a concrete notation—each "program" of this language denotes a set of executions. For any given concrete security property, we want to do the following:

- Understand it in terms of an abstract model
- Reason about it with respect to the language semantics
- Enforce it with a generic implementation
- Select (or unselect) it based on a particular agent
- Modify it independently of an implementation

The security-property language thus has the following goals:

Abstraction The language should be defined mathematically so that it specifies precisely what actions a security property permits. A developer should not need detailed knowledge of an enforcement mechanism.

Abstraction is a benefit because the language semantics can be defined without reference to an enforcement mechanism. Any enforcement mechanism that implements the language semantics will enforce the correct execution set, so a developer need check only that the security property has the desired characteristics. Most of the work put into verifying an enforcement mechanism can thus be leveraged by many security properties. Figure 3 illustrates this point: the language semantics is the common interface between reasoning about security properties and reasoning about the enforcement mechanism.

In this report, we show the soundness of a particular enforcement mechanism: to ensure that this mechanism enforces a given security property, one need show only that the security property has the desired characteristics. For example, for an instruction-bound security property, show that all executions permitted by the security property do not exceed the intended length.

Expressiveness The security-property language should encompass specialized security properties (*e.g.*, beyond programming language type safety). For example,

Resource bounds A resource bound limits consumption of system resources (*e.g.*, processor, memory). It is important for critical resources (*e.g.*, mutexes, input–output devices) to be released promptly once acquired. Other applications (*e.g.*, networking) must limit the rate at which a resource is used.

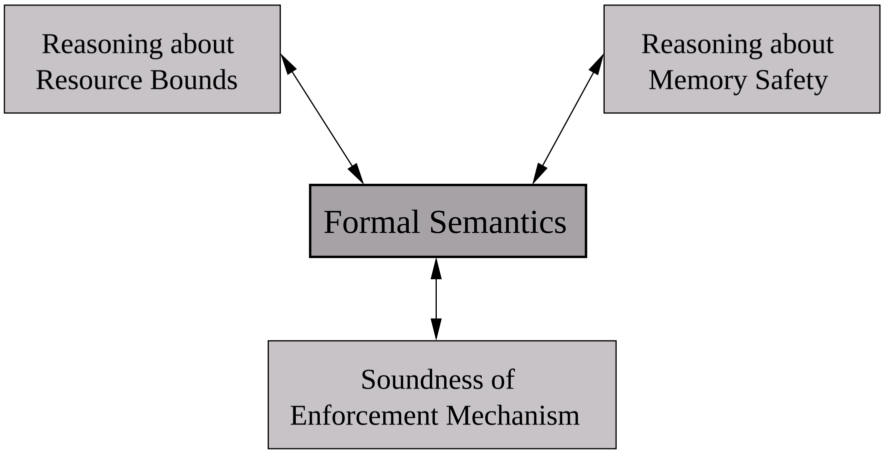

Figure 3: Localized Reasoning

Protection Operating system protection mechanisms prevent access to unauthorized resources [SG98, Lam71, WCC+74]. The JDK 1.2 Security Model [GMPS97] specifies an access control mechanism; we think that it is possible to encode the JDK 1.2 Security Model in our security-property language.

Confidentiality A *confidentiality property* [DD77, ML97] is a conservative approximation of an information flow policy that partitions the agent into distinct security classes. A communication channel is classified by the highest security class to which a message may belong. An enforcement mechanism prevents higher-security information from flowing to lower-security areas.

Modularity A modular security-property language supports independently developed security properties, and enforces them without interference. Specifically, if several security properties are enforced in combination, then the permitted executions should be the intersection of the independent execution sets. Thus, the soundness of an individual security property should not depend on whether it is enforced in combination with other security properties. Schneider [Sch99] identifies similar goals for security automata.

1.6 Our Approach

In this report, we present a specification for a security-property language that meets the goals identified in Section 1.5. The formal language semantics assigns an execution set to a security property. The semantics is defined in relation to a formal model of a RISC machine. We define formal models for the lan-

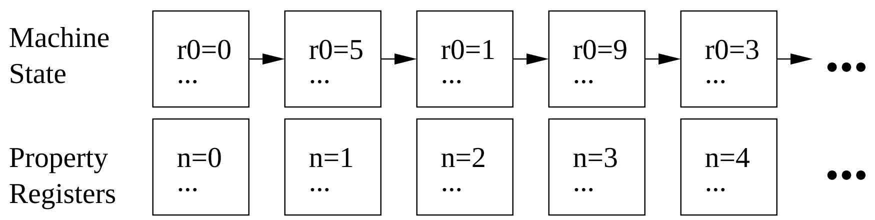

Figure 4: Instruction-Bound Property Register

guage and machine to enable mathematical reasoning about algorithms and security properties. Rigor is important for security because attackers will exploit enforcement-mechanism weaknesses—it is not sufficient to fix bugs as they are discovered.

We start from the operational semantics of an abstract RISC machine that simulates a real processor. The *trace semantics* of the machine defines the possible executions of a program based on initial conditions. The trace semantics enables us to compare the execution set of a program with the execution set of a security property. For example, returning to Section 1.4, we would show that the executions of the *alarm clock* agent are a subset of the *wait frequency* security property. We also model the system executing the agent to expose their interaction—in particular, we elide the actions of the host system from agent executions. The host system is defined by the agent procedures it calls, along with the entry points it exports to the agent.

We encode component specifications of the security-property language in a first-order logic; we selected this logic because it is simple to write specifications in, and because it is proven for self-certified code. These specifications can encode properties of individual machine states; to encode properties of executions, we enhance the machine model with *property registers* that record relevant properties of previous states. Property registers thus summarize the execution history prior to the current state. A specification applies to the latest state of an execution; to encode properties of earlier states, we refer to property registers. For example, to encode the *instruction-bound* security property from Section 1.4, we define a property register `n` that is initially zero and is incremented as each instruction is executed (see Figure 4). We check an execution by comparing `n` with the instruction bound. Note that a property register can be assigned any expression from the logic; thus, given a sufficiently expressive logic, we can encode an entire execution history, and thereby represent any safety property.

Property registers provide an intuitive model for encoding specifications for executions. We select the attributes of interest for a given security property and encode them directly. The imperative security-property language matches the imperative machine. Concrete security properties resemble programs that execute at every machine cycle. The imperative language model also suggests

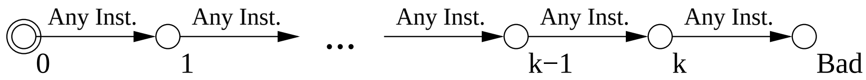

Figure 5: Instruction-Bound Security Automaton

a strategy for program instrumentation: reserve memory locations for property registers and update them according to the security property.

The property registers are effectively the current state of a security automaton [AS86, Sch99]. A security automaton is a state machine that responds to the actions of a target system; it enters a "bad" state when the target violates its security property. In fact, concrete security properties can be interpreted as security automata. Security automata are an attractive representation because they are enforceable with different mechanisms, and because they encompass all safety properties [Sch99]. For example, to represent the instruction-bound security property, we construct a sequence of automaton states of the same length as the bound k (see Figure 5). The automaton transitions to a successor state after executing an instruction; the successor of the kth state is the bad state.

We interpret security properties using a special symbolic evaluator; together with a proof checker, this is the standard enforcement mechanism for PCC. The symbolic evaluator interprets the program with symbolic expressions substituted for register values; a VC generator takes the output of the symbolic evaluator to a proposition that is valid only if the program satisfies the security property. We chose to use a VC generator because it is proven for PCC, and because symbolic evaluators are suited to our security-property language. Because the symbolic evaluator simulates each program instruction, we can simulate the security property by evaluating it in conjunction with the program simulation.

We chose to use PCC because it has a small trusted computing base, and because it can enforce complex security properties potentially without any runtime overhead. Additionally, because the PCC proof checker does not require an algorithm for discovering valid derivations, we need not constrain our security-property language.

1.7 This Report

The purpose of this report is to show the soundness of an enforcement algorithm for an expressive security-property language. Given a program and a security property, the algorithm computes a verification condition that is valid only if the program does not violate the security property. For PCC, an agent certification constitutes a proof of the VC.

In the body of this report (Section 2 through Section 4), we stratify each section into independent subsections that build only on preceding material. This strategy is intended to make the individual subsections easier to comprehend.

The soundness of the VC generator is shown with respect to an abstract machine model that we define in Section 2; the machine model derives executions from its operational semantics. Section 2 also contains a formal definition of our security-property language, including its operational semantics. In Section 3, we specify a symbolic evaluator and a VC generator; the symbolic evaluator intentionally resembles the operational semantics of the machine and the security-property language.

The remaining section is devoted to proving the soundness of the VC generator. The soundness proof shows that if the VC is valid, then all executions of the program are executions of the security property (in practice, we use a standard proof checker to show that the VC is valid). The main body of the proof is a series of lemmas that appear with supporting definitions in Section 4; these appear in "bottom-up" order so that each proof is based only on proven lemmas. Purely technical lemmas are relegated to the end of Section 4.

Appendix A contains examples of security properties written in our security-property language. Appendix B contains a type system that ensures the internal consistency of security properties, while Appendix C contains the semantics of pattern matching and derived forms (these are deferred because they are straightforward). Finally, in Appendix D, we collect the notation used in the report—it may help to refer to this section as new notation is encountered.

2 Semantics

The purpose of this section is to formalize the executions of an agent and a security property so that we can relate them in the soundness proof. We can also use the semantics to demonstrate that a given security property reflects our intended interpretation. Security properties are encoded in the language *Palladium*, which we introduce in Section 2.3.

We first define a simple machine model and show how program executions are derived from this model. To derive security-property executions, we extend the basic model to attach additional information to machine states; from extended executions, we erase the extra information to recover the executions of a security property. First, we introduce states and executions.

Based on Schneider [Sch99], an *execution* is a sequence of system states. Given states s_j: $j \geq 0$, an execution σ is

$$s_0 \rightarrow s_1 \rightarrow s_2 \ldots$$

A *step* is a transition from a state to its successor. Given states s and s', a step A is

$$s \rightarrow s'$$

We use states for a variant of Necula's *safe assembly language* [Nec98]:

$$\langle i, \rho \rangle$$

A state is a pair where i is the instruction counter, and ρ is the register environment. The register environment maps registers to their machine-word values. Memory is represented by a single register mapping words to words.

$$
\begin{array}{lrl}
\text{Instructions} & I ::= & \dot{r} \leftarrow n \mid \dot{r}_1 \leftarrow \dot{r}_2 \ \mathit{eop} \ \dot{r}_3 \mid \mathtt{ra} \leftarrow \mathtt{pc} \ \mathbf{addw} \ n \\
& & \mid \mathtt{cond} \ \mathit{cop} \ \dot{r}, n \mid \mathtt{call} \ n \mid \mathtt{ret} \\
& & \mid \dot{r}_1 \leftarrow M[\dot{r}_2] \mid M[\dot{r}_1] \leftarrow \dot{r}_2 \mid \mathit{Ann}
\end{array}
$$

Figure 6: PAL Abstract Syntax

2.1 Perilous Assembly Language

Our abstract machine interprets the *perilous assembly language* (PAL); PAL is a variant of Necula's safe assembly language (SAL). PAL has the syntax of SAL, but not its safety checks[4]. PAL is an effective machine model because it resembles a real RISC processor. Necula provides translations from Alpha and x86 assembly language to SAL which also apply to PAL. PAL interprets a program Φ, which is an abstraction of a program stored in memory. Φ cannot be changed by PAL instructions.

2.1.1 Abstract Syntax

Figure 6 contains a definition of the abstract syntax of PAL. PAL models the instruction set of a RISC processor augmented with annotations (Ann) that have no computational effect. The register `mem` contains the memory, and `ra` is designated for return addresses. A variable $\dot{r}$ ranges over general-purpose registers ($\mathtt{r}_i$ plus `ra`); r additionally ranges over `mem`.

The following program, for example, computes the factorial of register $\mathtt{r}_0$ in register $\mathtt{r}_2$:

$$
\begin{array}{ll}
\mathtt{r}_1 \leftarrow \mathbf{1w} & // \ \mathtt{r}_1 \text{ is current counter} \\
\mathtt{r}_2 \leftarrow \mathbf{1w} & // \ \mathtt{r}_2 \text{ is current product} \\
\mathtt{r}_3 \leftarrow \mathbf{1w} & // \ \mathtt{r}_3 \text{ is always one} \\
\mathtt{r}_4 \leftarrow \mathtt{r}_1 \ \mathbf{gtw} \ \mathtt{r}_0 & // \ \mathtt{r}_4 \text{ is nonzero iff } \mathtt{r}_1 > \mathtt{r}_0 \\
\mathtt{cond} \ \mathbf{neq0w} \ \mathtt{r}_4, \mathbf{3w} & // \text{ quit after all iterations} \\
\mathtt{r}_2 \leftarrow \mathtt{r}_2 \ \mathbf{mulw} \ \mathtt{r}_1 & \\
\mathtt{r}_1 \leftarrow \mathtt{r}_1 \ \mathbf{addw} \ \mathtt{r}_3 & \\
\mathtt{cond} \ \mathbf{truew} \ \mathtt{r}_0, -\mathbf{5w} & // \text{ keep looping}
\end{array}
$$

1w is the machine word with the value 1; **neq0w** and **truew** are unary conditionals; **gtw**, **mulw**, and **addw** are binary operators.

2.1.2 Operational Semantics

In this section, we define the PAL operational semantics. A *state transition relation* tells us how the machine transitions from state to state.

[4]We dispense with the SAL call history $\mathcal{H}$, because PAL does not distinguish safe from unsafe execution; a call history can be modeled by a security property. The SAL register move, unconditional jump, and stack load/store instructions can be emulated by PAL instructions.

$$\frac{}{\Phi \triangleright \langle i, \rho\rangle \xrightarrow{\mathsf{m}} s'}\ \mathsf{m}$$

Φ_i	s'
$r \leftarrow n$	$\langle i \dot{+} 1, \rho[r \mapsto \underline{n}]\rangle$
$r_1 \leftarrow r_2\ eop\ r_3$	$\langle i \dot{+} 1, \rho[r_1 \mapsto \mathcal{J}(eop)(\rho(r_2), \rho(r_3))]\rangle$
$\mathtt{ra} \leftarrow \mathtt{pc}\ \mathbf{addw}\ n$	$\langle i \dot{+} 1, \rho[\mathtt{ra} \mapsto \underline{n} \dot{+} i \dot{+} 1]\rangle$
$\mathtt{cond}\ cop\ r, n$	$\langle \underline{n} \dot{+} i \dot{+} 1, \rho\rangle$ if $\mathcal{J}(cop)(\rho(r)) \neq 0$ $\langle i \dot{+} 1, \rho\rangle$ if $\mathcal{J}(cop)(\rho(r)) = 0$
$\mathtt{call}\ n$	$\langle \underline{n}, \rho\rangle$
$\mathtt{ret}$	$\langle \rho(\mathtt{ra}), \rho\rangle$
$r_1 \leftarrow M[r_2]$	$\langle i \dot{+} 1, \rho[r_1 \mapsto \rho(\mathtt{mem})(\rho(r_2))]\rangle$
$M[r_1] \leftarrow r_2$	$\langle i \dot{+} 1, \rho[\mathtt{mem} \mapsto \rho(\mathtt{mem})[\rho(r_1) \mapsto \rho(r_2)]]\rangle$
Ann	$\langle i \dot{+} 1, \rho\rangle$

Figure 7: PAL Transition Relation ($\Phi \triangleright s \xrightarrow{\mathsf{m}} s'$)

$$\frac{\mathrm{Agent}_\Psi(\langle i, \rho\rangle, s) \quad i \in \mathrm{Dom}(\Phi)}{\Psi; \Phi \triangleright <\cdot \langle i, \rho\rangle}\ \mathsf{ma}_1 \qquad \frac{\Phi \triangleright s \xrightarrow{\mathsf{m}} s' \quad \mathrm{Agent}_\Psi(s_0, s')}{\Psi; \Phi \triangleright s \cdot >}\ \mathsf{ma}_4$$

$$\frac{\Phi \triangleright s \xrightarrow{\mathsf{m}} \langle i', \rho'\rangle \quad i' \in \mathrm{Dom}(\Phi)}{\Psi; \Phi \triangleright s \rightarrow \langle i', \rho'\rangle}\ \mathsf{ma}_2 \qquad \frac{\begin{array}{c}\Phi \triangleright s \xrightarrow{\mathsf{m}} s' \quad i'' \in \mathrm{Dom}(\Phi)\\ \mathrm{Skip}_\Psi(s', \langle i'', \rho''\rangle)\end{array}}{\Psi; \Phi \triangleright s \rightarrow \langle i'', \rho''\rangle}\ \mathsf{ma}_3$$

Figure 8: PAL Step Derivation ($\Psi; \Phi \triangleright A$)

Figure 7 contains a schematic inference rule the state transition relation $\xrightarrow{\mathsf{m}}$ (machine transitions are labeled with m). The judgment $\Phi \triangleright s \xrightarrow{\mathsf{m}} s'$ asserts that the machine goes from state s to state s' when executing program Φ. A register environment ρ is a total function on the registers. Φ_i denotes the instruction at address i of program Φ.

The set of all machine words $\mathcal{U}_{\mathbf{wd}}$ is a contiguous subset of the natural numbers. Thus, unsigned arithmetic on words is modulo 2^{WdBit}, where $WdBit$ is the word size in bits; $\dot{+}$ denotes modular addition ($+$ is reserved for non-modular addition). The two's complement sign bit is interpreted by signed operators (*e.g.* multiplication, division) and conditionals. $\underline{n}$ denotes the mathematical value of the numeric constant n, and $\bar{i}$ denotes the numeric constant for the number i.

The memory register $\mathtt{mem}$ contains a total function from words to words; the set of all such functions is $\mathcal{U}_{\mathbf{mapw}}$. $\rho[r \mapsto v]$ denotes the environment ρ except that r is mapped to v (see Section 2.2.2). The function $\mathcal{J}$ maps operator constants to their mathematical interpretations.

An agent program Φ is a sequence of procedures, each of which is a sequence of instructions. Each procedure F has a base address i_0 that is identified with its first instruction. Procedure address ranges must be disjoint. $\mathrm{Dom}(\Phi)$ denotes the instruction addresses of the program Φ. Note that if Φ is derived from some representation in memory, then it should be unwritable, because this semantics does not support self-modifying code.

Because a transition interprets Φ_i (for a state $\langle i, \rho \rangle$), PAL halts if $i \notin \mathrm{Dom}(\Phi)$—this represents abnormal termination, and models a run-time exception. The standard convention for *normal* termination is to execute `ret` with `ra` set to its initial contents.

2.1.3 Step Semantics

From the machine transition relation we define valid steps, and the transitive closure of the step relation defines the valid executions.

There are three kinds step for PAL: a normal transition step, a special "start" step, and a special "stop" step. A start step $<\!\cdot s$ marks s as an initial state, and a stop step $s'\!\cdot\!>$ marks s' as a final state. Start and stop steps delimit executions, and transition steps delimit intermediate states.

For the step semantics, we explicitly model the trusted system invoking the agent; the trusted system is the complete system except for the agent. In implementations of this model, trusted systems are operating system kernels, procedure libraries, or bare hardware (no physical boundary is necessary).

A trusted system Ψ is defined by two binary relations. The *skip* relation $\mathrm{Skip}_\Psi(s, s')$ represents the system calls or library procedures exported by Ψ to the agent. A transition to s triggers a temporary transfer of control to Ψ: s' is a result state once control returns to the agent. Intermediate trusted states are elided from the agent's execution so that Ψ is not subject to a security property[5]. Note that a single trigger state may have many return states to model nondeterminism (*e.g.* input/output). The control transfer is normally initiated by a procedure call to a designated address.

The *agent* relation $\mathrm{Agent}_\Psi(s, s')$ represents the possible agent executions for Ψ, safe or unsafe. The trusted system starts an agent in state s; the agent returns control by a transition to state s'. Thus, s is a possible initial state and s' is an acceptable final state. There may be many s' states for a single s state and Ψ may define several s states to reflect different entry points or initial conditions. The instruction addresses skip, agent return, and the program must be disjoint: thus, the trusted system and the untrusted agent must occupy distinct portions of the address space. Skip trigger states and agent return states may be partial to model run-time checks. Figure 9 illustrates representative transitions between the trusted system and the untrusted agent.

Figure 8 contains inference rules for valid PAL steps; the judgment $\Psi; \Phi \triangleright A$ asserts that A is a valid step when trusted system Ψ executes agent program Φ.

[5]This is why the trusted system is "trusted".

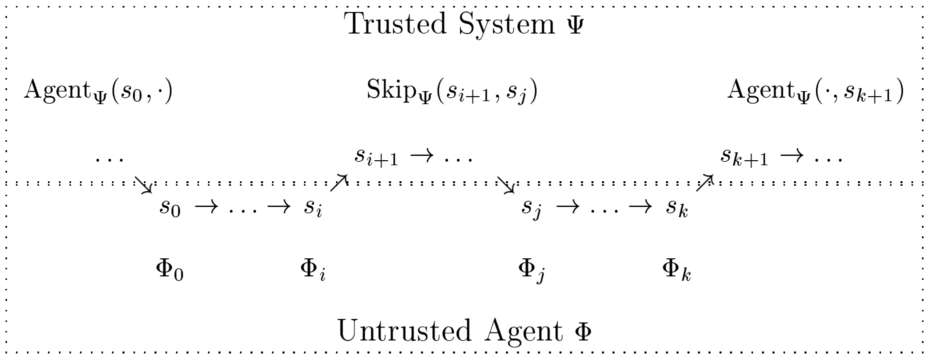

Figure 9: PAL Trusted System/Untrusted Agent Transitions

A start step must contain an initial state of Ψ, but must also match an address of Φ. A stop step is derived by a transition to a final state. A transition step is an ordinary instruction transition, perhaps triggering a skip transition of Ψ.

2.1.4 Trace Semantics

Based on the step relation, we now define the executions of an agent.

Figure 10 contains inference rules for deriving the $\triangleright\!\!\triangleright$ judgment on executions: read $\triangleright\!\!\triangleright\, \sigma$ as "execution σ *is permitted*". Thus, the judgment $\Psi; \Phi \triangleright\!\!\triangleright\, \sigma$ asserts that σ is an execution of agent Φ when executed by system Ψ. The inference rules are the transitive closure of the step rules.

Any valid execution begins with the start symbol $<\!\cdot$. For self-certified code, $<\!\cdot$ represents trusted execution that takes place before the agent is invoked, and is thus is not relevant to the security property. Any valid, *normally terminating* execution ends with the stop symbol $\cdot\!>$: normal termination arises from a transition back to Ψ. For self-certified code, $\cdot\!>$ represents trusted execution that takes place after the agent has terminated. Thus, both nonterminating and aborted executions never reach $\cdot\!>$. An execution is *aborted* when it fails a run-time check.

To summarize Figure 10, all start steps are valid executions, and a transition or stop step may be added to a valid (unterminated) execution if the step matches the final execution state. An execution is thus a sequence of states delimited by $<\!\cdot$, $\rightarrow$, and $\cdot\!>$, and, equivalently, a sequence of steps with overlapping states. Figure 9 illustrates the agent/system transitions for the execution $<\!\cdot s_0 \rightarrow \ldots \rightarrow s_i \rightarrow s_j \rightarrow \ldots \rightarrow s_k \cdot\!>$.

Once an agent relinquishes control, its execution is marked with $\cdot\!>$; nonterminating and aborted executions do not receive $\cdot\!>$; we can thus demonstrate a

$$\frac{\Psi;\Phi \triangleright \mathord{<}\!\cdot s}{\Psi;\Phi \triangleright\!\triangleright \mathord{<}\!\cdot s}\,\mathsf{me}_1 \qquad \frac{\Psi;\Phi \triangleright\!\triangleright \sigma s \quad \Psi;\Phi \triangleright s\cdot\!\mathord{>}}{\Psi;\Phi \triangleright\!\triangleright \sigma s\cdot\!\mathord{>}}\,\mathsf{me}_3$$

$$\frac{\Psi;\Phi \triangleright\!\triangleright \sigma s \quad \Psi;\Phi \triangleright s \rightarrow s'}{\Psi;\Phi \triangleright\!\triangleright \sigma s \rightarrow s'}\,\mathsf{me}_2$$

Figure 10: PAL Execution Derivation ($\Psi;\Phi \triangleright\!\triangleright \sigma$)

hierarchy of executions:

$$\begin{array}{ll} \mathord{<}\!\cdot s_0 \rightarrow s_1 \ldots s_k\cdot\!\mathord{>} & \text{terminated} \\ \mathord{<}\!\cdot s_0 \rightarrow s_1 \ldots s_k & \text{aborted} \\ \mathord{<}\!\cdot s_0 \rightarrow s_1 \ldots & \text{nonterminating} \end{array}$$

In order for a valid execution to be in the agent's execution set, its start and stop states must correspond according to the agent relation. The relation $\text{Agent}_{\Psi,\Phi}(\sigma)$ holds for executions whose stop states, if any, correspond to their start states:

$$\begin{array}{lll} \text{Agent}_{\Psi,\Phi}(\sigma\ s) & & \\ \text{Agent}_{\Psi,\Phi}(\mathord{<}\!\cdot s\cdot\!\mathord{>}) & \text{iff} & \Phi \triangleright s \xrightarrow{\mathsf{m}} s' \text{ and } \text{Agent}_{\Psi}(s, s') \\ \text{Agent}_{\Psi,\Phi}(\mathord{<}\!\cdot s\,\sigma\, s'\cdot\!\mathord{>}) & \text{iff} & \Phi \triangleright s' \xrightarrow{\mathsf{m}} s'' \text{ and } \text{Agent}_{\Psi}(s, s'') \end{array}$$

All aborted executions satisfy this relation, but an execution with a stop step must satisfy the agent relation of the trusted system.

Note that valid executions may still be *unsafe* with respect to a security property; PAL intentionally does not distinguish safe from unsafe executions (this is the function of the security property). $\Sigma_{\Psi,\Phi}$ denotes the set of executions of program Φ with respect to system Ψ; we use the *permits* judgment to define this set:

$$\sigma \in \Sigma_{\Psi,\Phi} \quad \text{iff} \quad \Psi;\Phi \triangleright\!\triangleright \sigma \text{ and } \text{Agent}_{\Psi,\Phi}(\sigma)$$

By defining $\Sigma_{\Psi,\Phi}$, we can reason about the possible executions when Ψ executes Φ. Later in this report, we will leverage the trace semantics to define the executions of a security property—systematically relating these execution sets is the focus of this report. We are nearly ready to introduce the language in which security properties are written; first, however, we define a simple logic that forms a "sublanguage" of the security-property language.

2.2 First-Order Logic

Security-property-language statements are based on first-order propositions. Not coincidentally, the VC-generator results of Section 3 are encoded in the

Expressions	$E ::= c \mid x \mid f(E_1, \ldots, E_k)$	$E \in \mathit{Exp}$
Propositions	$P ::= \top \mid R(E_1, \ldots, E_k) \mid E_1 = E_2 \mid E_1 \neq E_2$ $\mid P_1 \wedge P_2 \mid P_1 \supset P_2 \mid \forall x : \tau.P_1$	$P \in \mathit{Prop}$

Figure 11: Logic Abstract Syntax

same logic. The logic thus has a dual role: it is embedded in the security-property language, but it is also the VC object language. This logic is the least expressive such logic that suffices for our security-property language and VC generator: more expressive logics (*e.g.*, higher-order logic) can be substituted if they admit our fragment. A more expressive logic increases the expressiveness of the security-property language.

2.2.1 Abstract Syntax

The abstract syntax of the logic is defined in Figure 11. Type constructors τ include **wd**, the type of machine words, and **mapw**, the type of maps from words to words.

Expressions encompass constants, variables, and applications of constant functions; the set of functions includes selection (**selw**) and update (**updw**) on word maps. Note that PAL register identifiers are chosen from the variables as a technical convenience. The PAL meta-variable *eop* denotes a binary operator (*i.e.*, constant function) on words; the exact set of operators is determined by the processor, but we presume the existence of modular addition (**addw**) and subtraction (**subw**).

Logical connectives include conjunction and implication as well as truth and applications of constant relations. Universal quantifiers for all types are available. Equality and inequality relations are polymorphic and are well-formed for any single type. The PAL meta-variable *cop* denotes a unary conditional (*i.e.*, constant relation) on words; the exact set of conditionals is determined by the processor, but the VC generator requires the conditionals to be closed under complementation (*e.g.*, $(\neg <) \equiv (\geq)$). Note that we omit existential quantifiers to simplify proof-term representations as well as automatic theorem provers.

The free variables of an expression ($\mathrm{FV}(E)$) or proposition ($\mathrm{FV}(P)$) are defined in the customary way:

$$
\begin{array}{ll}
\mathrm{FV}(c) & = \emptyset \\
\mathrm{FV}(x) & = \{x\} \\
\mathrm{FV}(f(E_1, \ldots, E_k)) & = \mathrm{FV}(E_1) \cup \cdots \cup \mathrm{FV}(E_k) \\[4pt]
\mathrm{FV}(\top) & = \emptyset \\
\mathrm{FV}(R(E_1, \ldots, E_k)) & = \mathrm{FV}(E_1) \cup \cdots \cup \mathrm{FV}(E_k) \\
\mathrm{FV}(E_1 = E_2) & = \mathrm{FV}(E_1) \cup \mathrm{FV}(E_2) \\
\mathrm{FV}(E_1 \neq E_2) & = \mathrm{FV}(E_1) \cup \mathrm{FV}(E_2) \\
\mathrm{FV}(P_1 \wedge P_2) & = \mathrm{FV}(P_1) \cup \mathrm{FV}(P_2) \\
\mathrm{FV}(P_1 \supset P_2) & = \mathrm{FV}(P_1) \cup \mathrm{FV}(P_2) \\
\mathrm{FV}(\forall x : \tau . P_1) & = \mathrm{FV}(P_1) \setminus \{x\}
\end{array}
$$

2.2.2 Model-Theoretic Semantics

In this section, we define a model-theoretic semantics for the first-order logic; because the logic is embedded in the security-property language, its semantics grounds the language semantics.

We define a model $\mathcal{M}$ for the logic as follows:

$$\mathcal{M} = \langle \{\mathcal{U}_\tau\}_\tau, \mathcal{J} \rangle$$

A set of values $\mathcal{U}_\tau$ (a *universe*) is associated with each type symbol τ; the PAL semantics (see Section 2.1.2) uses these universes. $\mathcal{U}_{\mathbf{wd}}$ is an initial subrange of the natural numbers. $\mathcal{U}_{\mathbf{mapw}}$ is the total functions from words to words. Constants (including functions and relations) are assigned values by the denotation function $\mathcal{J}$. The notation for numeric constants abbreviates uses of $\mathcal{J}$:

$$\underline{n} = \mathcal{J}(n) \qquad i = \mathcal{J}(\bar{i})$$

Modular addition and subtraction similarly abbreviate constant functions:

$$i_1 \dot{+} i_2 = \mathcal{J}(\mathbf{addw})(i_1, i_2) \qquad i_1 \dot{-} i_2 = \mathcal{J}(\mathbf{subw})(i_1, i_2)$$

A valuation function assigns values to expressions; the valuation function $\mathcal{V}_{\mathcal{M},\phi}$ is defined in Figure 12; the environment ϕ assigns values to free variables. A validity judgment asserts that a given proposition holds; the validity judgment $\mathcal{M} \models_\phi$ is defined in Figure 12. Because we define only one model in this report, we abbreviate the valuation function $\mathcal{V}_\phi$, and the validity judgment $\models_\phi$.

$$E \in \operatorname{dom} \mathcal{V}_\phi \quad \text{iff} \quad \mathrm{FV}(E) \subseteq \operatorname{dom} \phi$$

$$\begin{array}{ll}
\mathcal{V}_\phi(c) & = \mathcal{J}(c) \\
\mathcal{V}_\phi(x) & = \phi(x) \\
\mathcal{V}_\phi(f(E_1, \ldots, E_k)) & = \mathcal{J}(f)(\mathcal{V}_\phi(E_1), \ldots, \mathcal{V}_\phi(E_k))
\end{array}$$

$$\begin{array}{lll}
\models_\phi P & \text{iff} & \mathrm{FV}(P) \subseteq \operatorname{dom} \phi \\
& & \text{and} \begin{cases}
\mathcal{J}(R)(\mathcal{V}_\phi(E_1), \ldots, \mathcal{V}_\phi(E_k)) \neq 0 & \text{if } P \equiv R(E_1, \ldots, E_k) \\
\mathcal{V}_\phi(E_1) = \mathcal{V}_\phi(E_2) & \text{if } P \equiv E_1 = E_2 \\
\mathcal{V}_\phi(E_1) \neq \mathcal{V}_\phi(E_2) & \text{if } P \equiv E_1 \neq E_2 \\
\models_\phi P_1 \text{ and } \models_\phi P_2 & \text{if } P \equiv P_1 \wedge P_2 \\
\models_\phi P_2 \text{ if } \models_\phi P_1 & \text{if } P \equiv P_1 \supset P_2 \\
\models_{\phi[x \mapsto v]} P_1 \text{ for all } v \in \mathcal{U}_\tau & \text{if } P \equiv \forall x : \tau . P_1
\end{cases}
\end{array}$$

Figure 12: Logic Valuation and Validity

The interpretation function is defined over the standard constants as follows:

$$\begin{array}{ll}
\mathcal{J}(\mathbf{selw})(v, i) & = v(i) \\
\mathcal{J}(\mathbf{updw})(v, i_1, i_2) & = v[i_1 \mapsto i_2] \\
\mathcal{J}(\mathbf{0w}) & = 0 \\
\mathcal{J}(\mathbf{1w}) & = 1 \\
\vdots & \\
\mathcal{J}(\mathbf{addw})(i_1, i_2) & = (i_1 + i_2) \bmod 2^{WdBit} \\
\mathcal{J}(\mathbf{subw})(i_1, i_2) & = (i_1 + 2^{WdBit} - i_2) \bmod 2^{WdBit} \\
\mathcal{J}(\mathbf{eq0w})(i) & = \begin{cases} 1 & \text{if } i = 0 \\ 0 & \text{otherwise} \end{cases} \\
\mathcal{J}(\mathbf{neq0w})(i) & = \begin{cases} 1 & \text{if } i \neq 0 \\ 0 & \text{otherwise} \end{cases}
\end{array}$$

where function extension satisfies the following equations:

$$\begin{array}{ll}
\operatorname{dom}(v_1[v_2 \mapsto v_3]) & = \operatorname{dom} v_1 \cup \{v_2\} \\
(v_1[v_2 \mapsto v_3])(v) & = \begin{cases} v_3 & \text{if } v = v_2 \\ v_1(v) & \text{otherwise} \end{cases}
\end{array}$$

2.2.3 Environments

An environment assigns values to the free variables of an expression or proposition and is represented by a partial function from variables to values. Two environments are equal when they have the same domain and they map each variable of this domain to an identical value:

$$\begin{array}{lll} \phi_1 = \phi_2 & \text{iff} & \mathrm{dom}\,\phi_1 = \mathrm{dom}\,\phi_2 \text{ and} \\ & & \phi_1(x) = \phi_2(x) \text{ for any } x \in \mathrm{dom}\,\phi_1 \end{array}$$

An environment is a subset of another environment when the smaller agrees with the larger over its entire domain:

$$\begin{array}{lll} \phi_1 \subseteq \phi_2 & \text{iff} & \mathrm{dom}\,\phi_1 \subseteq \mathrm{dom}\,\phi_2 \text{ and} \\ & & \phi_1(x) = \phi_2(x) \text{ for any } x \in \mathrm{dom}\,\phi_1 \end{array}$$

Environment operations include extension ($[\mapsto]$), concatenation ($\cup$), and restriction ($\setminus$); the environment with an empty domain is $\phi_\emptyset$:

$$\begin{array}{ll} \mathrm{dom}(\phi_1 \cup \phi_2) & = \mathrm{dom}\,\phi_1 \cup \mathrm{dom}\,\phi_2 \\ \mathrm{dom}(\phi[x \mapsto v]) & = \mathrm{dom}\,\phi \cup \{x\} \\ \mathrm{dom}(\phi \setminus X) & = \mathrm{dom}\,\phi \setminus X \\ \mathrm{dom}\,\phi_\emptyset & = \emptyset \\ (\phi_1 \cup \phi_2)(x) & = \begin{cases} \phi_2(x) & \text{if } x \in \mathrm{dom}\,\phi_2 \\ \phi_1(x) & \text{otherwise} \end{cases} \\ (\phi[y \mapsto v])(x) & = \begin{cases} v & \text{if } x = y \\ \phi(x) & \text{otherwise} \end{cases} \\ (\phi \setminus X)(x) & = \phi(x) \end{array}$$

Concatenation gives precedence to its right-hand operand. We abbreviate validity of a (closed) proposition by omitting the environment:

$$\models P \quad \text{iff} \quad \models_{\phi_\emptyset} P$$

Note that because PAL register identifiers are variables, a register environment ρ is also a ϕ. We identify a PAL *state* with an environment through the notation ϕ_s:

$$\phi_{\langle i,\rho\rangle} = \rho[\mathtt{pc} \mapsto i]$$

Environments play a crucial role in the language semantics; with the logic defined, we introduce the language.

2.3 Palladium Security Properties

Each term of a *security-property language* denotes an execution set. The syntax and semantics of our security-property language are defined mathematically; the semantics defines an execution set for each well-formed term. Based on the semantics, properties of individual terms can be demonstrated, and characteristics of the language itself can be established.

A security property filters the executions of a system. A security-property is by design unable to perturb the agent—it can only disallow certain executions. This property is essential to check an agent without actually executing it. Checking an *executing* agent can be implemented by monitoring the agent and aborting it as soon a *safety* property would be violated.

A security-property language can be defined without defining an enforcement mechanism. If an enforcement mechanism implements the language semantics, then a developer need verify only that a security property defines the correct execution set. This is a mathematical process because the language is defined mathematically; in Appendix A we demonstrate this technique.

In this section, we define the security-property language *Palladium*. Palladium is suitable for a symbolic evaluator or a safe interpreter. In Section 3 we specify a VC-generator interpreter of this language, and in Section 4 its soundness. Palladium expresses only safety properties, but in Section A.4 we discuss the possibility of enforcing liveness properties.

2.3.1 Programming Model

The activity of developing a Palladium security property resembles programming. In particular, the imperative language model entails dependencies on run-time state and the order of evaluation rules is significant. A security property resembles a program for an agent reference monitor. We believe that non-imperative security-property languages are also possible, but we think that an imperative language suits an imperative machine.

Many security properties manipulate *property registers*, mutable locations outside the machine that encode fragments of execution history. Property registers tailor an execution history to enable concise safety conditions.

Although we can picture a Palladium security property as a reference monitor, such security properties can be enforced without any run-time overhead: with PCC, for example, we can reason about property registers in the absence of a concrete representation. The reference monitor is a visualization of execution sets: ensuring that a program is bounded by an execution set is a separate matter. The property registers are a security-automaton state in the sense of Schneider [Sch99]—Palladium is thus a concrete notation for security automata.

A Palladium security property is composed of declarations and rules. *Declarations* introduce new language elements such as property registers; *rules* determine the executions of the security property. A *requirement rule* specifies a precondition for a step based on a formal proposition; an *evaluation rule* assigns a property register based on a formal expression.

We can define a safe interpreter for Palladium by extending the trace semantics. Picture the interpreter as a safety kernel [Rus89, WK95] or reference monitor observing the agent, and picture the property registers as the kernel state. An evaluation rule changes the kernel state; a requirement rule speci-

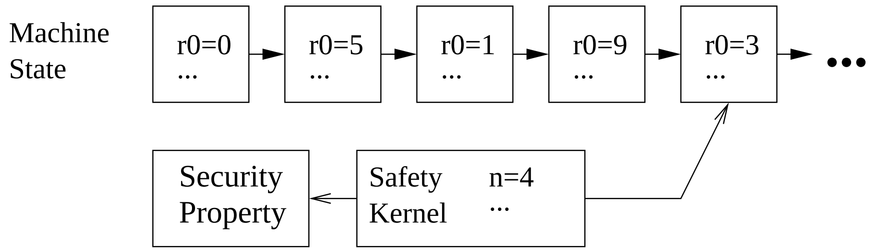

Figure 13: Parameterized Safety Kernel

$$\begin{array}{lll} \text{Sec. Prop.} & \mathcal{P} ::= \cdot \mid \texttt{reg}\ \hat{r} : \tau ; \mathcal{P} & \mathcal{P} \in \mathit{SecProp} \\ & \quad \mid \texttt{require}\ A^{\square} \Rightarrow P ; \mathcal{P} \mid \texttt{admit}\ A^{\square} \Rightarrow P ; \mathcal{P} & \\ & \quad \mid \texttt{eval}\ A^{\square} \Rightarrow \hat{r} := E ; \mathcal{P} \mid \texttt{new}\ A^{\square} \Rightarrow \hat{r} : \tau ; \mathcal{P} & \\ & \quad \mid \texttt{scs}\ n^{\square} \Rightarrow r ; \mathcal{P} \mid \texttt{ucs}\ n^{\square} \Rightarrow r ; \mathcal{P} & \end{array}$$

Figure 14: Security-Property Abstract Syntax

fies when to abort the agent. Labeling the kernel a *monitor* suggests that it should not affect the agent, other than to abort it—we prove this property in Section 2.3.5.

Picture the kernel evaluating the security property at each agent step (see Figure 13). Rules are evaluated in textual order: the kernel updates property registers for evaluation rules, but it aborts the agent if a requirement rule fails. A rule is guarded by a *step pattern* that restricts the steps to which it applies. Patterns may have free variables to let a single rule serve as a schema.

Rules are given a sequential order to mimic a programming language; state dependencies enhance expressiveness. Requirement rules are conjunctive to support local reasoning: whether a given requirement fails is not a function of other requirements.

Given this intuition, we now define Palladium.

2.3.2 Abstract Syntax

The abstract syntax of Palladium is defined in Figure 14. Types, expressions and propositions are drawn from Section 2.2; $\mathcal{P}$ is a security-property term.

A declaration `reg` $\hat{r} : \tau$ introduces property register $\hat{r}$ with type τ; property-register identifiers are variables. All declarations must introduce distinct identifiers. The $\hat{}$ accent distinguishes property registers from machine registers.

A rule `require` $A^{\square} \Rightarrow P$ is evaluated before a step if it matches the pattern $A^{\square}$: the execution aborts if P does not hold. The $^{\square}$ superscript distinguishes a *pattern* for a step from a step. The proposition P may refer to property registers, machine registers, and pattern variables. Property registers are given

$$
\begin{array}{lll}
\text{Registers} & r^\square & ::= \dot{r} \mid x \mid \_ \\[4pt]
\text{Words} & n^\square & ::= n \mid x \mid \_ \\[4pt]
\text{Instructions} & I^\square & ::= r^\square \leftarrow n^\square \mid r_1^\square \leftarrow r_2^\square \ \mathit{eop}\ r_3^\square \mid \texttt{ra} \leftarrow \texttt{pc}\ \textbf{addw}\ n^\square \\
 & & \quad \mid \texttt{cond}\ \mathit{cop}\ r^\square, n^\square \mid \texttt{call}\ n^\square \mid \texttt{ret} \\
 & & \quad \mid r_1^\square \leftarrow M[r_2^\square] \mid M[r_1^\square] \leftarrow r_2^\square \\[4pt]
\text{States} & s^\square & ::= \texttt{proc}\ n^\square \mid I^\square \mid \&n^\square \mid \_ \\[4pt]
\text{Steps} & A^\square & ::= <\!\cdot s^\square \mid \ \to s^\square \mid s^\square \to \ \mid s^\square\cdot\!> \mid \ \stackrel{<}{\to} s^\square \mid s^\square \stackrel{>}{\to}
\end{array}
$$

Figure 15: Pattern Abstract Syntax

values when the rule is evaluated: thus, the placement of an evaluation rule can affect whether a requirement rule holds. Machine register values are taken from the step according to the form of $A^\square$. For a step $s \to s'$, an *entering* pattern $\to s^\square$ uses s', but a *leaving* pattern $s^\square \to$ uses s.

The abstract syntax of patterns is defined in Figure 15. A step pattern $A^\square$ contains a *state pattern* $s^\square$: a state matches a state pattern according to its instruction counter, which can match an instruction pattern, a procedure pattern, or a word constant. For example, the pattern `ret` matches a state executing the `ret` instruction. The state pattern matches according to the shape of the step pattern: if s matches $s^\square$, then

$$
\begin{array}{llll}
<\!\cdot s & \text{matches} & <\!\cdot s^\square & \text{or} \quad \stackrel{<}{\to} s^\square \\
s' \to s & \text{matches} & \to s^\square & \text{or} \quad \stackrel{<}{\to} s^\square \\
s \to s' & \text{matches} & s^\square \to & \text{or} \quad s^\square \stackrel{>}{\to} \\
s\cdot\!> & \text{matches} & s^\square\cdot\!> & \text{or} \quad s^\square \stackrel{>}{\to}
\end{array}
$$

Thus, $<\!\cdot s^\square$ matches start steps, $s^\square\cdot\!>$ matches stop steps, $\stackrel{<}{\to} s^\square$ matches both transition and start steps, and $s^\square \stackrel{>}{\to}$ matches both transition and stop steps.

A requirement rule asserts a bounded universal quantifier over all steps: read `require` $A^\square \Rightarrow P$ as "for all steps matching A, P must hold." Equivalently, a requirement rule asserts an *always* operator of temporal logic [Eme90].

An assumption rule `admit` $A^\square \Rightarrow P$ is evaluated after a step if it matches $A^\square$: the agent can then be aborted only if P holds. An assumption rule thus specifies a condition that the agent can assume for the future: if the assumption fails, then no further requirements will be enforced. Assumption rules make it easier to construct certified agents. Trusted-procedure postconditions are encoded in assumptions and provide proof-invariants for PCC when trusted procedures are involved; the assumptions make it possible to prove that later requirements are satisfied. For example, an assumption rule might specify that the result of the `open` procedure is a file descriptor: later, we can show that an argument to `put` is also a file descriptor. Note that if an assumption always holds, then it has no semantic effect; if an assumption fails, it effectively weakens a security property.

An evaluation rule `eval` $A^\square \Rightarrow \hat{r} := E$ is evaluated after a step if it matches $A^\square$: the rule assigns $\hat{r}$ the value of E. An expression is assigned a value in the same way that a proposition is checked.

A rule `new` $A^\square \Rightarrow \hat{r} : \tau$ is similar to an evaluation rule, except that the new value of $\hat{r}$ is an unspecified function of the current step; the new value of $\hat{r}$ can be constrained using assumption rules. One use of this rule is to conditionally assign a property register. For example, the following fragment assigns `f` zero or one according to the value of `k`:

$$\begin{array}{l} \texttt{reg f} : \textbf{nat} \\ \texttt{reg k} : \textbf{nat} \end{array}$$

$$\begin{array}{l} \texttt{new} \mathrel{\_} \Rightarrow \texttt{f} : \textbf{nat} \\ \texttt{admit} \mathrel{\_} \Rightarrow \texttt{k} \leq 10 \supset \texttt{f} = 0 \\ \texttt{admit} \mathrel{\_} \Rightarrow \texttt{k} > 10 \supset \texttt{f} = 1 \end{array}$$

A rule `scs` $n^\square \Rightarrow r$ admits that register r will be preserved by all trusted calls matching $n^\square$. A rule `ucs` $n^\square \Rightarrow r$ requires register r to be preserved by all agent procedures matching $n^\square$.

As an illustrative example, consider a security property that allows up to 10 calls to the procedure `put`:

$$\texttt{reg nPut} : \textbf{nat}$$

$$\begin{array}{l} \texttt{eval} \ <\cdot\_ \Rightarrow \texttt{nPut} := 0 \\ \texttt{require call put} \rightarrow\Rightarrow \texttt{nPut} \leq 9 \\ \texttt{eval call put} \rightarrow\Rightarrow \texttt{nPut} := \texttt{nPut} + 1 \end{array}$$

`nPut` is a property register that counts the number of times `put` has been called; `nPut` is declared with a natural number type. The rule `eval` $<\cdot\_ \Rightarrow$ `nPut` $:= 0$ specifies that an execution begins without any calls to `put` ($<\cdot\_$ matches at execution start). The rule `require call put` $\rightarrow\Rightarrow$ `nPut` ≤ 9 specifies that any execution of the instruction `call put` can be preceded by up to 9 calls to `put`. Finally, the rule `eval call put` $\rightarrow\Rightarrow$ `nPut` $:=$ `nPut` $+ 1$ specifies that `nPut` is incremented for each call to `put`.

2.3.3 Operational Semantics

To define the executions of a security property, we extended the semantics by evaluating the security property at each step. But before defining security-property evaluation, we first define the effect of a single rule on a *security-property state*.

Security-property states are triples $\langle \mathcal{P}, \hat{\rho}, q \rangle$, where $\mathcal{P}$ is the security property being evaluated, $\hat{\rho}$ is the current property-register environment, and q is one if all prior assumptions have held. A property-register environment $\hat{\rho}$ maps property registers to values and is total on the property registers. The value of q is one initially, and becomes zero only when an assumption fails. Figure 16 defines the transition relation for security-property states $\hat{s}$.

The security property is evaluated for each step: to evaluate a security property $\mathcal{P}$, we construct a state with $\mathcal{P}$ and make successive transitions until $\mathcal{P}$ is empty. The judgment $\Phi; A \triangleright \hat{s} \stackrel{\mathsf{s}}{\to} \hat{s}'$ asserts that there is a transition from state $\hat{s}$ to state $\hat{s}'$ with respect to step A and program Φ. A *large transition* is its reflexive, transitive closure: the judgment $\Phi; A \triangleright \hat{s} \stackrel{\mathsf{s*}}{\to} \hat{s}'$ asserts that there is a large transition from $\hat{s}$ to $\hat{s}'$. Transitions are labeled with `s` and large transitions are labeled with `s*`.

There is no transition if a requirement fails when all prior assumptions hold; thus, a large transition gets "stuck" when an execution fails a requirement. Because the security property must evaluate completely at each step, this effectively aborts the execution.

The function $\text{New}_\Phi(A, \langle \mathcal{P}, \hat{\rho}, q\rangle)$ returns a value $v \in \mathcal{U}_\tau$ for $\hat{r}$ when $\mathcal{P} = \texttt{new}\ A^\square \Rightarrow \hat{r} : \tau; \mathcal{P}'$. v is chosen so that no assumptions will fail in $\mathcal{P}'$, if such a choice is possible. v depends only on the arguments to New to make security-property evaluation deterministic; *i.e.*, if both $\Phi; A \triangleright \hat{s} \stackrel{\mathsf{s*}}{\to} \hat{s}_1$ and $\Phi; A \triangleright \hat{s} \stackrel{\mathsf{s*}}{\to} \hat{s}_2$, then $\hat{s}_1 = \hat{s}_2$.

The following lemma is useful later in this section:

Lemma 2.1 (Absence of Assumptions) $\Phi; A \triangleright \langle \mathcal{P}, \hat{\rho}, q\rangle \stackrel{\mathsf{s*}}{\to} \langle \mathcal{P}', \hat{\rho}', q'\rangle$ *implies* $q' = q$ *if* $\mathcal{P}$ *contains no* `admit` *rules*

PROOF:
By induction on the derivation of $\Phi; A \triangleright \langle \mathcal{P}, \hat{\rho}, q\rangle \stackrel{\mathsf{s*}}{\to} \langle \mathcal{P}', \hat{\rho}', q'\rangle$

$\square$

The judgment $\Phi \triangleright A^\square; A \stackrel{\mathsf{pa}}{\to} \theta; \phi$ asserts that pattern $A^\square$ matches step A; the substitution θ binds the pattern variables of $A^\square$ (a substitution θ maps variables to expressions). The environment ϕ binds the machine registers based on the appropriate state of A. Pattern matching is standard, so the definition of this judgment is relegated to Appendix C.

We have shown how property registers are updated by the security property for a single step. In the next section, we attach property registers to PAL states to link successive steps of an execution: each execution will then have a "shadow" sequence of property register states.

2.3.4 Extended Step Semantics

In this section, we extend the PAL semantics to maintain property registers from state to state. An extended state $\check{s}$ is a triple $\langle s, \hat{\rho}, q\rangle$ where s is a PAL state, $\hat{\rho}$ is the property-register environment for s, and q is one if all prior assumptions have held. Figure 18 contains the step semantics for extended states. Extended states carry $\hat{\rho}$ and q between successive security-property evaluations.

The judgment $\Psi; \mathcal{P}; \Phi \triangleright \check{A}$ asserts that $\check{A}$ is a valid extended step with respect to Ψ, $\mathcal{P}$, and Φ. Extended steps are PAL steps with extended states. Such a step is valid if the corresponding PAL step is valid and if the security property evaluates completely. The initial property register environment $\hat{\rho}_\mathcal{P}$ must be

$$\frac{}{\Phi; A \triangleright \langle \texttt{reg}\ \hat{r} : \tau; \mathcal{P}, \hat{\rho}, q\rangle \xrightarrow{\mathsf{s}} \langle \mathcal{P}, \hat{\rho}, q\rangle}\ \mathsf{s}_1$$

$$\frac{\Phi \triangleright A^{\square}; A \overset{\mathsf{pa}}{\not\to}}{\Phi; A \triangleright \langle \mathcal{P}, \hat{\rho}, q\rangle \xrightarrow{\mathsf{s}} \langle \mathcal{P}', \hat{\rho}, q\rangle}\ \mathsf{s}_3$$

$\mathcal{P}$
$\texttt{require}\ A^{\square} \Rightarrow P; \mathcal{P}'$
$\texttt{admit}\ A^{\square} \Rightarrow P; \mathcal{P}'$
$\texttt{eval}\ A^{\square} \Rightarrow \hat{r} := E; \mathcal{P}'$
$\texttt{new}\ A^{\square} \Rightarrow \hat{r} : \tau; \mathcal{P}'$

$$\frac{\Phi \triangleright A^{\square}; A \xrightarrow{\mathsf{pa}} \theta; \phi}{\Phi; A \triangleright \langle \mathcal{P}, \hat{\rho}, q\rangle \xrightarrow{\mathsf{s}} \langle \mathcal{P}', \hat{\rho}', q'\rangle}\ \mathsf{s}_2$$

$\mathcal{P}$	Case	$\hat{\rho}'$	q'
$\texttt{require}\ A^{\square} \Rightarrow P; \mathcal{P}'$	$\vDash_{\phi \cup \hat{\rho}} \theta(P)$ if $q \neq 0$	$\hat{\rho}$	q
$\texttt{admit}\ A^{\square} \Rightarrow P; \mathcal{P}'$	$\vDash_{\phi \cup \hat{\rho}} \theta(P)$	$\hat{\rho}$	q
$\texttt{admit}\ A^{\square} \Rightarrow P; \mathcal{P}'$	$\nvDash_{\phi \cup \hat{\rho}} \theta(P)$	$\hat{\rho}$	0
$\texttt{eval}\ A^{\square} \Rightarrow \hat{r} := E; \mathcal{P}'$	$v = \mathcal{V}_{\phi \cup \hat{\rho}}(\theta(E))$	$\hat{\rho}[\hat{r} \mapsto v]$	q
$\texttt{new}\ A^{\square} \Rightarrow \hat{r} : \tau; \mathcal{P}'$	$v = \mathrm{New}_{\Phi}(A, \langle \mathcal{P}, \hat{\rho}, q\rangle)$	$\hat{\rho}[\hat{r} \mapsto v]$	q

Figure 16: Security-Property Transition Relation ($\Phi; A \triangleright \hat{s} \xrightarrow{\mathsf{s}} \hat{s}'$)

$$\frac{}{\Phi; A \triangleright \hat{s} \xrightarrow{\mathsf{s*}} \hat{s}}\ \mathsf{s*}_1 \qquad \frac{\Phi; A \triangleright \hat{s}_0 \xrightarrow{\mathsf{s*}} \hat{s} \quad \Phi; A \triangleright \hat{s} \xrightarrow{\mathsf{s}} \hat{s}'}{\Phi; A \triangleright \hat{s}_0 \xrightarrow{\mathsf{s*}} \hat{s}'}\ \mathsf{s*}_2$$

Figure 17: Security-Property Large-Transition Relation ($\Phi; A \triangleright \hat{s} \xrightarrow{\mathsf{s*}} \hat{s}'$)

well-typed with respect to $\mathcal{P}$ and zero elsewhere:

$$\begin{array}{ll} \hat{\rho}_{\mathcal{P}}(\hat{r}) \in \mathcal{U}_\tau & \text{for each } \hat{r} : \tau \in \Gamma_{\mathcal{P}} \\ \hat{\rho}_{\mathcal{P}}(\hat{r}) = 0 & \text{for each } \hat{r} \notin \operatorname{dom} \Gamma_{\mathcal{P}} \end{array}$$

Finally, in order to check specifications, the notation $\phi_{\check{s}}$ gives us an environment for an extended state:

$$\phi_{\langle s,\hat{\rho},q\rangle} = \phi_s \cup \hat{\rho}$$

We can now connect extended steps to form extended executions.

2.3.5 Extended Trace Semantics

Extended executions are PAL executions with a property-register history. The extended trace semantics follows the PAL trace semantics; an extended execution $\check{\sigma}$ is a PAL execution of extended states. The judgment $\Psi; \mathcal{P}; \Phi \rhd\!\!\rhd \check{\sigma}$ asserts that $\check{\sigma}$ is a valid extended execution with respect to Ψ, $\mathcal{P}$, and Φ.

The function $\lfloor\cdot\rfloor$ *erases* $\hat{\rho}$ and q from a state or execution, and thereby recovers a PAL state or execution:

$$\lfloor\langle s,\hat{\rho},q\rangle\rfloor = s \qquad \begin{array}{ll} \lfloor\cdot\rfloor & = \cdot \\ \lfloor<\!\cdot\rfloor & = <\!\cdot \\ \lfloor\check{\sigma}\rightarrow\rfloor & = \lfloor\check{\sigma}\rfloor\rightarrow \\ \lfloor\check{\sigma}\cdot\!>\rfloor & = \lfloor\check{\sigma}\rfloor\cdot\!> \\ \lfloor\check{\sigma}\check{s}\rfloor & = \lfloor\check{\sigma}\rfloor\lfloor\check{s}\rfloor \end{array}$$

By erasing extended executions, we can recover the PAL executions that are valid for a given security property. $\mathcal{P}(\Sigma_{\Psi,\Phi})$ is the execution set of $\mathcal{P}$ with respect to Ψ and Φ. An execution σ is in $\mathcal{P}$ if σ is the erasure of some $\check{\sigma}$ which is permitted by $\mathcal{P}$:

$$\lfloor\check{\sigma}\rfloor \in \mathcal{P}(\Sigma_{\Psi,\Phi}) \quad \text{iff} \quad \Psi;\mathcal{P};\Phi \rhd\!\!\rhd \check{\sigma} \text{ and } \mathrm{Agent}_{\Psi,\Phi}(\lfloor\check{\sigma}\rfloor)$$

Thus Φ is safe with respect to $\mathcal{P}$ if $\Sigma_{\Psi,\Phi} \subseteq \mathcal{P}(\Sigma_{\Psi,\Phi})$.

Note that by this definition, a security property includes the partial executions of all valid executions: thus, an agent can abort at any prefix of a valid execution. This is harmless for safety properties, because they are inherently prefix closed, but a different strategy is needed to treat liveness properties.

We can now show that a security property cannot affect the agent; *i.e.*, it can introduce no new executions. $\mathcal{P}(\Sigma_{\Psi,\Phi}) \subseteq \Sigma_{\Psi,\Phi}$ for any $\mathcal{P}$ is a consequence of the following theorem:

Theorem 2.1 (Transparency) $\Psi;\mathcal{P};\Phi \rhd\!\!\rhd \check{\sigma}$ *implies* $\Psi;\Phi \rhd\!\!\rhd \lfloor\check{\sigma}\rfloor$

PROOF:
By induction on the derivation of $\Psi;\mathcal{P};\Phi \rhd\!\!\rhd \check{\sigma}$

□

This implies that if Φ satisfies $\mathcal{P}$, then $\mathcal{P}(\Sigma_{\Psi,\Phi}) = \Sigma_{\Psi,\Phi}$. A final theorem is useful for proving other theorems about specific security properties:

$$\frac{\Psi;\Phi \triangleright <\!\cdot s \quad \Phi; <\!\cdot s \triangleright \langle \mathcal{P}, \hat{\rho}_{\mathcal{P}}, 1\rangle \overset{\mathsf{s}*}{\rightarrow} \langle \cdot, \hat{\rho}, q\rangle}{\Psi;\mathcal{P};\Phi \triangleright <\!\cdot \langle s, \hat{\rho}, q\rangle}\ \mathsf{ea}_1$$

$$\frac{\Psi;\Phi \triangleright A \quad \Phi; A \triangleright \langle \mathcal{P}, \hat{\rho}, q\rangle \overset{\mathsf{s}*}{\rightarrow} \langle \cdot, \hat{\rho}', q'\rangle}{\Psi;\mathcal{P};\Phi \triangleright \check{A}}\ \mathsf{ea}_2$$

A	$\check{A}$
$s \to s'$	$\langle s, \hat{\rho}, q\rangle \to \langle s', \hat{\rho}', q'\rangle$
$s \cdot >$	$\langle s, \hat{\rho}, q\rangle \cdot >$

Figure 18: Extended Step Derivation ($\Psi;\mathcal{P};\Phi \triangleright \check{A}$)

$$\frac{\Psi;\mathcal{P};\Phi \triangleright <\!\cdot \check{s}}{\Psi;\mathcal{P};\Phi \mathrel{\triangleright\!\triangleright} <\!\cdot \check{s}}\ \mathsf{ee}_1$$

$$\frac{\Psi;\mathcal{P};\Phi \mathrel{\triangleright\!\triangleright} \check{\sigma}\check{s} \quad \Psi;\mathcal{P};\Phi \triangleright \check{s} \to \check{s}'}{\Psi;\mathcal{P};\Phi \mathrel{\triangleright\!\triangleright} \check{\sigma}\check{s} \to \check{s}'}\ \mathsf{ee}_2 \qquad \frac{\Psi;\mathcal{P};\Phi \mathrel{\triangleright\!\triangleright} \check{\sigma}\check{s} \quad \Psi;\mathcal{P};\Phi \triangleright \check{s} \cdot >}{\Psi;\mathcal{P};\Phi \mathrel{\triangleright\!\triangleright} \check{\sigma}\check{s} \cdot >}\ \mathsf{ee}_3$$

Figure 19: Extended Execution Derivation ($\Psi;\mathcal{P};\Phi \mathrel{\triangleright\!\triangleright} \check{\sigma}$)

Theorem 2.2 (Absence of Assumptions) $\Psi; \mathcal{P}; \Phi \rhd\!\!\rhd \breve{\sigma}\ \langle s, \hat{\rho}, q\rangle$ *implies* $q = 1$ *if* $\mathcal{P}$ *contains no* `admit` *rules*

PROOF:
By induction on the derivation of $\Psi; \mathcal{P}; \Phi \rhd\!\!\rhd \breve{\sigma}\ \langle s, \hat{\rho}, q\rangle$ using Lemma 2.1

□

3 Symbolic Evaluator/VC Generator

In this section, we define a symbolic evaluator [NL98b] and a VC generator [Kin71] that interprets Palladium security properties. The symbolic evaluator is a program analysis based on symbolic executions. The VC generator uses the symbolic executions to produce a proposition that is true only if the program does not violate the security property. The symbolic evaluator restricts the control flow of the program to make the analysis tractable (there are safe programs with unsafe control flow that do not have valid VCs).

The principal difficulty in constructing this algorithm is to reconcile the unstructured, linear orientation of the security-property language with the structured, procedural orientation of the symbolic evaluator. The symbolic evaluator treats branches specially to ensure that the analysis terminates even if the program being analyzed does not. The security-property language, on the other hand, treats all operations uniformly to realize a simple programming model.

For each operational judgment of Section 2, we define a corresponding symbolic judgment. A symbolic state, for example, simulates many distinct concrete states. To ensure termination, each symbolic execution is only a fragment of a complete execution—the fragments correspond roughly to basic blocks. The VC generator constructs a VC from the set of symbolic executions that together cover the entire program.

$\mathrm{VC}_{\Psi,\mathcal{P},\Phi}$ is the VC of agent program Φ with respect to trusted system Ψ and security property $\mathcal{P}$. The soundness of the VC generator is asserted by the following theorem:

Theorem 3.1 (Soundness) $\Sigma_{\Psi,\Phi} \subseteq \mathcal{P}(\Sigma_{\Psi,\Phi})$ *if* $\models \mathrm{VC}_{\Psi,\mathcal{P},\Phi}$

PROOF:

for each $\sigma \in \Sigma_{\Psi,\Phi}$

$\Psi; \Phi \rhd\!\!\rhd \sigma$	$\mathrm{Agent}_{\Psi,\Phi}(\sigma)$	Def. $\Sigma_{\Psi,\Phi}$
$\Psi; \mathcal{P}; \Phi \rhd\!\!\rhd \breve{\sigma}$	$\lfloor \breve{\sigma} \rfloor = \sigma$	Lemma 4.40
$\sigma \in \mathcal{P}(\Sigma_{\Psi,\Phi})$		Def. $\mathcal{P}(\Sigma_{\Psi,\Phi})$

□

That is, the program execution set is a subset of the security-property execution set if the VC is valid. The soundness proof is developed in Section 4 as a series of lemmas, of which Lemma 4.40 is the conclusion.

In the remainder of this section, we specify the symbolic evaluator and the VC generator. We start from a basic judgment on transitions and work our way up to extended executions as in Section 2. Once the extended-execution judgment is defined, we define the VC generator.

3.1 Transition Symbolic Evaluation

The symbolic evaluator interprets PAL instructions with symbolic expressions for the contents of registers. In this section, we present symbolic inference rules that mimic the PAL operational semantics. We use the special turnstile $\tilde{\triangleright}$ to distinguish symbolic from concrete judgments.

Two annotations are interpreted by the symbolic evaluator, but are ignored by the concrete machine:

$$\text{Annotations} \quad Ann ::= \texttt{proc}\ P, P', X \mid \texttt{inv}\ P, X$$

$\texttt{proc}\ P, P', X$ is a procedure specification that must be the first instruction of each procedure. P is the precondition of the procedure, and P' is its postcondition. The precondition must hold before the procedure is called, and the postcondition will hold after it returns. The variable set X contains the machine registers and property registers that are changed by the procedure: all other registers are preserved.

$\texttt{inv}\ P, X$ is an invariant that must be the destination of every backwards branch. P holds each time the invariant is executed. X contains the machine registers and property registers that are changed by any cycle that includes the instruction: all other registers are preserved.

The symbolic contents of a register is an expression of the logic. *Symbolic* states t are pairs $\langle i, \eta \rangle$ where i is the instruction counter, and η is a register substitution. A register substitution is a total map from machine register identifiers to expressions[6]. We assign a variable to a register to let it stand for many possible values; specifications encode universal properties of these values. A symbolic state can be treated as a substitution using the notation θ_t:

$$\theta_{\langle i, \eta \rangle} = \eta[\texttt{pc} \mapsto \bar{i}]$$

Figure 20 contains inference rules for the symbolic transition relation. Procedure calls, loop invariants, and conditional branches are not covered by this relation because they have special steps. The judgment $\Phi \,\tilde{\triangleright}\, t \stackrel{\mathsf{m}}{\rightarrow} t'$ asserts that there is a transition from state t to state t' with respect to program Φ.

3.2 Steps and Specifications

Symbolic steps resemble concrete steps: $<\!\cdot t$ is a start step, $t \rightarrow t'$ is a transition step, and $t \cdot\!>$ is a stop step, but to delimit partial executions we introduce two new steps. A *head* step $\prec\!* \, t$ marks the start of a partial execution that has no start step (*i.e.*, it follows another partial execution). A *tail* step $t \, *\!\succ \langle p^{\mathsf{l}}_1, \ldots, p^{\mathsf{l}}_k \rangle$ marks the end of a partial execution that has no stop step; p^{l}_j is a *link specification* that determines a successor partial execution.

Link specifications enable us to treat transitions between symbolic executions uniformly and thereby evaluate the security property during symbolic

[6]Register substitutions can be systematically evaluated to register environments.

transformations. Link specifications contain context, transition, and abstraction specifications: the next paragraph outlines this relationship and a more detailed discussion follows. These specifications encode symbolic transformations that the VC generator can apply interpretively. The transformations typically replace the contents of certain registers with fresh variables.

Link specifications define the head states of *successor* executions that cover the possible paths the program after the current execution. A link specification is comprised of a transition specification and zero or more context specifications. A *transition specification* p^{t} specifies an abstraction operation and an optional transition; it is comprised of an abstraction specification and an optional target address. A register is *abstracted* when it is assigned a fresh variable[7]. An *abstraction specification* p^{a} specifies which machine registers and property registers are abstracted. A *context specification* p^{c} specifies a constraint for the "background" state of the symbolic evaluator; these constraints ensure that procedure and loop specifications are consistent.

The transition specification p^{t} of link specification $\langle p^{\mathsf{t}}, p_1^{\mathsf{c}}, \ldots, p_k^{\mathsf{c}} \rangle$ specifies the head state of a successor execution based on the current tail state. The context specifications $p_1^{\mathsf{c}}, \ldots p_k^{\mathsf{c}}$ constrain the background of the successor.

A context specification is comprised of an activation address and a transition specification. A context specification provides an implicit successor to any execution with a tail state matching the activation address. For the context specification $\langle i, p^{\mathsf{t}} \rangle$, p^{t} determines the implicit successor, and i is the activation address. For example, a procedure-call context specification is an abstraction of the caller to which the procedure will return.

Transition specifications come in four different varieties. $\mathtt{id}\langle p^{\mathsf{a}} \rangle$ specifies that the successor head state is an abstraction of the current tail state; p^{a} specifies the abstraction operation. $\mathtt{tol}\langle p^{\mathsf{a}}, i \rangle$ specifies that the successor head state results from a transition to address i after abstracting the tail state. $\mathtt{tor}\langle i, p^{\mathsf{a}} \rangle$ is similar, except that the transition comes before the abstraction operation. $\mathtt{tos}\langle i \rangle$ is a special case of $\mathtt{tol}\langle p^{\mathsf{a}}, i \rangle$ in which all machine registers are abstracted.

Abstraction specifications come in two different varieties. $\mathtt{all}\langle P, P' \rangle$ specifies that all machine registers and property registers are abstracted; abstracted registers are assigned variables that will be universally quantified in the final VC. P is required to hold in the unabstracted state, and P' is assumed to hold in the abstracted state. In practice, P is either P' or $\top$. In the first case, P checks the unabstracted state to ensure that P' will hold for the abstracted state. In the second case, P' is ensured by some other means. $\mathtt{some}\langle X, P, P' \rangle$ is similar to $\mathtt{all}\langle P, P' \rangle$, except that just the registers in X are abstracted.

We now employ specifications in deriving symbolic steps.

[7] Abstraction enables us to express general properties of specific program points because the variable is a "stand in" for many possible run-time values. Think of abstraction as changing the symbolic evaluator's perspective on a run-time value.

3.3 Step Symbolic Evaluation

Figure 21 contains inference rules for valid symbolic steps. $\Psi; \Phi \tilde{\triangleright} B$ asserts that B is a valid step with respect to Ψ and Φ. The step rules require that all transitions be within the program or to the trusted system, thus ensuring that no execution will be abort because of an undefined instruction address. All branches must be within a procedure so that procedure calls are identified by the `call` instruction. Note that rule $\mathtt{ma}_5^{\sim}$ checks against an unspecified security property $\mathcal{P}$: these checks should actually be part of the *extended* step rules, but are shown here in the interest of brevity.

We assume that the skip and agent relations resemble procedure calls. In particular, for a start step, `ra` must contain a corresponding stop address. In addition, each skip source `ra` must contain the corresponding target address:

$$\begin{array}{lll} \rho(\mathtt{ra}) = i' & \text{if} & \mathrm{Agent}_\Psi(\langle i, \rho\rangle, \langle i', \rho'\rangle) \\ \rho(\mathtt{ra}) = i' & \text{if} & \mathrm{Skip}_\Psi(\langle i, \rho\rangle, \langle i', \rho'\rangle) \end{array}$$

These assumptions ensure consistent calling conventions for agent entry points, internal procedures, and system calls. Finally, we require each start to be paired with at least one stop state by the Agent relation.

Two functions are used in the step rules. $\mathrm{Jump}_\Phi(i)$ is the set of addresses in the same procedure as address i, and $\mathrm{Ret}_\Phi(i_0)$ is the set of return-instruction addresses in procedure i_0:

$$\begin{array}{lll} i' \in \mathrm{Jump}_\Phi(i) & \text{iff} & i' \in \mathrm{Dom}_\Phi(i_0) \text{ for some } i_0 \text{ such that } i \in \mathrm{Dom}_\Phi(i_0) \\ i \in \mathrm{Ret}_\Phi(i_0) & \text{iff} & i \in \mathrm{Dom}_\Phi(i_0) \text{ and } \Phi_i = \mathtt{ret} \end{array}$$

The latter definition identifies the exit points of a procedure.

Our choice of link specifications deserves discussion. Conditional branches have a pair of links that follow both possibilities of the branch; each branch assumes that the corresponding conditional holds. System calls (*i.e.*, $\underline{n} \in \mathrm{Skip}(\Psi)$) abstract all machine registers; callee-save registers are declared by the security-property `scs` rule. The effects of system calls on the property registers are declared by the `eval` and `new` rules. Internal procedure calls (*i.e.*, $\underline{n} \in \mathrm{Proc}(\Phi)$) first transition to the procedure base address; the precondition is then checked and, after all registers are abstracted, the precondition is assumed and symbolic evaluation resumes in the callee. A context specification is constructed for each callee `ret` instruction from the postcondition. Upon reaching a return instruction, all *modified* registers are abstracted and the postcondition is assumed by the caller at the instruction following the call. Note that because all registers are abstracted, a given procedure need never be evaluated more than once (see Section 3.7). Invariants are simpler than procedure calls: only a single condition is checked and assumed, and only modified registers are abstracted; the single context specification applies to the head of the loop. A return instruction requires no action unless it triggers a stop—internal returns are handled by the context mechanism (see Section 3.7).

$$\frac{}{\Phi \mathbin{\tilde{\triangleright}} \langle i, \eta\rangle \overset{\mathsf{m}}{\to} \langle i \dotplus 1, \eta'\rangle}\ \mathsf{m}_1^{\sim}$$

Φ_i	η'
$r \leftarrow n$	$\eta[r \mapsto n]$
$r_1 \leftarrow r_2\ \mathit{eop}\ r_3$	$\eta[r_1 \mapsto \mathit{eop}(\eta(r_2), \eta(r_3))]$
$r_1 \leftarrow M[r_2]$	$\eta[r_1 \mapsto \mathbf{selw}(\eta(\mathtt{mem}), \eta(r_2))]$
$M[r_1] \leftarrow r_2$	$\eta[\mathtt{mem} \mapsto \mathbf{updw}(\eta(\mathtt{mem}), \eta(r_1), \eta(r_2))]$
$\mathtt{proc}\ P, P', X$	η

$$\frac{\Phi_i = \mathtt{ra} \leftarrow \mathtt{pc}\ \mathbf{addw}\ n \qquad \underline{n} \dotplus i \dotplus 1 \in \mathrm{Jump}_\Phi(i)}{\Phi \mathbin{\tilde{\triangleright}} \langle i, \eta\rangle \overset{\mathsf{m}}{\to} \langle i \dotplus 1, \eta[\mathtt{ra} \mapsto \overline{\underline{n} \dotplus i \dotplus 1}]\rangle}\ \mathsf{m}_3^{\sim}$$

Figure 20: PAL Symbolic Transition Relation ($\Phi \mathbin{\tilde{\triangleright}} t \overset{\mathsf{m}}{\to} t'$)

$$\frac{i_0 \in \mathrm{Start}(\Psi) \quad \eta(\mathtt{ra}) = \overline{i'} \quad i' \in \mathrm{Stop}(\Psi)}{\Psi; \Phi \mathbin{\tilde{\triangleright}} <\!\cdot \langle i_0, \eta\rangle}\ \mathsf{ma}_1^{\sim} \qquad \frac{\Phi \mathbin{\tilde{\triangleright}} \langle i, \eta\rangle \overset{\mathsf{m}}{\to} \langle i', \eta'\rangle \quad i' \in \mathrm{Jump}_\Phi(i)}{\Psi; \Phi \mathbin{\tilde{\triangleright}} \langle i, \eta\rangle \to \langle i', \eta'\rangle}\ \mathsf{ma}_2^{\sim}$$

$$\frac{i_0 \in \mathrm{Start}(\Psi) \quad i \in \mathrm{Ret}_\Phi(i_0) \quad \eta(\mathtt{ra}) = \overline{i'} \quad i' \in \mathrm{Stop}(\Psi)}{\Psi; \Phi \mathbin{\tilde{\triangleright}} \langle i, \eta\rangle \cdot\!>}\ \mathsf{ma}_3^{\sim} \qquad \frac{i \in \mathrm{Dom}(\Phi)}{\Psi; \Phi \mathbin{\tilde{\triangleright}} \prec\!* \langle i, \eta\rangle}\ \mathsf{ma}_4^{\sim}$$

$$\frac{}{\Psi; \Phi \mathbin{\tilde{\triangleright}} \langle i, \eta\rangle *\!\succ \langle p^{\mathsf{l}}_1, \ldots, p^{\mathsf{l}}_k\rangle}\ \mathsf{ma}_5^{\sim}$$

Φ_i	Case	$p^{\mathsf{l}}_1 \ \ldots\ p^{\mathsf{l}}_k$
$\mathtt{cond}\ \mathit{cop}\ r, n$	$\underline{n} \dotplus i \dotplus 1, i \dotplus 1 \in \mathrm{Jump}_\Phi(i)$ $\Phi_{\underline{n} \dotplus i \dotplus 1} = \mathtt{inv}\ \ldots$ if $\underline{n} \dotplus i \dotplus 1 < i \dotplus 1$	$\langle \mathtt{tol}\langle \mathtt{some}\langle \emptyset, \top, \mathit{cop}(r)\rangle, \underline{n} \dotplus i \dotplus 1\rangle\rangle$ $\langle \mathtt{tol}\langle \mathtt{some}\langle \emptyset, \top, (\neg\mathit{cop})(r)\rangle, i \dotplus 1\rangle\rangle$
$\mathtt{call}\ n$	$\underline{n} \in \mathrm{Skip}(\Psi)$ $i \dotplus 1 \in \mathrm{Jump}_\Phi(i)$ $\eta(\mathtt{ra}) = \overline{i \dotplus 1}$	$\langle \mathtt{tos}\langle i \dotplus 1\rangle\rangle$
$\mathtt{call}\ n$	$\underline{n} \in \mathrm{Proc}(\Phi)$ $\quad \mathtt{ra} \notin X$ $i \dotplus 1 \in \mathrm{Jump}_\Phi(i)$ $\quad \eta(\mathtt{ra}) = \overline{i \dotplus 1}$ $\Phi_{\underline{n}} = \mathtt{proc}\ P, P', X$ $\quad \mathcal{P} \vdash P$ spec $\mathcal{P} \vdash P'$ spec	$\langle \mathtt{tor}\langle \underline{n}, \mathtt{all}\langle P, P\rangle\rangle, \langle i_1, p^{\mathsf{t}}\rangle, \ldots, \langle i_{k'}, p^{\mathsf{t}}\rangle\rangle$ where $\{i_1, \ldots, i_{k'}\} = \mathrm{Ret}_\Phi(\underline{n})$ and $p^{\mathsf{t}} = \mathtt{tol}\langle \mathtt{some}\langle X, \top, P'\rangle, i \dotplus 1\rangle$
$\mathtt{ret}$	$\eta(\mathtt{ra}) \neq n$	
$\mathtt{inv}\ P, X$	$i \dotplus 1 \in \mathrm{Jump}_\Phi(i)$ $\mathcal{P} \vdash P$ spec	$\langle p^{\mathsf{t}}, \langle i, p^{\mathsf{t}}\rangle\rangle$ where $p^{\mathsf{t}} = \mathtt{tol}\langle \mathtt{some}\langle X, P, P\rangle, i \dotplus 1\rangle$

Figure 21: PAL Symbolic Step Derivation ($\Psi; \Phi \mathbin{\tilde{\triangleright}} B$)

The definition of valid steps enables us to define security-property evaluation.

3.4 Security-Property Symbolic Evaluation

In this section, we present symbolic evaluation rules that mimic the concrete security-property rules. Evaluating security properties enables us to evaluate extended steps, which, in turn, lead to extended executions.

The symbolic evaluator accumulates requirements and assumptions in a *proposition context*; a proposition context can be thought of as partial VC. Proposition contexts are found in security-property states as well as extended states. A proposition context Q resembles a proposition P, except that Q contains a single "hole" $\bullet$ in its rightmost position. A proposition context can be applied to a proposition or another proposition context—application substitutes the operand for the hole:

$$\begin{array}{ll@{\qquad}ll}
\bullet(Q) & = Q & \bullet(P) & = P \\
(P \wedge Q_1)(Q_2) & = P \wedge Q_1(Q_2) & (P_1 \wedge Q)(P_2) & = P_1 \wedge Q(P_2) \\
(P \supset Q_1)(Q_2) & = P \supset Q_1(Q_2) & (P_1 \supset Q)(P_2) & = P_1 \supset Q(P_2) \\
(\forall x : \tau . Q_1)(Q_2) & = \forall x : \tau . Q_1(Q_2) & (\forall x : \tau . Q)(P) & = \forall x : \tau . Q(P)
\end{array}$$

A Q applied to a P is a P; a Q applied to another Q is a also a Q.

The bound variables of a proposition context are exactly those that include the hole in their scope:

$$\begin{array}{ll}
\mathrm{BV}(\bullet) & = \emptyset \\
\mathrm{BV}(P \wedge Q) & = \mathrm{BV}(Q) \\
\mathrm{BV}(P \supset Q) & = \mathrm{BV}(Q) \\
\mathrm{BV}(\forall x : \tau . Q) & = \{x\} \cup \mathrm{BV}(Q)
\end{array}$$

Note that variables bound by component propositions are not considered.

Symbolic security-property states $\hat{t}$ are triples $\langle \mathcal{P}, \hat{\eta}, Q \rangle$ where $\mathcal{P}$ is the security property being evaluated, $\hat{\eta}$ is the current property-register substitution, and Q is the current proposition context. A property-register substitution $\hat{\eta}$ maps property registers to expressions, and is total on the property registers. Q accumulates requirements and assumptions as the security property is evaluated. For example, applying Q to $P \wedge \bullet$ accumulates the requirement P: this simulates the effect of a requirement rule.

Figure 22 contains inference rules for the security-property transition relation. The judgment $\Phi; B \,\tilde{\triangleright}\, \hat{t} \xrightarrow{\mathsf{s}} \hat{t}'$ asserts that there is a transition from state $\hat{t}$ to state $\hat{t}'$ with respect to program Φ and symbolic step B. The large-transition judgment $\Phi; B \,\tilde{\triangleright}\, \hat{t} \xrightarrow{\mathsf{s}*} \hat{t}'$ represents the reflexive, transitive closure of the transition relation.

Requirement and assumption rules accumulate propositions into the current proposition context. An evaluation rule updates the current property-register substitution. The `new` rule picks a fresh variable for the target register and universally quantifies it in the proposition context. All variables generated so far are bound in the current proposition context.

$$\frac{}{\Phi; B \,\tilde{\triangleright} \langle \texttt{reg}\ \hat{r} : \tau; \mathcal{P}, \hat{\eta}, Q\rangle \xrightarrow{\mathsf{s}} \langle \mathcal{P}, \hat{\eta}, Q\rangle}\ \mathsf{s}_1^{\sim}$$

$$\frac{\Phi \,\tilde{\triangleright}\, A^{\square}; B \overset{\mathsf{pa}}{\not\to}}{\Phi; B \,\tilde{\triangleright} \langle \mathcal{P}, \hat{\eta}, Q\rangle \xrightarrow{\mathsf{s}} \langle \mathcal{P}', \hat{\eta}, Q\rangle}\ \mathsf{s}_2^{\sim}$$

$\mathcal{P}$
`require` $A^{\square} \Rightarrow P; \mathcal{P}'$
`admit` $A^{\square} \Rightarrow P; \mathcal{P}'$
`eval` $A^{\square} \Rightarrow \hat{r} := E; \mathcal{P}'$
`new` $A^{\square} \Rightarrow \hat{r} : \tau; \mathcal{P}'$

$$\frac{\Phi \,\tilde{\triangleright}\, A^{\square}; B \xrightarrow{\mathsf{pa}} \theta_1; \theta_2}{\Phi; B \,\tilde{\triangleright} \langle \mathcal{P}, \hat{\eta}, Q\rangle \xrightarrow{\mathsf{s}} \langle \mathcal{P}', \hat{\eta}', Q'\rangle}\ \mathsf{s}_3^{\sim}$$

$\mathcal{P}$	$\hat{\eta}'$	Q'
`require` $A^{\square} \Rightarrow P; \mathcal{P}'$	$\hat{\eta}$	$Q((\theta_2 \cup \hat{\eta})(\theta_1(P)) \wedge \bullet)$
`admit` $A^{\square} \Rightarrow P; \mathcal{P}'$	$\hat{\eta}$	$Q((\theta_2 \cup \hat{\eta})(\theta_1(P)) \supset \bullet)$
`eval` $A^{\square} \Rightarrow \hat{r} := E; \mathcal{P}'$	$\hat{\eta}[\hat{r} \mapsto (\theta_2 \cup \hat{\eta})(\theta_1(E))]$	Q
`new` $A^{\square} \Rightarrow \hat{r} : \tau; \mathcal{P}'$	$\hat{\eta}[\hat{r} \mapsto y]$ $y \notin \mathrm{BV}(Q)$ $y \notin \mathit{XReg} \cup \mathit{PReg}$	$Q(\forall y : \tau . \bullet)$

Figure 22: Security-Property Symbolic Transition Relation ($\Phi; B \,\tilde{\triangleright}\, \hat{t} \xrightarrow{\mathsf{s}} \hat{t}'$)

$$\frac{}{\Phi; B \,\tilde{\triangleright}\, \hat{t} \xrightarrow{\mathsf{s*}} \hat{t}}\ \mathsf{s*}_1^{\sim} \qquad \frac{\Phi; B \,\tilde{\triangleright}\, \hat{t}_0 \xrightarrow{\mathsf{s*}} \hat{t} \quad \Phi; B \,\tilde{\triangleright}\, \hat{t} \xrightarrow{\mathsf{s}} \hat{t}'}{\Phi; B \,\tilde{\triangleright}\, \hat{t}_0 \xrightarrow{\mathsf{s*}} \hat{t}'}\ \mathsf{s*}_2^{\sim}$$

Figure 23: Security-Property Symbolic Large-Transition Relation ($\Phi; B \,\tilde{\triangleright}\, \hat{t} \xrightarrow{\mathsf{s*}} \hat{t}'$)

With security-property evaluation defined, we can now extend symbolic states.

3.5 Extended-Step Symbolic Evaluation

To evaluate symbolic executions, we first need to extend symbolic steps. Extending symbolic states is similar to extending concrete states: extended states are triples $\langle t, \hat{\eta}, Q\rangle$ where t is the state being extended, $\hat{\eta}$ is the current property-register substitution, and Q is the current proposition context. The property-register substitution and proposition context carry over from one extended state to another by evaluating the security property for each step. An extended state can be treated as a substitution using the notation $\theta_{\check{t}}$:

$$\theta_{\langle t,\hat{\eta},Q\rangle} = \theta_t \cup \hat{\eta}$$

Figure 24 contains inference rules for valid extended symbolic steps; the judgment $\Psi; \mathcal{P}; \Phi \,\tilde{\triangleright}\, \check{B}$ asserts that $\check{B}$ is a valid extended step with respect to Ψ, $\mathcal{P}$, and Φ. The transition rule is a direct translation of the concrete transition rule. The start rule resembles a procedure call, abstracting all registers and placing a return address in `ra`. In this rule, i_0 is the base address of an entry point into the agent. Other rules are transcriptions of unextended rules.

The initial register substitution η_* maps each machine register to a distinct variable. The initial property-register substitution $\hat{\eta}_{\mathcal{P}}$ maps each property register of $\mathcal{P}$ to a distinct variable not in η_*. The null substitution $\hat{\eta}_*$ maps all property registers to zero:

$$\eta_* = \mathrm{Gen}_{\emptyset}(\mathit{Reg}) \qquad \hat{\eta}_{\mathcal{P}} = \hat{\eta}_* \cup \mathrm{Gen}_{\mathrm{FV}(\eta_*)}(\mathrm{dom}\,\Gamma_{\mathcal{P}}) \qquad \hat{\eta}_*(\hat{r}) = 0$$

These substitutions appear in initial states.

The extended step rules are based on abstraction functions. $\mathrm{Gen}_{X_0}(X)$ is a substitution that maps each variable in X to a fresh variable not in X_0; implementations of this function *must* be idempotent:

$$\begin{aligned}\mathrm{Gen}_{X_0}(X) = \theta \quad & \\ & \text{where } \mathrm{dom}\,\theta = X \\ & \text{and } \theta(x) = y \text{ such that } y \notin X_0 \cup \mathit{XReg} \cup \mathit{PReg} \\ & \text{and } \theta(x_1) \neq \theta(x_2) \text{ if } x_1 \neq x_2\end{aligned}$$

$\mathrm{Abs}_{\mathcal{P}}(\theta_1, \theta_2)$ introduces universal quantifiers for two substitutions. This function is based on $\mathrm{Abs}_{\Gamma}(\theta)$, which constructs a string of universal quantifiers for all variables in the range of θ; the context Γ (see Appendix B) assigns types to these variables:

$$\begin{aligned}\mathrm{Abs}_{\mathcal{P}}(\theta_1, \theta_2) &= (\mathrm{Abs}_{\Gamma_{\mathit{Reg}}}(\theta_1))(\mathrm{Abs}_{\Gamma_{\mathcal{P}}}(\theta_2)) \\ \mathrm{Abs}_{\Gamma}(\theta) &= \forall y_1 : \tau_1 \ldots \forall y_k : \tau_k.\bullet \\ & \quad \text{where } y_1, \ldots, y_k = \theta(x_1), \ldots, \theta(x_k) \\ & \quad \text{and } x_1 : \tau_1, \ldots, x_k : \tau_k \in \Gamma \\ & \quad \text{and } \{x_1, \ldots, x_k\} = \{x \in \mathrm{dom}\,\theta \cap \mathrm{dom}\,\Gamma \mid \theta(x) \in \mathit{Var}\}\end{aligned}$$

$$\frac{\Psi;\Phi \,\tilde{\triangleright}\, {<}\cdot\langle i_0,\eta\rangle \quad \eta = \eta_*[\mathtt{ra} \mapsto \overline{i'}] \quad i' \in \mathrm{Stop}(\Psi) \qquad \Phi;{<}\cdot\langle i_0,\eta\rangle \,\tilde{\triangleright}\langle \mathcal{P},\hat{\eta}_{\mathcal{P}},\mathrm{Abs}_{\mathcal{P}}(\eta,\hat{\eta}_{\mathcal{P}})\rangle \xrightarrow{\mathsf{s*}} \langle\cdot,\hat{\eta},Q\rangle}{\Psi;\mathcal{P};\Phi \,\tilde{\triangleright}\, {<}\cdot\langle\langle i_0,\eta\rangle,\hat{\eta},Q\rangle}\ \mathsf{ea}_1^{\sim}$$

$$\frac{\Psi;\Phi \,\tilde{\triangleright}\, t \to t' \quad \Phi; t \to t' \,\tilde{\triangleright}\langle \mathcal{P},\hat{\eta},Q\rangle \xrightarrow{\mathsf{s*}} \langle\cdot,\hat{\eta}',Q'\rangle}{\Psi;\mathcal{P};\Phi \,\tilde{\triangleright}\langle t,\hat{\eta},Q\rangle \to \langle t',\hat{\eta}',Q'\rangle}\ \mathsf{ea}_2^{\sim} \qquad \frac{\Psi;\Phi \,\tilde{\triangleright}\, t\cdot{>}}{\Psi;\mathcal{P};\Phi \,\tilde{\triangleright}\langle t,\hat{\eta},Q\rangle\cdot{>}}\ \mathsf{ea}_3^{\sim}$$

$$\frac{\Psi;\Phi \,\tilde{\triangleright} \prec\!\!* \, t}{\Psi;\mathcal{P};\Phi \,\tilde{\triangleright} \prec\!\!* \,\langle t,\hat{\eta},Q\rangle}\ \mathsf{ea}_4^{\sim} \qquad \frac{\Psi;\Phi \,\tilde{\triangleright}\, t \ast\!\!\succ \langle p_1^{!},\ldots,p_k^{!}\rangle}{\Psi;\mathcal{P};\Phi \,\tilde{\triangleright}\langle t,\hat{\eta},Q\rangle \ast\!\!\succ \langle p_1^{!},\ldots,p_k^{!}\rangle}\ \mathsf{ea}_5^{\sim}$$

Figure 24: Extended Symbolic Step Derivation ($\Psi;\mathcal{P};\Phi \,\tilde{\triangleright}\, \check{B}$)

Finally, $\mathrm{Gen}_{\mathcal{P},X}(\check{s})$ applies Gen to each substitution of a state and accumulates the resulting universal quantifiers:

$$\begin{aligned}\mathrm{Gen}_{\mathcal{P},X}(\langle\langle i,\eta\rangle,\hat{\eta},Q\rangle) = {} & \langle\langle i,\eta\cup\theta_1\rangle,\hat{\eta}\cup\theta_2,Q(\mathrm{Abs}_{\mathcal{P}}(\theta_1,\theta_2))\rangle \\ & \text{where } \theta_1 = \mathrm{Gen}_{\mathrm{BV}(Q)}(X \cap \mathit{Reg}) \\ & \text{and } \theta_2 = \mathrm{Gen}_{\mathrm{BV}(Q)\cup\mathrm{FV}(\theta_1)}(X \cap \mathrm{dom}\,\Gamma_{\mathcal{P}})\end{aligned}$$

Now we can derive extended symbolic executions—note that we have no use for unextended symbolic executions.

3.6 Extended-Execution Symbolic Evaluation

Figure 25 contains inference rules for valid extended symbolic executions. The judgment $\Psi;\mathcal{P};\Phi \,\tilde{\triangleright\!\!\triangleright}\, \check{\pi}$ asserts that $\check{\pi}$ is a valid symbolic execution with respect to Ψ, $\mathcal{P}$, and Φ. The start rule resembles a procedure call because we assume that the trusted system uses the standard calling convention. The link-specification discussion for an internal procedure call (see Section 3.3) applies to this rule as well, except that there is no extra transition. In this rule, i_0 is the base address of an external procedure. Other inference rules are transcriptions of concrete inference rules.

3.7 VC Generator

The VC generator derives a proposition from a complete set of symbolic executions that is true only if the security property is satisfied. This is accomplished by combining the partial VCs of the symbolic executions. $\mathrm{VC}_{\Psi,\mathcal{P},\Phi}$, the VC of Φ, is defined in Figure 27. A complete execution set is derived by transitive closure from the start addresses of the trusted system.

An execution $\check{\sigma}$ of Φ can be divided into fragments that correspond to the symbolic executions of Φ. This correspondence is developed precisely Section 4;

$$
\frac{\begin{array}{c}\Psi;\mathcal{P};\Phi \mathrel{\tilde{\triangleright}} <\!\cdot\langle\langle i,\eta\rangle,\hat{\eta},Q\rangle \quad \Phi_i = \texttt{proc}\ P,P',X \\ \texttt{ra} \notin X \quad \mathcal{P} \vdash P\ \mathsf{spec} \quad \mathcal{P} \vdash P'\ \mathsf{spec} \\ \{i_1,\ldots,i_k\} = \mathrm{Ret}_{\Phi}(i) \quad p^{\mathsf{t}} = \texttt{id}\langle\texttt{some}\langle X,\top,P'\rangle\rangle\end{array}}{\Psi;\mathcal{P};\Phi \mathrel{\widetilde{\triangleright\!\triangleright}} <\!\cdot\ \langle\langle i,\eta\rangle,\hat{\eta},Q\rangle \mathrel{*\!\succ} \langle\langle\texttt{id}\langle\texttt{all}\langle P,P\rangle\rangle,\langle i_1,p^{\mathsf{t}}\rangle,\ldots,\langle i_k,p^{\mathsf{t}}\rangle\rangle\rangle}\ \mathsf{ee}_1^{\sim}
$$

$$
\frac{\Psi;\mathcal{P};\Phi \mathrel{\tilde{\triangleright}} \prec\!*\,\check{t}}{\Psi;\mathcal{P};\Phi \mathrel{\widetilde{\triangleright\!\triangleright}} \prec\!*\,\check{t}}\ \mathsf{ee}_2^{\sim}
\qquad
\begin{array}{|l|}\hline \dfrac{\Psi;\mathcal{P};\Phi \mathrel{\widetilde{\triangleright\!\triangleright}} \breve{\pi}\,\check{t} \quad \Psi;\mathcal{P};\Phi \mathrel{\tilde{\triangleright}} \check{B}}{\Psi;\mathcal{P};\Phi \mathrel{\widetilde{\triangleright\!\triangleright}} \breve{\pi}\,\check{B}}\ \mathsf{ee}_3^{\sim} \\ \hline \check{B} \\ \hline \check{t} \to \check{t}' \\ \hline \check{t}\!\cdot\!> \\ \hline \check{t} \mathrel{*\!\succ} \langle p_1^{|},\ldots,p_k^{|}\rangle \\ \hline \end{array}
$$

Figure 25: Extended Symbolic Execution Derivation ($\Psi;\mathcal{P};\Phi \mathrel{\widetilde{\triangleright\!\triangleright}} \breve{\pi}$)

for now, think of a fragment as the value of some $\breve{\pi}$, given a suitable environment. A single $\breve{\pi}$ may correspond to many distinct fragments of a given $\breve{\sigma}$.

3.7.1 Terminology

Each symbolic execution $\breve{\pi}$ has a set of *successor* executions that represent symbolic evaluation after $\breve{\pi}$. If $\breve{\pi}$ corresponds to some fragment of $\breve{\sigma}$, then some successor $\breve{\pi}'$ of $\breve{\pi}$ corresponds to an adjacent fragment of $\breve{\sigma}$. The tail of $\breve{\pi}$ may correspond to the same concrete state as the head of $\breve{\pi}'$, or there may be an intervening transition. The successor graph may contain cycles: cycles enable us to account for an arbitrarily long $\breve{\sigma}$. For example, we might break down $\breve{\sigma}$ as follows:

$$
\overbrace{<\!\cdot\,\check{s}_0}^{\breve{\pi}_0}\ \overbrace{\check{s}_0 \to \check{s}_1 \to \check{s}_2 \to \check{s}_3 \to \check{s}_4}^{\breve{\pi}_1} \to \overbrace{\check{s}_5 \to \check{s}_6 \to \check{s}_7}^{\breve{\pi}_2} \to \overbrace{\check{s}_8 \to \check{s}_9 \to \check{s}_{10}}^{\breve{\pi}_2} \to \ldots
$$

$\breve{\pi}_0$ corresponds to the start step; $\breve{\pi}_1$ shares $\check{s}_0$ with $\breve{\pi}_0$; $\breve{\pi}_2$ is a loop body, so it repeats for several successive fragments.

The VC generator starts with symbolic start steps that are trivial executions that directly lead to successor states. These head states are in turn evaluated, resulting in more executions, each of which specify *their* successors. This process continues until there are no fresh executions to evaluate.

Successors are derived from abstract specifications to ensure that the set of symbolic executions is finite. For example, all procedure-call successors are identical for a given procedure: the head state is based only on the procedure's precondition to guarantee that all calls will converge.

Before discussing the VC generator itself, we introduce two concepts:

- A *trace context* $\mathcal{C}$ is a partial function from addresses to propositions. Trace contexts comprise the background state of the symbolic evaluator and ensure that specifications are consistent. If $\mathcal{C}$ maps i to P, then evaluation stops at program counter i and P is required to hold in place of further symbolic evaluation. For example, when a procedure is evaluated, its trace context contains its postcondition at each return instruction. When a return is evaluated, we accumulate the postcondition as a requirement instead of simulating a return.

- A *trace schedule* $\mathcal{S}$ is a set of trace context/symbolic execution pairs. If $\langle \mathcal{C}, \breve{\pi} \rangle$ is in $\mathcal{S}$, then $\breve{\pi}$ either has been evaluated, or is scheduled to be evaluated with background $\mathcal{C}$.

The VC generator maintains a set of completed traces $\mathcal{S}_1$ and a set of scheduled traces $\mathcal{S}_2$; at each iteration, a trace is selected from $\mathcal{S}_2 \setminus \mathcal{S}_1$ and evaluated.

The VC of a program is a conjunction of closed propositions. Each symbolic execution is associated with a subterm proposition based on its proposition contexts: each proposition context is a *prefix* of the subterm; Q is a prefix of P if there is some P' such that $P = Q(P')$. Thus, a subterm proposition is the limit of a series of proposition contexts: this is the VC of an *execution.* Two executions are *independent* when they have different VCs[8]. The successor of an execution $\breve{\pi}$ may be independent of $\breve{\pi}$: this happens when all registers are abstracted or when a trace context is matched; in these cases, the current VC is complete and a new VC is started. Note that the start and stop steps of a given entry point have the same VC.

We now describe the VC generator following the order of presentation in Figure 26 and Figure 27. Figure 26 contains definitions for specification evaluation; we *evaluate* a specification with respect to a trace context/tail state pair to reach a (successor) trace context/head state pair. Figure 27 contains definitions for generating VCs from trace schedules.

3.7.2 Specification Evaluation

$\mathrm{PreVC}_{\mathcal{P}}(\check{t}, p^{\mathsf{a}})$ is the VC that results from evaluating abstraction specification p^{a} with respect to symbolic state $\check{t}$; the result is the special symbol `open` if the VC is not closed. Other PreVC functions are similar except that link specifications evaluate to a sequence of VCs—one for each successor specification. A VC is closed when all registers are abstracted in its successor, thus making the successor independent.

$\mathrm{Eval}_{\mathcal{P}}(\langle \mathcal{C}, \check{t} \rangle, p^{\mathsf{a}})$ is the successor trace context/state pair of $\langle \mathcal{C}, \check{t} \rangle$ according to the abstraction specification p^{a}. The successor state will be the head state of a new symbolic execution, using the successor trace context as background. Other Eval functions are similar except that link specifications evaluate to a sequence of trace context/state pairs.

$\mathrm{Need}_{\mathcal{P}}(\check{t}, p^{\mathsf{a}})$ is a proposition that implies the assumptions introduced by the abstraction specification p^{a}. It additionally implies that the registers not abstracted by p^{a} equal the corresponding registers of $\check{t}$.

[8]The bound variables of independent executions have disjoint scopes.

$\text{In}_{\mathcal{P},\Phi}(\check{t}, p^{\mathsf{t}})$ and $\text{Out}_{\mathcal{P},\Phi}(\check{t}, p^{\mathsf{t}})$ partially evaluate a transition specification with respect to $\check{t}$. In evaluates transitions that occur *before* abstraction, and Out evaluates transitions that occur *after* abstraction. $\text{ASp}(p^{\mathsf{t}})$ is the abstraction-specification subterm of transition specification p^{t}.

3.7.3 VC Generation

$\text{Conj}_{\check{t}}(\{P_1, \ldots, P_k\})$ is the conjunction of $P_1, \ldots, P_k$ except that their common prefix is factored according to the proposition context of $\check{t}$. This function "rejoins" the VCs of multiple control paths. $\text{Trace}_{\Psi,\mathcal{P},\Phi}(\{\langle \mathcal{C}_1, \check{t}_1\rangle, \ldots, \langle \mathcal{C}_k, \check{t}_k\rangle\})$ is a trace schedule for a set of trace context/state pairs. Each execution of the result is derived from a head step for the corresponding state.

$\text{ClosedVCS}_{\mathcal{P},\Phi,\check{t}}(p^{\mathsf{l}},)$ is the set of closed VCs that result from evaluating p^{l} with respect to $\check{t}$. $\text{Closed}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C}, \check{t}\rangle, p^{\mathsf{l}})$ is the trace schedule containing of the *independent* successors that result from evaluating p^{l} with respect to $\langle \mathcal{C}, \check{t}\rangle$; similarly, $\text{Open}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C}, \check{t}\rangle, p^{\mathsf{l}})$ is the trace schedule of *non-independent* successors.

$\text{VC}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C}, \breve{\pi}\rangle)$ is the VC of $\breve{\pi}$ with background $\mathcal{C}$, and is a closed proposition that is a subterm of the complete VC. $\text{VC}_{\Psi,\mathcal{P},\Phi}(\mathcal{S})$ is the VC of trace schedule $\mathcal{S}$ (the conjunction of the VCs of the members of $\mathcal{S}$).

$\text{Next}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C}, \breve{\pi}\rangle)$ is the trace schedule of the *independent* successors of $\breve{\pi}$ with background $\mathcal{C}$. $\text{Fix}_{\Psi,\mathcal{P},\Phi}(\mathcal{S}_1, \mathcal{S}_2)$ is the fixed point of $\mathcal{S}_2$, given that $\mathcal{S}_1$ and its successors are already in $\mathcal{S}_2$; the fixed point is found when $\mathcal{S}_1 = \mathcal{S}_2$. The fixed point of a trace schedule $\mathcal{S}$ is those independent executions that are reachable from members of $\mathcal{S}$.

$$
\begin{array}{ll}
\mathrm{PreVC}_{\mathcal{P}}(\langle t, \hat{\eta}, Q\rangle, \mathtt{all}\langle P, P'\rangle) & = Q(\theta_{\langle t,\hat{\eta},Q\rangle}(P)) \\
\mathrm{PreVC}_{\mathcal{P}}(\check{t}, \mathtt{some}\langle X, P, P'\rangle) & = \mathtt{open}
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{Eval}_{\mathcal{P}}(\langle \mathcal{C}, \langle\langle i, \eta\rangle, \hat{\eta}, Q\rangle\rangle, \mathtt{all}\langle P, P'\rangle) & = \langle \emptyset, \langle\langle i, \eta_*\rangle, \hat{\eta}_{\mathcal{P}}, (\mathrm{Abs}_{\mathcal{P}}(\eta_*, \hat{\eta}_{\mathcal{P}}))(\theta_{\langle\langle i,\eta_*\rangle,\hat{\eta}_{\mathcal{P}},\bullet\rangle}(P') \supset \bullet)\rangle\rangle \\
\mathrm{Eval}_{\mathcal{P}}(\langle \mathcal{C}, \langle t, \hat{\eta}, Q\rangle\rangle, \mathtt{some}\langle X, P, P'\rangle) & = \langle \mathcal{C}, \langle t', \hat{\eta}', Q'(\theta_{\langle t',\hat{\eta}',Q'\rangle}(P') \supset \bullet)\rangle\rangle \\
 & \quad \text{where } \langle t', \hat{\eta}', Q'\rangle = \mathrm{Gen}_{\mathcal{P},X}(\langle t, \hat{\eta}, Q(\theta_{\langle t,\hat{\eta},Q\rangle}(P) \wedge \bullet)\rangle)
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{Need}_{\mathcal{P}}(\check{t}, \mathtt{all}\langle P, P'\rangle) & = P' \\
\mathrm{Need}_{\mathcal{P}}(\check{t}, \mathtt{some}\langle X, P, P'\rangle) & = \bigwedge_{x \in (\mathit{Reg} \cup \mathrm{dom}\,\Gamma_{\mathcal{P}}) \smallsetminus X} x = \theta_{\check{t}}(x) \wedge P'
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{In}_{\mathcal{P},\Phi}(\langle\langle i, \eta\rangle, \hat{\eta}, Q\rangle, \mathtt{tor}\langle i', p^{\mathsf{a}}\rangle) & = \langle\langle i', \eta\rangle, \hat{\eta}', Q'\rangle \quad \text{if } \Phi; \langle i, \eta\rangle \to \langle i', \eta\rangle \,\tilde{\triangleright}\, \langle \mathcal{P}, \hat{\eta}, Q\rangle \xrightarrow{\mathsf{s}*} \langle \cdot, \hat{\eta}', Q'\rangle \\
\mathrm{In}_{\mathcal{P},\Phi}(\check{t}, p^{\mathsf{t}}) & = \check{t} \quad \text{if } p^{\mathsf{t}} \neq \mathtt{tor}\ldots
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{Out}_{\mathcal{P},\Phi}(\langle\langle i, \eta\rangle, \hat{\eta}, Q\rangle, \mathtt{tol}\langle p^{\mathsf{a}}, i'\rangle) & = \langle\langle i', \eta\rangle, \hat{\eta}', Q'\rangle \quad \text{if } \Phi; \langle i, \eta\rangle \to \langle i', \eta\rangle \,\tilde{\triangleright}\, \langle \mathcal{P}, \hat{\eta}, Q\rangle \xrightarrow{\mathsf{s}*} \langle \cdot, \hat{\eta}', Q'\rangle \\
\mathrm{Out}_{\mathcal{P},\Phi}(\langle\langle i, \eta\rangle, \hat{\eta}, Q\rangle, \mathtt{tos}\langle i'\rangle) & = \langle\langle i', \eta'\rangle, \hat{\eta}', Q'\rangle \quad \text{if } \Phi; \langle i, \eta\rangle \to \langle i', \eta'\rangle \,\tilde{\triangleright}\, \langle \mathcal{P}, \hat{\eta}, Q(\mathrm{Abs}_{\Gamma_{\mathit{Reg}}}(\theta))\rangle \xrightarrow{\mathsf{s}*} \langle \cdot, \hat{\eta}', Q'\rangle \\
 & \quad \text{where } \eta' = \eta \cup \theta \text{ and } \theta = \mathrm{Gen}_{\mathrm{BV}(Q)}(\mathit{Reg}) \\
\mathrm{Out}_{\mathcal{P},\Phi}(\check{t}, p^{\mathsf{t}}) & = \check{t} \quad \text{if } p^{\mathsf{t}} = \mathtt{id}\ldots \text{ or } p^{\mathsf{t}} = \mathtt{tor}\ldots
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{ASp}(\mathtt{tos}\langle i\rangle) & = \mathtt{some}\langle \emptyset, \top, \top\rangle \\
\mathrm{ASp}(p^{\mathsf{t}}) & = p^{\mathsf{a}} \quad \text{if } p^{\mathsf{t}} = \mathtt{id}\langle p^{\mathsf{a}}\rangle \text{ or } p^{\mathsf{t}} = \mathtt{tol}\langle p^{\mathsf{a}}, i\rangle \text{ or } p^{\mathsf{t}} = \mathtt{tor}\langle i, p^{\mathsf{a}}\rangle
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{PreVC}_{\mathcal{P},\Phi}(\check{t}, p^{\mathsf{t}}) & = \mathrm{PreVC}_{\mathcal{P}}(\mathrm{In}_{\mathcal{P},\Phi}(\check{t}, p^{\mathsf{t}}), \mathrm{ASp}(p^{\mathsf{t}})) \\
\mathrm{Eval}_{\mathcal{P},\Phi}(\langle \mathcal{C}, \check{t}\rangle, p^{\mathsf{t}}) & = \langle \mathcal{C}', \mathrm{Out}_{\mathcal{P},\Phi}(\check{t}', p^{\mathsf{t}})\rangle \quad \text{where } \langle \mathcal{C}', \check{t}'\rangle = \mathrm{Eval}_{\mathcal{P}}(\langle \mathcal{C}, \mathrm{In}_{\mathcal{P},\Phi}(\check{t}, p^{\mathsf{t}})\rangle, \mathrm{ASp}(p^{\mathsf{t}})) \\
\mathrm{Need}_{\mathcal{P},\Phi}(\check{t}, p^{\mathsf{t}}) & = \mathrm{Need}_{\mathcal{P}}(\check{t}, \mathrm{ASp}(p^{\mathsf{t}}))
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{PreVC}_{\mathcal{P},\Phi}(\langle\langle i, \eta\rangle, \hat{\eta}, Q\rangle, \langle i', p^{\mathsf{t}}\rangle) & = \mathrm{PreVC}_{\mathcal{P},\Phi}(\langle\langle i', \eta\rangle, \hat{\eta}, Q\rangle, p^{\mathsf{t}}) \\
\mathrm{Eval}_{\mathcal{P},\Phi}(\langle \mathcal{C}, \langle\langle i, \eta\rangle, \hat{\eta}, Q\rangle\rangle, \langle i', p^{\mathsf{t}}\rangle) & = \mathrm{Eval}_{\mathcal{P},\Phi}(\langle \mathcal{C}, \langle\langle i', \eta\rangle, \hat{\eta}, Q\rangle\rangle, p^{\mathsf{t}}) \\
\mathrm{Need}_{\mathcal{P},\Phi}(\check{t}, \langle i, p^{\mathsf{t}}\rangle) & = \mathrm{Need}_{\mathcal{P},\Phi}(\check{t}, p^{\mathsf{t}})
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{PreVC}_{\mathcal{P},\Phi}(\check{t}, \langle p^{\mathsf{t}}, p^{\mathsf{c}}_1, \ldots, p^{\mathsf{c}}_k\rangle) & = \langle \mathrm{PreVC}_{\mathcal{P},\Phi}(\check{t}, p^{\mathsf{t}}), \mathrm{PreVC}_{\mathcal{P},\Phi}(\check{t}', p^{\mathsf{c}}_1), \ldots, \mathrm{PreVC}_{\mathcal{P},\Phi}(\check{t}', p^{\mathsf{c}}_k)\rangle \\
 & \quad \text{where } \check{t}' = \mathrm{In}_{\mathcal{P},\Phi}(\check{t}, p^{\mathsf{t}}) \\
\mathrm{Eval0}_{\mathcal{P},\Phi}(\langle \mathcal{C}, \check{t}\rangle, \langle p^{\mathsf{t}}, p^{\mathsf{c}}_1, \ldots, p^{\mathsf{c}}_k\rangle) & = \langle \mathcal{C}'[i_1 \mapsto \mathrm{Need}_{\mathcal{P},\Phi}(\check{t}', p^{\mathsf{c}}_1), \ldots, i_k \mapsto \mathrm{Need}_{\mathcal{P},\Phi}(\check{t}', p^{\mathsf{c}}_k)], \mathrm{Out}_{\mathcal{P},\Phi}(\check{t}', p^{\mathsf{t}})\rangle \\
 & \quad \text{if } i_1, \ldots, i_k \notin \mathrm{dom}\,\mathcal{C}' \\
 & \quad \text{where } \langle \mathcal{C}', \check{t}'\rangle = \mathrm{Eval}_{\mathcal{P}}(\langle \mathcal{C}, \mathrm{In}_{\mathcal{P},\Phi}(\check{t}, p^{\mathsf{t}})\rangle, \mathrm{ASp}(p^{\mathsf{t}})) \\
 & \quad \text{and } \langle i_1, p^{\mathsf{t}}_1\rangle, \ldots, \langle i_k, p^{\mathsf{t}}_k\rangle = p^{\mathsf{c}}_1, \ldots, p^{\mathsf{c}}_k \\
\mathrm{Eval}_{\mathcal{P},\Phi}(\langle \mathcal{C}, \check{t}\rangle, \langle p^{\mathsf{t}}, p^{\mathsf{c}}_1, \ldots, p^{\mathsf{c}}_k\rangle) & = \langle\langle \mathcal{C}'_0, \check{t}'_0\rangle, \ldots, \langle \mathcal{C}'_k, \check{t}'_k\rangle\rangle \\
 & \quad \text{where } \langle \mathcal{C}'_j, \check{t}'_j\rangle \\
 & \qquad = \begin{cases} \mathrm{Eval0}_{\mathcal{P},\Phi}(\langle \mathcal{C}, \check{t}\rangle, \langle p^{\mathsf{t}}, p^{\mathsf{c}}_1, \ldots, p^{\mathsf{c}}_k\rangle) & \text{if } j = 0 \text{ or } p^{\mathsf{c}}_j = \langle i, p^{\mathsf{t}}\rangle \text{ for } k = 1 \\ \mathrm{Eval}_{\mathcal{P},\Phi}(\langle \mathcal{C}, \mathrm{In}_{\mathcal{P},\Phi}(\check{t}, p^{\mathsf{t}})\rangle, p^{\mathsf{c}}_j) & \text{otherwise} \end{cases} \\
 & \quad \text{and } \langle\langle i, \eta\rangle, \hat{\eta}, Q\rangle = \check{t}
\end{array}
$$

Figure 26: Specification Evaluation

$$\begin{aligned}
&\mathrm{Conj}_{\langle t,\hat{\eta},Q\rangle}(\{Q(P_1),\ldots,Q(P_k)\}) &&= Q(P_1 \wedge \cdots \wedge P_k)\\
&\mathrm{Trace}_{\Psi,\mathcal{P},\Phi}(\{\langle \mathcal{C}_1,\check{t}_1\rangle,\ldots,\langle \mathcal{C}_k,\check{t}_k\rangle\}) &&= \{\langle \mathcal{C}_1, \prec\!\ast\, \check{t}_1\, \breve{\pi}_1\rangle,\ldots,\langle \mathcal{C}_k, \prec\!\ast\, \check{t}_k\, \breve{\pi}_k\rangle\}\\
& &&\quad \text{if } \Psi;\mathcal{P};\Phi \mathrel{\widetilde{\triangleright\!\!\triangleright}} \prec\!\ast\, \check{t}_j\, \breve{\pi}_j \text{ for all } j \in \{1,\ldots,k\}
\end{aligned}$$

$$\begin{aligned}
\mathrm{ClosedVCS}_{\mathcal{P},\Phi,\check{t}}(p^{|},\quad) &= \{(\mathrm{PreVC}_{\mathcal{P},\Phi}(\check{t},p^{|}))_j \mid (\mathrm{PreVC}_{\mathcal{P},\Phi}(\check{t},p^{|}))_j \neq \mathtt{open}\}\\
\mathrm{Closed}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C},\check{t}\rangle,p^{|}) &= \mathrm{Trace}_{\Psi,\mathcal{P},\Phi}(\{(\mathrm{Eval}_{\mathcal{P},\Phi}(\langle \mathcal{C},\check{t}\rangle,p^{|}))_j \mid (\mathrm{PreVC}_{\mathcal{P},\Phi}(\check{t},p^{|}))_j \neq \mathtt{open}\})\\
\mathrm{Open}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C},\check{t}\rangle,p^{|}) &= \mathrm{Trace}_{\Psi,\mathcal{P},\Phi}(\{(\mathrm{Eval}_{\mathcal{P},\Phi}(\langle \mathcal{C},\check{t}\rangle,p^{|}))_j \mid (\mathrm{PreVC}_{\mathcal{P},\Phi}(\check{t},p^{|}))_j = \mathtt{open}\})
\end{aligned}$$

$$\begin{aligned}
\mathrm{VC}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C},\breve{\pi}\,\langle t,\hat{\eta},Q\rangle\cdot\!>\rangle) &= Q'(\top) \quad \text{if } \Phi; t\cdot\!> \tilde{\triangleright}\langle \mathcal{P},\hat{\eta},Q\rangle \xrightarrow{\mathsf{s}*} \langle \cdot,\hat{\eta}',Q'\rangle\\
\mathrm{VC}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C},\breve{\pi}\,\check{t} \ast\!\succ \langle p^{|}_1,\ldots,p^{|}_k\rangle\rangle) &= \begin{cases} Q(\theta_{\check{t}}(\mathcal{C}(i))) & \text{if } i \in \operatorname{dom}\mathcal{C}\\ \mathrm{Conj}_{\check{t}}\left(\bigcup_{j=1}^{k} \mathrm{ClosedVCS}_{\mathcal{P},\Phi,\check{t}}(p^{|}_j,\cup)\mathrm{VCS}_j\right) & \text{if } i \notin \operatorname{dom}\mathcal{C} \text{ and } k > 0 \end{cases}\\
&\quad \text{where } \mathrm{VCS}_j = \{\mathrm{VC}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C}',\breve{\pi}'\rangle) \mid \langle \mathcal{C}',\breve{\pi}'\rangle \in \mathrm{Open}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C},\check{t}\rangle,p^{|}_j)\}\\
&\quad \text{and } \langle\langle i,\eta\rangle,\hat{\eta},Q\rangle = \check{t}\\
\mathrm{Next}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C},\breve{\pi}\,\check{t}\cdot\!>\rangle) &= \emptyset\\
\mathrm{Next}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C},\breve{\pi}\,\check{t} \ast\!\succ \langle p^{|}_1,\ldots,p^{|}_k\rangle\rangle) &= \begin{cases} \emptyset & \text{if } i \in \operatorname{dom}\mathcal{C}\\ \bigcup_{j=1}^{k} \mathrm{Closed}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C},\check{t}\rangle,p^{|}_j) \cup \mathrm{Next}_j & \text{otherwise} \end{cases}\\
&\quad \text{where } \mathrm{Next}_j = \bigcup\nolimits_{\langle \mathcal{C}',\breve{\pi}'\rangle \in \mathrm{Open}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C},\check{t}\rangle,p^{|}_j)} \mathrm{Next}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C}',\breve{\pi}'\rangle)\\
&\quad \text{and } \langle\langle i,\eta\rangle,\hat{\eta},Q\rangle = \check{t}
\end{aligned}$$

$$\begin{aligned}
\mathrm{Fix}_{\Psi,\mathcal{P},\Phi}(\mathcal{S}_1,\mathcal{S}_2) &= \begin{cases} \mathrm{Fix}_{\Psi,\mathcal{P},\Phi}(\mathcal{S}_1 \cup \{\langle \mathcal{C},\breve{\pi}\rangle\}, \mathcal{S}_2 \cup \mathrm{Next}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C},\breve{\pi}\rangle)) & \text{if } \langle \mathcal{C},\breve{\pi}\rangle \in \mathcal{S}_2 \setminus \mathcal{S}_1\\ \mathcal{S}_1 & \text{if } \mathcal{S}_2 \subseteq \mathcal{S}_1 \end{cases}\\
\mathrm{VC}_{\Psi,\mathcal{P},\Phi}(\mathcal{S}) &= \bigwedge\nolimits_{\langle \mathcal{C},\breve{\pi}\rangle \in \mathrm{Fix}_{\Psi,\mathcal{P},\Phi}(\emptyset,\mathcal{S})} \mathrm{VC}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C},\breve{\pi}\rangle)
\end{aligned}$$

$$\begin{aligned}
\mathrm{VC}_{\Psi,\mathcal{P},\Phi} = {}&\mathrm{VC}_{\Psi,\mathcal{P},\Phi}(\{\langle \emptyset, <\!\cdot\langle\langle i_j,\eta_j\rangle,\hat{\eta}_j,Q_j\rangle\,\breve{\pi}_j\rangle \mid 1 \leq j \leq k\})\\
&\text{if } \Psi;\mathcal{P};\Phi \mathrel{\widetilde{\triangleright\!\!\triangleright}} <\!\cdot\langle\langle i_j,\eta_j\rangle,\hat{\eta}_j,Q_j\rangle\,\breve{\pi}_j \text{ and } \eta_j(\mathtt{ra}) = \overline{i'_j} \text{ for all } j \in \{1,\ldots,k\}\\
&\text{and } \vdash \mathcal{P}\ \mathsf{sp} \text{ where } \{\langle i_1,i'_1\rangle,\ldots,\langle i_k,i'_k\rangle\} = \{\langle i,i'\rangle \mid \mathrm{Agent}_{\Psi}(\langle i,\rho\rangle,\langle i',\rho'\rangle)\}
\end{aligned}$$

Figure 27: VC Generation

3.8 Optimizations

The symbolic evaluator of this report can be optimized like Necula's symbolic evaluator [Nec98]. These optimizations reduce the size of program VCs as well as evaluation time, but are not incorporated here to simplify presentation.

A register move instruction

$$\dot{r}_1 \leftarrow \dot{r}_2$$

is encoded in PAL by

$$\dot{r}_1 \leftarrow \dot{r}_2 \ \mathbf{orw} \ \dot{r}_2$$

where **orw** is the logical "or" operator. Our symbolic evaluator could recognize the above pattern and simply transfer the contents of $\dot{r}_2$ to $\dot{r}_2$.

An unconditional jump instruction

$$\texttt{jump}\ n$$

is encoded in PAL by

$$\texttt{cond}\ \mathbf{truew}\ \dot{r}_0, n$$

where **truew** is always true. Our symbolic evaluator could recognize the above pattern and only include the branch target in the link-specification set.

Necula's symbolic evaluator enforces a stack discipline to optimize local stack-frame accesses. Essentially, the top stack frame is treated as a register file so that load and store can be treated as register assignments. Optimizing our symbolic evaluator in this fashion is a more substantial undertaking, but we believe it to be important for machine architectures with few registers.

The `scs` rule declares machine registers that are unchanged system calls. When this rule is interpreted, we introduce an assertion that the register before the call is equal to the register after the call; the new register always contains a fresh variable. Instead of introducing the variable, we could propagate the register unchanged across matching calls.

4 Soundness Proof

This section contains a sequence of lemmas and supporting definitions that prove the soundness of the VC generator. This section has a "bottom-up" structure: simplest definitions appear first, and from these, successively more complex definitions are constructed. First, however, we overview the entire proof.

To show soundness, we must show that every program execution is a security-property execution. Formally, we must show

$$\Sigma_{\Psi,\Phi} \subseteq \mathcal{P}(\Sigma_{\Psi,\Phi})$$

assuming that

$$\models \mathrm{VC}_{\Psi,\mathcal{P},\Phi}$$

To approach this, we show that for each execution in $\Sigma_{\Psi,\Phi}$, there is an execution in $\mathcal{P}(\Sigma_{\Psi,\Phi})$. This amounts to showing that for each $\Psi; \Phi \rhd\!\!\rhd \sigma$, there is a $\Psi; \mathcal{P}; \Phi \rhd\!\!\rhd \breve{\sigma}$, such that $\breve{\sigma}$ erases to σ.

The essence of the proof is in showing that for each unextended state, there is a corresponding extended state; the extended execution is constructed from these individual states. The validity of the VC ensures that security-property checks succeed. Symbolic executions relate the VC to a concrete execution, and thus constitute a "blueprint" for constructing the extended execution.

The symbolic evaluator substitutes logical variables for the concrete values of a real execution; specifications assert abstract invariants on these variables. For the proof, we must derive a concrete execution from symbolic ones: this is accomplished by exhibiting an environment ϕ that assigns concrete values to the variables appearing in the symbolic execution; a concrete execution is the value of a symbolic execution in ϕ. We can define a valuation function for simple states (see Section 2.2.2), but extended states are more complicated, so we use a simulation relation instead (see Section 4.4).

The remainder of this section follows this format: a prose description of a definition or lemma is given, followed by its mathematical content (the prose description is only illustrative). Note that proofs contain occasional judgments from Appendix B (static semantics) and Appendix C (pattern matching). Purely technical lemmas appear in Section 4.8.

Application of an inference rule is abbreviated "App.", and inversion is abbreviated "Inv." Inverting an inference rule asserts one or more of its *premises* based on its *conclusion*. Inversion is possible when the inference rule for a judgment can be identified (*e.g.*, only one rule is applicable), or, more generally, when all possible rules have a common premise.

4.1 Substitutions

A substitution θ maps variables to expressions; if we valuate each expression in an environment ϕ, we have an environment with the domain of θ. The result environment is the *value* of θ in ϕ:

Definition 4.1 (Valuation)

1. $\theta \in \operatorname{dom} \mathcal{V}_\phi$ *iff* $\theta(x) \in \mathcal{V}_\phi$ *for all* $x \in \operatorname{dom} \theta$,
2. $\operatorname{dom}(\mathcal{V}_\phi(\theta)) = \operatorname{dom} \theta$ *for all* $\theta \in \operatorname{dom} \mathcal{V}_\phi$, *and*
3. $\mathcal{V}_\phi(\theta)(x) = \mathcal{V}_\phi(\theta(x))$ *for all* $\theta \in \operatorname{dom} \mathcal{V}_\phi$ *and* $x \in \operatorname{dom} \theta$

4.2 Steps

The value of a symbolic state is obtained from the value of its register substitution; the value is a concrete state:

Definition 4.2 (State Valuation)

1. $\langle i, \eta \rangle \in \operatorname{dom} \mathcal{V}_\phi$ *iff* $\eta \in \mathcal{V}_\phi$, *and*

2. $\mathcal{V}_\phi(\langle i, \eta \rangle) = \langle i, \mathcal{V}_\phi(\eta) \rangle$ *for all* $\langle i, \eta \rangle \in \operatorname{dom} \mathcal{V}_\phi$

ϕ provides values for variables introduced by the symbolic evaluator: a single symbolic state may evaluate to many distinct concrete states depending on ϕ.

We extend valuation to steps: the value of a symbolic step is obtained from the value(s) of its component state(s); the value is a concrete step. Head and tail steps are not valuable:

Definition 4.3 (Step Valuation)

1. $$\begin{array}{lll} {<}\!\cdot t \in \operatorname{dom} \mathcal{V}_\phi & \textit{iff} & t \in \operatorname{dom} \mathcal{V}_\phi \quad \textit{, and} \\ t \to t' \in \operatorname{dom} \mathcal{V}_\phi & \textit{iff} & t, t' \in \operatorname{dom} \mathcal{V}_\phi \\ t \cdot\!{>} \in \operatorname{dom} \mathcal{V}_\phi & \textit{iff} & t \in \operatorname{dom} \mathcal{V}_\phi \end{array}$$

2. $$\begin{array}{ll} \prec\!\ast\, t & \notin \operatorname{dom} \mathcal{V}_\phi \ \textit{, and} \\ t \ast\!\succ \langle p_1^{\mathsf{I}}, \ldots, p_k^{\mathsf{I}} \rangle & \notin \operatorname{dom} \mathcal{V}_\phi \end{array}$$

3. $$\mathcal{V}_\phi(B) = \begin{cases} {<}\!\cdot \mathcal{V}_\phi(t) & \textit{if } B = {<}\!\cdot t \\ \mathcal{V}_\phi(t) \to \mathcal{V}_\phi(t') & \textit{if } B = t \to t' \\ \mathcal{V}_\phi(t) \cdot\!{>} & \textit{if } B = t \cdot\!{>} \end{cases}$$
   for all $B \in \operatorname{dom} \mathcal{V}_\phi$

A jump within a procedure is within its program:

Lemma 4.1 (Jump Target) $i' \in \operatorname{Dom}(\Phi)$ *if* $i' \in \operatorname{Jump}_\Phi(i)$

PROOF:
Consequence of applicable definitions

$\square$

The value of a symbolic state substitution is the concrete state substitution for the value of the symbolic state:

Lemma 4.2 (State-Substitution Valuation) $\mathcal{V}_\phi(\theta_t) = \phi_{\mathcal{V}_\phi(t)}$ *if* $\theta_t \in \operatorname{dom} \mathcal{V}_\phi$

PROOF:

let $t = \langle i, \eta \rangle$	
$\mathcal{V}_\phi(\theta_{\langle i, \eta \rangle}) = \mathcal{V}_\phi(\eta[\mathtt{pc} \mapsto \bar{i}])$	Def. θ_t
$= \mathcal{V}_\phi(\eta)[\mathtt{pc} \mapsto \mathcal{V}_\phi(\bar{i})]$	Lemma 4.57
$= \mathcal{V}_\phi(\eta)[\mathtt{pc} \mapsto i]$	Def. $\mathcal{V}_\phi$
$= \phi_{\langle i, \mathcal{V}_\phi(\eta) \rangle}$	Def. ϕ_s
$= \phi_{\mathcal{V}_\phi(\langle i, \eta \rangle)}$	Def. 4.2

$\square$

Variables that are not free in a state or step do not affect its valuation:

Lemma 4.3 (Left Valuation)

1. $\mathcal{V}_{\phi_1 \cup \phi_2}(t) = \mathcal{V}_{\phi_1}(t)$ *if* $t \in \operatorname{dom} \mathcal{V}_{\phi_1}$ *and* $\operatorname{dom} \phi_1 \cap \operatorname{dom} \phi_2 = \emptyset$, *and*

2. $\mathcal{V}_{\phi_1 \cup \phi_2}(B) = \mathcal{V}_{\phi_1}(B)$ *if* $B \in \operatorname{dom} \mathcal{V}_{\phi_1}$ *and* $\operatorname{dom} \phi_1 \cap \operatorname{dom} \phi_2 = \emptyset$

PROOF:
Consequence of Lemma 4.55, Definition 4.2, and Definition 4.3

□

The value of a state or step in a concatenation is the value of the state or step in the right-hand environment, if such a value exists:

Lemma 4.4 (Right Valuation)

1. $\mathcal{V}_{\phi_1 \cup \phi_2}(t) = \mathcal{V}_{\phi_2}(t)$ *if* $t \in \operatorname{dom} \mathcal{V}_{\phi_2}$, *and*

2. $\mathcal{V}_{\phi_1 \cup \phi_2}(B) = \mathcal{V}_{\phi_2}(B)$ *if* $B \in \operatorname{dom} \mathcal{V}_{\phi_2}$

PROOF:
Consequence of Lemma 4.56, Definition 4.2, and Definition 4.3

□

Given a symbolic transition $t \xrightarrow{\mathtt{m}} t'$, a concrete transition from the value of t in some environment ϕ results in the value of t' in ϕ:

Lemma 4.5 (Transition Simulation) $\mathcal{V}_\phi(t') = s'$
if $\Phi \mathrel{\tilde{\triangleright}} t \xrightarrow{\mathtt{m}} t'$ *and* $\Phi \triangleright \mathcal{V}_\phi(t) \xrightarrow{\mathtt{m}} s'$

PROOF:

let $t = \langle i, \eta \rangle, t' = \langle i', \eta' \rangle$

$\mathcal{V}_\phi(t) = \langle i, \mathcal{V}_\phi(\eta) \rangle \qquad \mathcal{V}_\phi(t') = \langle i', \mathcal{V}_\phi(\eta') \rangle$ — Def. 4.2

Case: $\Phi_i = r_1 \leftarrow r_2\ eop\ r_3, i' = i \dot{+} 1, \eta' = \eta[r_1 \mapsto eop(\eta(r_2), \eta(r_3))]$,
$s' = \langle i \dot{+} 1, \mathcal{V}_\phi(\eta)[r_1 \mapsto \mathcal{J}(eop)(\mathcal{V}_\phi(\eta)(r_2), \mathcal{V}_\phi(\eta)(r_3))] \rangle$

$\mathcal{V}_\phi(\eta)[r_1 \mapsto \mathcal{J}(eop)(\mathcal{V}_\phi(\eta)(r_2), \mathcal{V}_\phi(\eta)(r_3))]$

$= \mathcal{V}_\phi(\eta)[r_1 \mapsto \mathcal{J}(eop)(\mathcal{V}_\phi(\eta(r_2)), \mathcal{V}_\phi(\eta(r_3)))]$ — Def. 4.1

$= \mathcal{V}_\phi(\eta)[r_1 \mapsto \mathcal{V}_\phi(eop(\eta(r_2), \eta(r_3)))]$ — Def. $\mathcal{V}_\phi(E)$

$= \mathcal{V}_\phi(\eta[r_1 \mapsto eop(\eta(r_2), \eta(r_3))]) = \mathcal{V}_\phi(\eta')$ — Lemma 4.57

Case: $\Phi_i = \mathtt{ra} \leftarrow \mathtt{pc}\ \mathbf{addw}\ n, i' = i \dot{+} 1, \eta' = \eta[\mathtt{ra} \mapsto \overline{\underline{n} \dot{+} i \dot{+} 1}]$,
$s' = \langle i \dot{+} 1, \mathcal{V}_\phi(\eta)[\mathtt{ra} \mapsto \underline{n} \dot{+} i \dot{+} 1] \rangle$

$\mathcal{V}_\phi(\eta)[\mathtt{ra} \mapsto \underline{n} \dot{+} i \dot{+} 1] = \mathcal{V}_\phi(\eta)[\mathtt{ra} \mapsto \mathcal{J}(\overline{\underline{n} \dot{+} i \dot{+} 1})]$ — Def. $\bar{\bar{i}}$

$= \mathcal{V}_\phi(\eta)[\mathtt{ra} \mapsto \mathcal{V}_\phi(\overline{\underline{n} \dot{+} i \dot{+} 1})]$ — Def. $\mathcal{V}_\phi$

$= \mathcal{V}_\phi(\eta[\mathtt{ra} \mapsto \overline{\underline{n} \dot{+} i \dot{+} 1}]) = \mathcal{V}_\phi(\eta')$ — Lemma 4.57

Other cases are similar to the first case

□

Given a symbolic step $t \to t'$, a concrete step from the value of t in some environment ϕ results in the value of t' in ϕ:

Lemma 4.6 (Next-Step Simulation) $\mathcal{V}_\phi(t') = s'$
if $\Psi; \Phi \mathrel{\tilde{\triangleright}} t \to t'$ *and* $\Psi; \Phi \triangleright \mathcal{V}_\phi(t) \to s'$

PROOF:

let $t = \langle i, \eta \rangle, t' = \langle i', \eta' \rangle$

$\Phi \tilde{\triangleright} t \xrightarrow{\mathsf{m}} t' \qquad i' \in \mathrm{Jump}_\Phi(i)$ — Inv. $\mathsf{ma}_2^{\sim}$

$\Phi \triangleright \mathcal{V}_\phi(t) \xrightarrow{\mathsf{m}} s''$ — Inv. $\mathsf{ma}_2, \mathsf{ma}_3$

$\mathcal{V}_\phi(t') = s''$ — Lemma 4.5

$s'' = \langle i', \mathcal{V}_\phi(\eta') \rangle$ — Def. 4.2

$s' = s''$ — Insp. of ma rules

□

4.3 Pattern Matching

This section is based on the pattern-matching semantics of Appendix C. The domain of a pattern-variable substitution is the domain of the context in which the pattern type checks; the free variables of the substitution are among the machine registers:

Lemma 4.7 (Pattern Variables)

1. $\mathrm{dom}\,\theta = \mathrm{dom}\,\Gamma$ *if* $\Gamma \vdash A^\square$ pat *and* $\Phi \triangleright A^\square; A \xrightarrow{\mathsf{pa}} \theta; \phi$, *and*

2. $\mathrm{FV}(\theta) \subseteq \mathit{UReg}$ *if* $\Phi \triangleright A^\square; A \xrightarrow{\mathsf{pa}} \theta; \phi$

PROOF:
By inspection of the pattern well-formedness and match rules

□

The environment produced by a pattern-matching rule is defined only on the machine registers:

Lemma 4.8 (Environment Domain)

1. $\mathrm{dom}\,\phi = \mathit{XReg}$ *if* $\Phi \triangleright s^\square; s \xrightarrow{\mathsf{ps}} \theta; \phi$, *and*

2. $\mathrm{dom}\,\phi = \mathit{XReg}$ *if* $\Phi \triangleright A^\square; A \xrightarrow{\mathsf{pa}} \theta; \phi$

PROOF:

Item 1:

by inspection of the state match rules, $\phi = \phi_{\langle i, \rho \rangle}$ for some i, ρ

$\mathrm{dom}(\phi_{\langle i, \rho \rangle}) = \mathrm{dom}(\rho[\mathsf{pc} \mapsto i])$ — Def. ϕ_s

$= \mathrm{dom}\,\rho \cup \{\mathsf{pc}\} = \mathit{XReg}$ — Def. $\phi[\mapsto]$

Item 2:

by inspection of the step match rules and item 1

□

If a symbolic state t matches a pattern $s^\square$ with respect to substitutions θ_1 and θ_2, then the value of t matches $s^\square$ with respect to θ_1 and the value of θ_2:

Lemma 4.9 (State Simulation) $\Phi \triangleright s^\square; \mathcal{V}_\phi(t) \xrightarrow{\mathsf{ps}} \theta_1; \mathcal{V}_\phi(\theta_2)$ *if* $\Phi \tilde{\triangleright} s^\square; t \xrightarrow{\mathsf{ps}} \theta_1; \theta_2$ *and* $t \in \mathrm{dom}\,\mathcal{V}_\phi$

Proof:

Case: $s^{\square} = I^{\square}, t = \langle i, \eta \rangle, \theta_2 = \theta_{\langle i,\eta\rangle}$

$\triangleright I^{\square}; \Phi_i \stackrel{\mathsf{pi}}{\to} \theta_1$	Inv. $\mathsf{ps}_2^{\sim}$
$\Phi \triangleright I^{\square}; \langle i, \mathcal{V}_\phi(\eta)\rangle \stackrel{\mathsf{ps}}{\to} \theta_1; \phi_{\langle i, \mathcal{V}_\phi(\eta)\rangle}$	App. ps_2
$\Phi \triangleright I^{\square}; \mathcal{V}_\phi(\langle i, \eta\rangle) \stackrel{\mathsf{ps}}{\to} \theta_1; \phi_{\mathcal{V}_\phi(\langle i,\eta\rangle)}$	Def. 4.2
$\Phi \triangleright I^{\square}; \mathcal{V}_\phi(\langle i, \eta\rangle) \stackrel{\mathsf{ps}}{\to} \theta_1; \mathcal{V}_\phi(\theta_{\langle i,\eta\rangle})$	Lemma 4.2

Other cases are similar

□

Lemma 4.10 is Lemma 4.9 for steps:

Lemma 4.10 (Step Simulation) $\Phi \triangleright A^{\square}; \mathcal{V}_\phi(B) \stackrel{\mathsf{pa}}{\to} \theta_1; \mathcal{V}_\phi(\theta_2)$ *if* $\Phi \,\tilde{\triangleright}\, A^{\square}; B \stackrel{\mathsf{pa}}{\to} \theta_1; \theta_2$ *and* $B \in \operatorname{dom} \mathcal{V}_\phi$

Proof:

Case: $A^{\square} = <\!\cdot s^{\square}, B = <\!\cdot t$

$\Phi \,\tilde{\triangleright}\, s^{\square}; t \stackrel{\mathsf{ps}}{\to} \theta_1; \theta_2$	Inv. $\mathsf{pa}_1^{\sim}$
$t \in \operatorname{dom} \mathcal{V}_\phi$	Def. 4.3
$\Phi \triangleright s^{\square}; \mathcal{V}_\phi(t) \stackrel{\mathsf{ps}}{\to} \theta_1; \mathcal{V}_\phi(\theta_2)$	Lemma 4.9
$\Phi \triangleright <\!\cdot s^{\square}; <\!\cdot \mathcal{V}_\phi(t) \stackrel{\mathsf{pa}}{\to} \theta_1; \mathcal{V}_\phi(\theta_2)$	App. pa_1
$\Phi \triangleright <\!\cdot s^{\square}; \mathcal{V}_\phi(<\!\cdot t) \stackrel{\mathsf{pa}}{\to} \theta_1; \mathcal{V}_\phi(\theta_2)$	Def. 4.3

Case: $A^{\square} = s^{\square}\!\cdot\!>, B = t\!\cdot\!>$
  similar to previous case

Case: $A^{\square} = \to s^{\square}, B = t' \to t, s^{\square} \neq \texttt{proc} \ \ldots$

$\Phi \,\tilde{\triangleright}\, s^{\square}; t \stackrel{\mathsf{ps}}{\to} \theta_1; \theta_2$	Inv. $\mathsf{pa}_1^{\sim}$
$t', t \in \operatorname{dom} \mathcal{V}_\phi$	Def. 4.3
$\Phi \triangleright s^{\square}; \mathcal{V}_\phi(t) \stackrel{\mathsf{ps}}{\to} \theta_1; \mathcal{V}_\phi(\theta_2)$	Lemma 4.9
$\Phi \triangleright \to s^{\square}; \mathcal{V}_\phi(t') \to \mathcal{V}_\phi(t) \stackrel{\mathsf{pa}}{\to} \theta_1; \mathcal{V}_\phi(\theta_2)$	App. pa_1
$\Phi \triangleright \to s^{\square}; \mathcal{V}_\phi(B) \stackrel{\mathsf{pa}}{\to} \theta_1; \mathcal{V}_\phi(\theta_2)$	Def. 4.3

Cases $A^{\square} = \to s^{\square}$ or $A^{\square} = s^{\square} \to$ are similar

Cases $A^{\square} = \stackrel{<}{\to} s^{\square}$ and $A^{\square} = s^{\square} \stackrel{>}{\to}$ follow by induction

□

4.4 Security Properties

A security-property state is a *prefix* if its proposition context is:

Definition 4.4 (Prefix State) $\langle \mathcal{P}, \hat{\eta}, Q \rangle \preceq P$ *iff* $P = Q(P')$ *for some* P'

Because the VC accumulates in the proposition context, this definition tells us whether a given state leads to a given VC.

We would like to valuate security-property states, but a symbolic state does not have determine a single concrete state. For example, the q component can equally well be zero or nonzero. Instead, we assert that a symbolic state *simulates* a concrete state. Simulation is with respect to an environment ϕ and a proposition P: ϕ assigns values to variables, and P is the VC of the current symbolic execution.

For state $\langle \mathcal{P}, \hat{\eta}, Q\rangle$ to simulate state $\langle \mathcal{P}, \hat{\rho}, q\rangle$, $\hat{\rho}$ must be the value of $\hat{\eta}$ in some environment ϕ, and the remainder of P (after extracting Q) must be true in ϕ. The last requirement is waived if q is zero (this indicates that an assumption has failed). Other parts of Definition 4.5 are technical invariants:

Definition 4.5 (State Simulation) $\langle \mathcal{P}, \hat{\eta}, Q\rangle \sim \langle \mathcal{P}, \mathcal{V}_\phi(\hat{\eta}), q\rangle \ (P, \phi)$
iff

1. $\Gamma \vdash_\Delta \mathcal{P}$ sp *for some* Δ, Γ *such that* $\operatorname{dom}\Gamma \subseteq \mathit{XReg} \cup \mathit{PReg}$,
2. $P = Q(P')$ *for some* P' *such that* $q \neq 0$ *implies* $\vDash_\phi P'$, *and*
3. $\operatorname{dom}\phi \subseteq \mathrm{BV}(Q)$ *and* $\mathrm{BV}(Q) \cap (\mathit{XReg} \cup \mathit{PReg}) = \emptyset$

The concatenation of derivable large steps is derivable:

Lemma 4.11 (Concatenation) $\Phi; A \triangleright \hat{s}_0 \xrightarrow{\mathsf{s}*} \hat{s}'$
if $\Phi; A \triangleright \hat{s}_0 \xrightarrow{\mathsf{s}*} \hat{s}$ *and* $\Phi; A \triangleright \hat{s} \xrightarrow{\mathsf{s}*} \hat{s}'$

Proof:
By induction on the derivation of $\Phi; A \triangleright \hat{s} \xrightarrow{\mathsf{s}*} \hat{s}'$

□

After q goes to zero, we cannot get stuck:

Lemma 4.12 (Liberation) $\Phi; A \triangleright \langle \mathcal{P}, \hat{\rho}, 0\rangle \xrightarrow{\mathsf{s}*} \langle \cdot, \hat{\rho}', 0\rangle$ *for some* $\hat{\rho}'$

Proof:
By induction on the structure of $\mathcal{P}$

□

Given a transition $\hat{t} \xrightarrow{\mathsf{s}} \hat{t}'$, $\hat{t}'$ contains the proposition context of $\hat{t}$ as a prefix. So, $\hat{t}$ is a prefix of a proposition if $\hat{t}'$ is:

Lemma 4.13 (Continuity)

1. $Q' = Q(Q_1)$ *for some* Q_1 *if* $\Phi; B \,\tilde{\triangleright} \langle \mathcal{P}, \hat{\eta}, Q\rangle \xrightarrow{\mathsf{s}} \langle \mathcal{P}', \hat{\eta}', Q'\rangle$
2. $\hat{t} \preceq P$ *if* $\Phi; B \,\tilde{\triangleright}\, \hat{t} \xrightarrow{\mathsf{s}} \hat{t}'$ *and* $\hat{t}' \preceq P$
3. $\hat{t}_0 \preceq P$ *if* $\Phi; B \,\tilde{\triangleright}\, \hat{t}_0 \xrightarrow{\mathsf{s}*} \hat{t}'$ *and* $\hat{t}' \preceq P$

Proof:
Item 1 by inspection of the security-property transition rules and Lemma 4.61

Item 2:

let $\hat{t} = \langle \mathcal{P}, \hat{\eta}, Q\rangle, \hat{t}' = \langle \mathcal{P}', \hat{\eta}', Q'\rangle$	
$Q' = Q(Q_1)$	Item 1
$P = Q'(P')$	Def. 4.4
$Q(Q_1(P')) = (Q(Q_1))(P') = Q'(P') = P$	Lemma 4.62

$\hat{t} \preceq P$ Def. 4.4

Item 3 by induction on the derivation of $\Phi; B \,\tilde{\triangleright}\, \hat{t}_0 \xrightarrow{\mathsf{s*}} \hat{t}'$

□

There is a concrete transition $\hat{s} \xrightarrow{\mathsf{s}} \hat{s}'$ for each symbolic transition $\hat{t} \xrightarrow{\mathsf{s}} \hat{t}'$, assuming that $\hat{t}$ simulates $\hat{s}$ and $\hat{t}'$ is a prefix of the simulation proposition. The transition preserves simulation, but in an expanded environment:

Lemma 4.14 (Transition Simulation)

1. *$\Phi; B \,\tilde{\triangleright}\, \hat{t} \xrightarrow{\mathsf{s}} \hat{t}'$ for $B \in \operatorname{dom} \mathcal{V}_\phi$ and $\hat{t}' \preceq P_0$, and*
2. *$\hat{t} \sim \hat{s}$ (P_0, ϕ)*

implies

1. *$\Phi; \mathcal{V}_{\phi'}(B) \triangleright \hat{s} \xrightarrow{\mathsf{s}} \hat{s}'$ for some $\hat{s}', \phi'$ such that $\phi \subseteq \phi'$, and*
2. *$\hat{t}' \sim \hat{s}'$ (P_0, ϕ')*

PROOF:

let $\hat{t} = \langle \mathcal{P}, \hat{\eta}, Q \rangle, \hat{t}' = \langle \mathcal{P}', \hat{\eta}', Q' \rangle$

$\hat{s} = \langle \mathcal{P}, \hat{\rho}, q \rangle \qquad \mathcal{V}_\phi(\hat{\eta}) = \hat{\rho}$ Def. 4.5

$\Gamma \vdash_\Delta \mathcal{P}$ sp $\qquad \operatorname{dom} \Gamma \subseteq \mathit{XReg} \cup \mathit{PReg}$ Def. 4.5

$\operatorname{dom} \phi \subseteq \mathrm{BV}(Q) \qquad \mathrm{BV}(Q) \cap (\mathit{XReg} \cup \mathit{PReg}) = \emptyset$ Def. 4.5

Case: $\mathcal{P} = \mathtt{reg}\ \hat{r} : \tau; \mathcal{P}', \hat{\eta}' = \hat{\eta}, Q' = Q$

let $\phi' = \phi, \hat{s}' = \langle \mathcal{P}', \hat{\rho}, q \rangle$

$\Phi; \mathcal{V}_\phi(B) \triangleright \hat{s} \xrightarrow{\mathsf{s}} \hat{s}'$ App. s_1

$\Gamma, \hat{r} : \tau \vdash_\Delta \mathcal{P}'$ sp Inv.

$P_0 = Q(P) \qquad \vDash_\phi P$ if $q \neq 0$ Def. 4.5

$P_0 = Q'(P')$ Def. 4.4

$P' = P$ Lemma 4.63

$\hat{t}' \sim \hat{s}'$ (P_0, ϕ') Def. 4.5

Cases $\Phi \,\tilde{\triangleright}\, A^{\square}; B \overset{\mathsf{pa}}{\not\to}$ are similar to previous case

Case: $\mathcal{P} = \mathtt{require}\ A^{\square} \Rightarrow P_1; \mathcal{P}', \hat{\eta}' = \hat{\eta}, Q' = Q((\theta_2 \cup \hat{\eta})(\theta_1(P_1)) \wedge \bullet)$

let $\phi' = \phi, \hat{s}' = \langle \mathcal{P}', \hat{\rho}, q \rangle$

$\Phi \,\tilde{\triangleright}\, A^{\square}; B \xrightarrow{\mathsf{pa}} \theta_1; \theta_2$ Inv. $\mathsf{s}_3^{\sim}$

$\Phi \triangleright A^{\square}; \mathcal{V}_\phi(B) \xrightarrow{\mathsf{pa}} \theta_1; \mathcal{V}_\phi(\theta_2)$ Lemma 4.10

$\operatorname{dom}(\mathcal{V}_\phi(\theta_2)) = \mathit{XReg}$ Lemma 4.8

$\Gamma_2 \vdash A^{\square}$ pat $\qquad \Gamma, \Gamma_2 \vdash_\Delta P_1$ prop Inv.

$\mathrm{FV}(\theta_1(P_1)) \subseteq (\mathrm{FV}(P_1) \setminus \operatorname{dom} \theta_1) \cup \mathrm{FV}(\theta_1)$ Lemma 4.53

$\subseteq (\mathrm{FV}(P_1) \setminus \operatorname{dom} \Gamma_2) \cup \mathit{UReg}$ Lemma 4.7

$\subseteq ((\operatorname{dom} \Gamma \cup \operatorname{dom} \Gamma_2) \setminus \operatorname{dom} \Gamma_2) \cup \mathit{UReg}$ Lemma 4.66

$\subseteq (\operatorname{dom} \Gamma \setminus \operatorname{dom} \Gamma_2) \cup \mathit{UReg}$

$\subseteq ((\mathit{XReg} \cup \mathit{PReg}) \setminus \operatorname{dom} \Gamma_2) \cup \mathit{UReg} \subseteq \mathit{XReg} \cup \mathit{PReg}$

$\Gamma \vdash_\Delta \mathcal{P}'$ sp Inv.

$P_0 = Q(P) \qquad \vDash_\phi P$ if $q \neq 0$ Def. 4.5

$P_0 = Q'(P')$ Def. 4.4

$(Q((\theta_2 \cup \hat{\eta})(\theta_1(P_1)) \wedge \bullet))(P') = Q((\theta_2 \cup \hat{\eta})(\theta_1(P_1)) \wedge P')$ Lemma 4.62

$\models_\phi (\theta_2 \cup \hat{\eta})(\theta_1(P_1)) \wedge P'$ if $q \neq 0$ Lemma 4.63

$\models_{\phi'} P'$ if $q \neq 0$ Def. $\models_\phi$

$\hat{t}' \sim \hat{s}'\ (P_0, \phi')$ Def. 4.5

$\models_\phi (\theta_2 \cup \hat{\eta})(\theta_1(P_1))$ if $q \neq 0$ Def. $\models_\phi$

$\models_{\phi \cup \mathcal{V}_\phi(\theta_2 \cup \hat{\eta})} \theta_1(P_1)$ if $q \neq 0$ Lemma 4.60

$\models_{\mathcal{V}_\phi(\theta_2 \cup \hat{\eta})} \theta_1(P_1)$ if $q \neq 0$ Lemma 4.49

$\models_{\mathcal{V}_\phi(\theta_2) \cup \mathcal{V}_\phi(\hat{\eta})} \theta_1(P_1)$ if $q \neq 0$ Lemma 4.58

$\Phi; \mathcal{V}_\phi(B) \triangleright \hat{s} \xrightarrow{\mathsf{s}} \hat{s}'$ App. s_2

Case: $\mathcal{P} = \texttt{admit}\ A^\square \Rightarrow P_1; \mathcal{P}', \hat{\eta}' = \hat{\eta}, Q' = Q((\theta_2 \cup \hat{\eta})(\theta_1(P_1)) \supset \bullet)$

let $\phi' = \phi$

$\Phi \triangleright A^\square; \mathcal{V}_\phi(B) \xrightarrow{\mathsf{pa}} \theta_1; \mathcal{V}_\phi(\theta_2)$ See prev. case

let $q' = \begin{cases} q & \text{if } \models_{\mathcal{V}_\phi(\theta_2) \cup \mathcal{V}_\phi(\hat{\eta})} \theta_1(P_1) \\ 0 & \text{otherwise} \end{cases}$

let $\hat{s}' = \langle \mathcal{P}', \hat{\rho}, q' \rangle$

$\Phi; \mathcal{V}_\phi(B) \triangleright \hat{s} \xrightarrow{\mathsf{s}} \hat{s}'$ App. s_2

$\mathrm{dom}(\mathcal{V}_\phi(\theta_2)) = \mathit{XReg}$ Lemma 4.8

$\mathrm{FV}(\theta_1(P_1)) \subseteq \mathit{XReg} \cup \mathit{PReg}$ See prev. case

$\Gamma \vdash_\Delta \mathcal{P}'\ \mathsf{sp}$ Inv.

$\models_\phi (\theta_2 \cup \hat{\eta})(\theta_1(P_1)) \supset P'$ if $q \neq 0$ See prev. case

$\models_\phi P'$ if $\models_\phi (\theta_2 \cup \hat{\eta})(\theta_1(P_1))$ and $q \neq 0$ Def. $\models_\phi$

$\models_\phi P'$ if $\models_{\mathcal{V}_\phi(\theta_2) \cup \mathcal{V}_\phi(\hat{\eta})} \theta_1(P_1)$ and $q \neq 0$ See prev. case

$\models_{\phi'} P'$ if $q' \neq 0$

$\hat{t}' \sim \hat{s}'\ (P_0, \phi')$ Def. 4.5

Case: $\mathcal{P} = \texttt{eval}\ A^\square \Rightarrow \hat{r} := E; \mathcal{P}', \hat{\eta}' = \hat{\eta}[\hat{r} \mapsto (\theta_2 \cup \hat{\eta})(\theta_1(E))], Q' = Q$

let $\phi' = \phi$

$\Phi \triangleright A^\square; \mathcal{V}_\phi(B) \xrightarrow{\mathsf{pa}} \theta_1; \mathcal{V}_\phi(\theta_2)$ See prev. case

$\mathrm{dom}(\mathcal{V}_\phi(\theta_2)) = \mathit{XReg}$ Lemma 4.8

$\mathrm{FV}(\theta_1(E)) \subseteq \mathit{XReg} \cup \mathit{PReg}$ See prev. case

$\mathcal{V}_\phi(\hat{\eta}') = \mathcal{V}_\phi(\hat{\eta})[\hat{r} \mapsto \mathcal{V}_\phi((\theta_2 \cup \hat{\eta})(\theta_1(E)))]$ Lemma 4.57

$= \mathcal{V}_\phi(\hat{\eta})[\hat{r} \mapsto \mathcal{V}_{\mathcal{V}_\phi(\theta_2) \cup \mathcal{V}_\phi(\hat{\eta})}(\theta_1(E))]$ See prev. case

let $\hat{\rho}' = \hat{\rho}[\hat{r} \mapsto \mathcal{V}_{\mathcal{V}_\phi(\theta_2) \cup \hat{\rho}}(\theta_1(E))], \hat{s}' = \langle \mathcal{P}', \hat{\rho}', q \rangle$

$\Phi; \mathcal{V}_\phi(B) \triangleright \hat{s} \xrightarrow{\mathsf{s}} \hat{s}'$ App. s_2

$\Gamma \vdash_\Delta \mathcal{P}'\ \mathsf{sp}$ Inv.

$P_0 = Q'(P') \qquad \models_\phi P'$ if $q \neq 0$ See first case

$\hat{t}' \sim \hat{s}'\ (P_0, \phi')$ Def. 4.5

Case: $\mathcal{P} = \texttt{new}\ A^\square \Rightarrow \hat{r} : \tau; \mathcal{P}', y \notin \mathrm{BV}(Q) \cup \mathit{XReg} \cup \mathit{PReg}$,
$\hat{\eta}' = \hat{\eta}[\hat{r} \mapsto y], Q' = Q(\forall y : \tau.\bullet)$

let $v = \mathrm{New}_\Phi(\mathcal{V}_\phi(B), \hat{s}), \phi' = \phi[y \mapsto v]$

$\Phi \triangleright A^\square; \mathcal{V}_\phi(B) \xrightarrow{\mathsf{pa}} \theta_1; \mathcal{V}_\phi(\theta_2)$ See prev. case

$\mathrm{FV}(\hat{\eta}) \subseteq \mathrm{dom}\,\phi \qquad y \notin \mathrm{dom}\,\phi \qquad y \notin \mathrm{FV}(\hat{\eta})$ Lemma 4.54

$\mathrm{FV}(\hat{\eta}') \subseteq \mathrm{FV}(\hat{\eta}) \cup \{y\} \subseteq \mathrm{dom}\,\phi \cup \{y\} \subseteq \mathrm{dom}\,\phi'$ Lemma 4.51

$\hat{\eta}' \in \mathrm{dom}\,\mathcal{V}_{\phi'}$ Lemma 4.54

$\mathcal{V}_{\phi'}(\hat{\eta}') = \mathcal{V}_{\phi'}(\hat{\eta})[\hat{r} \mapsto \mathcal{V}_{\phi'}(y)]$	Lemma 4.57
$= \mathcal{V}_{\phi'}(\hat{\eta})[\hat{r} \mapsto v]$	Def. $\mathcal{V}_\phi$
$= \mathcal{V}_{\phi}(\hat{\eta})[\hat{r} \mapsto v]$	Lemma 4.45, 4.55
let $\hat{\rho}' = \hat{\rho}[\hat{r} \mapsto v], \hat{s}' = \langle \mathcal{P}', \hat{\rho}', q \rangle$	
$\phi \subseteq \phi'$	Def. $\phi_1 \subseteq \phi_2$
$\mathcal{V}_{\phi'}(B) = \mathcal{V}_\phi(B)$	Lemma 4.4
$\Phi; \mathcal{V}_{\phi'}(B) \triangleright \hat{s} \xrightarrow{\mathsf{s}} \hat{s}'$	App. s_2
$\Gamma \vdash_\Delta \mathcal{P}'\ \mathsf{sp}$	Inv.
$P_0 = Q(P) \qquad \vDash_\phi P$ if $q \neq 0$	Def. 4.5
$P_0 = Q'(P')$	Def. 4.4
$(Q(\forall y : \tau.\bullet))(P') = Q((\forall y : \tau.\bullet)(P'))$	Lemma 4.62
$= Q(\forall y : \tau.P')$	Def. $Q(P)$
$\vDash_\phi \forall y : \tau.P'$ if $q \neq 0$	Lemma 4.63
$(\vDash_{\phi[y \mapsto v_1]} P'$ for all $v_1 \in \mathcal{U}_\tau)$ if $q \neq 0$	Def. $\vDash_\phi$
$v \in \mathcal{U}_\tau$	Def. New
$\vDash_{\phi'} P'$ if $q \neq 0$	
$\operatorname{dom} \phi' \subseteq \mathrm{BV}(Q) \cup \{y\} \subseteq \mathrm{BV}(Q')$	Def. $\mathrm{BV}(Q)$
$\hat{t}' \sim \hat{s}'\ (P_0, \phi')$	Def. 4.5

□

Lemma 4.15 is Lemma 4.14 extended to large transitions:

Lemma 4.15 (Large-Transition Simulation)

1. *$\Phi; B \,\tilde{\triangleright}\, \hat{t}_0 \xrightarrow{\mathsf{s}*} \hat{t}'$ for $B \in \operatorname{dom} \mathcal{V}_{\phi_0}$ and $\hat{t}' \preceq P$, and*
2. *$\hat{t}_0 \sim \hat{s}_0\ (P, \phi_0)$*

implies

1. *$\Phi; \mathcal{V}_{\phi'}(B) \triangleright \hat{s}_0 \xrightarrow{\mathsf{s}*} \hat{s}'$ for some $\hat{s}', \phi'$ such that $\phi_0 \subseteq \phi'$, and*
2. *$\hat{t}' \sim \hat{s}'\ (P, \phi')$*

Proof:
By induction on the derivation of $\Phi; B \,\tilde{\triangleright}\, \hat{t}_0 \xrightarrow{\mathsf{s}*} \hat{t}'$

let $\hat{t}_0 = \langle \mathcal{P}_0, \hat{\eta}_0, Q_0 \rangle, \hat{t}' = \langle \mathcal{P}', \hat{\eta}', Q' \rangle$	
Case: $\mathcal{P}' = \mathcal{P}_0, \hat{t}' = \hat{t}_0$	
let $\hat{s}' = \hat{s}_0, \phi' = \phi_0$	
$\Phi; \mathcal{V}_{\phi'}(B) \triangleright \hat{s}_0 \xrightarrow{\mathsf{s}*} \hat{s}'$	App. $\mathsf{s}*_1$
Case: $\mathcal{P}' \neq \mathcal{P}_0$	
$\Phi; B \,\tilde{\triangleright}\, \hat{t}_0 \xrightarrow{\mathsf{s}*} \hat{t} \qquad \Phi; B \,\tilde{\triangleright}\, \hat{t} \xrightarrow{\mathsf{s}} \hat{t}'$	Inv. $\mathsf{s}*_2^\sim$
$\hat{t} \preceq P$	Lemma 4.13
$\Phi; \mathcal{V}_\phi(B) \triangleright \hat{s}_0 \xrightarrow{\mathsf{s}*} \hat{s} \qquad \phi_0 \subseteq \phi \qquad \hat{t} \sim \hat{s}\ (P, \phi)$	I.H.
$\Phi; \mathcal{V}_{\phi'}(B) \triangleright \hat{s} \xrightarrow{\mathsf{s}} \hat{s}' \qquad \phi \subseteq \phi' \qquad \hat{t}' \sim \hat{s}'\ (P, \phi')$	Lemma 4.14
$\mathcal{V}_{\phi'}(B) = \mathcal{V}_\phi(B)$	Lemma 4.4
$\Phi; \mathcal{V}_{\phi'}(B) \triangleright \hat{s}_0 \xrightarrow{\mathsf{s}*} \hat{s}'$	App. $\mathsf{s}*_2$
$\phi_0 \subseteq \phi'$	Def. $\phi_1 \subseteq \phi_2$

□

4.5 Extended Steps

Definition 4.6 is similar to Definition 4.4:

Definition 4.6 (Prefix State) *$\langle t, \hat{\eta}, Q\rangle \preceq P$ iff $P = Q(P')$ for some P'*

Definition 4.7 is similar to Definition 4.5, except valuation must hold for component states.

Definition 4.7 (State Simulation) *$\langle t, \hat{\eta}, Q\rangle \sim \langle \mathcal{V}_\phi(t), \mathcal{V}_\phi(\hat{\eta}), q\rangle\ (P, \phi)$ iff*

1. *$P = Q(P')$ for some P' such that $q \neq 0$ implies $\vDash_\phi P'$, and*
2. *$\operatorname{dom}\phi \subseteq \mathrm{BV}(Q)$ such that $\mathrm{BV}(Q) \cap (\mathit{XReg} \cup \mathit{PReg}) = \emptyset$*

Lemma 4.16 is Lemma 4.2 for extended states:

Lemma 4.16 (State-Substitution Valuation) *$\mathcal{V}_\phi(\theta_{\check{t}}) = \phi_{\check{s}}$ if $\check{t} \sim \check{s}\ (P, \phi)$*

PROOF:

let $\check{t} = \langle t, \hat{\eta}, Q\rangle$	
$\check{s} = \langle \mathcal{V}_\phi(t), \mathcal{V}_\phi(\hat{\eta}), q\rangle$	Def. 4.7
$\mathcal{V}_\phi(\theta_{\check{t}}) = \mathcal{V}_\phi(\theta_t \cup \hat{\eta})$	Def. $\theta_{\check{t}}$
$= \mathcal{V}_\phi(\theta_t) \cup \mathcal{V}_\phi(\hat{\eta})$	Lemma 4.58
$= \phi_{\mathcal{V}_\phi(t)} \cup \mathcal{V}_\phi(\hat{\eta})$	Lemma 4.2
$= \phi_{\check{s}}$	Def. $\phi_{\check{s}}$

□

A security-property state and an extended state with the same proposition context are prefixes of the same propositions:

Lemma 4.17 (Prefix State) *$\langle t, \hat{\eta}, Q\rangle \preceq P$ iff $\langle \mathcal{P}, \hat{\eta}, Q\rangle \preceq P$*

PROOF:
Consequence of Definition 4.4 and Definition 4.6

□

Simulation of security-property states implies simulation of extended states, and vice-versa:

Lemma 4.18 (State Simulation)

1. *$\langle t, \hat{\eta}, Q\rangle \sim \langle \mathcal{V}_\phi(t), \hat{\rho}, q\rangle\ (P, \phi)$ if $\langle \mathcal{P}, \hat{\eta}, Q\rangle \sim \langle \mathcal{P}, \hat{\rho}, q\rangle\ (P, \phi)$ and $t \in \operatorname{dom}\mathcal{V}_\phi$*
2. *$\langle \mathcal{P}, \hat{\eta}, Q\rangle \sim \langle \mathcal{P}, \hat{\rho}, q\rangle\ (P, \phi)$ if $\langle t, \hat{\eta}, Q\rangle \sim \langle s, \hat{\rho}, q\rangle\ (P, \phi)$ and $\vdash \mathcal{P}\ \mathsf{sp}$*

PROOF:
Consequence of Definition 4.5 and Definition 4.7

□

Given an extended step $\check{t} \to \check{t}'$, $\check{t}$ is a prefix of some proposition if $\check{t}'$ is:

Lemma 4.19 (Continuity) *$\check{t} \preceq P$ if $\Psi; \mathcal{P}; \Phi \tilde{\triangleright} \check{t} \to \check{t}'$ and $\check{t}' \preceq P$*

PROOF:
Consequence of Lemma 4.13 and Lemma 4.17

□

Given a symbolic step $\check{t} \to \check{t}'$ and a concrete step $s \to s'$, there is a concrete step $\check{s} \to \check{s}'$, provided that $\check{t}$ simulates $\check{s}$, $\check{s}$ erases to s, and $\check{t}'$ is a prefix of the simulation proposition. Furthermore, $\check{t}'$ will simulate $\check{s}'$ in an expanded environment, and $\check{s}'$ will erase to s':

Lemma 4.20 (Transition Simulation)

1. *$\Psi; \mathcal{P}; \Phi \tilde{\triangleright} \check{t} \to \check{t}'$ for $\check{t}' \preceq P$ and $\vdash \mathcal{P}$ sp,*
2. *$\check{t} \sim \check{s}$ (P, ϕ), and*
3. *$\Psi; \Phi \triangleright \lfloor \check{s} \rfloor \to s'$*

implies

1. *$\Psi; \mathcal{P}; \Phi \triangleright \check{s} \to \check{s}'$ for some $\check{s}'$ such that $\lfloor \check{s}' \rfloor = s'$, and*
2. *$\check{t}' \sim \check{s}'$ (P, ϕ') for some ϕ' such that $\phi \subseteq \phi'$*

PROOF:

let $\check{t} = \langle t, \hat{\eta}, Q \rangle, \check{t}' = \langle t', \hat{\eta}', Q' \rangle, \check{s} = \langle s, \hat{\rho}, q \rangle$	
$\Psi; \Phi \tilde{\triangleright} t \to t' \qquad \Phi; t \to t' \tilde{\triangleright} \langle \mathcal{P}, \hat{\eta}, Q \rangle \xrightarrow{\mathsf{s*}} \langle \cdot, \hat{\eta}', Q' \rangle$	Inv. $\mathsf{ea}_2^{\sim}$
$\mathcal{V}_\phi(t) = s \qquad \mathcal{V}_\phi(\hat{\eta}) = \hat{\rho}$	Def. 4.7
$\Psi; \Phi \triangleright \mathcal{V}_\phi(t) \to s'$	Def. $\lfloor \check{s} \rfloor$
$s' = \mathcal{V}_\phi(t')$	Lemma 4.6
$\langle \cdot, \hat{\eta}', Q' \rangle \preceq P$	Lemma 4.17
$\langle \mathcal{P}, \hat{\eta}, Q \rangle \sim \langle \mathcal{P}, \mathcal{V}_\phi(\hat{\eta}), q \rangle$ (P, ϕ)	Lemma 4.18
$\mathcal{V}_\phi(t) \to \mathcal{V}_\phi(t') = \mathcal{V}_\phi(t \to t')$	Def. 4.3
$\Phi; \mathcal{V}_{\phi'}(t \to t') \triangleright \langle \mathcal{P}, \mathcal{V}_\phi(\hat{\eta}), q \rangle \xrightarrow{\mathsf{s*}} \hat{s}' \qquad \phi \subseteq \phi'$	Lemma 4.15
$\langle \cdot, \hat{\eta}', Q' \rangle \sim \hat{s}'$ (P, ϕ')	Lemma 4.15
$\hat{s}' = \langle \cdot, \hat{\rho}', q' \rangle \qquad \mathcal{V}_{\phi'}(\hat{\eta}') = \hat{\rho}'$	Def. 4.5
let $\check{s}' = \langle s', \hat{\rho}', q' \rangle$	
$\mathcal{V}_{\phi'}(t \to t') = \mathcal{V}_\phi(t \to t') \qquad \mathcal{V}_{\phi'}(t') = \mathcal{V}_\phi(t')$	Lemma 4.4
$\Psi; \mathcal{P}; \Phi \triangleright \check{s} \to \check{s}'$	App. ea_2
$\check{t}' \sim \check{s}'$ (P, ϕ')	Lemma 4.18
$\lfloor \check{s}' \rfloor = \mathcal{V}_{\phi'}(t')$	Def. $\lfloor \check{s} \rfloor$

□

4.6 Extended Executions

$\sigma_1 \leq \sigma$ holds when σ_1 is an initial fragment σ:

Definition 4.8 (Prefix Execution) *$\sigma_1 \leq \sigma$ iff $\sigma_1 \, \sigma_2 = \sigma$ for some σ_2*

$\check{s}$ is an *intermediate state* of σ if there is some $\check{\sigma}$ that erases to a prefix of σ; furthermore, $\check{s}$ must be the last state of $\check{\sigma}$ if $\check{\sigma}$ is shorter than σ:

Definition 4.9 (Intermediate State) $\mathrm{Int}_{\Psi,\mathcal{P},\Phi}(\sigma, \check{s})$
iff $\Psi; \mathcal{P}; \Phi \rhd\!\rhd \check{\sigma}$ *for some* $\check{\sigma}$ *such that* $\lfloor \check{\sigma} \rfloor \leq \sigma$
and $\lfloor \check{\sigma} \rfloor \neq \sigma$ *implies* $\check{\sigma} = \check{\sigma}_0\, \check{s}$ *for some* $\check{\sigma}_0$

The *head state* of a symbolic execution is the first state appearing in the execution; similarly, the *tail* state is the last state:

Definition 4.10 (Execution Head/Tail)

1. $\mathrm{Head}(\breve{\pi}) = \check{t} \quad$ *if* $\breve{\pi} = <\!\cdot \check{t}\, \breve{\pi}'$ *or* $\breve{\pi} = \prec\!* \check{t}\, \breve{\pi}'$
2. $\mathrm{Tail}(\breve{\pi}) = \check{t} \quad$ *if* $\breve{\pi} = \breve{\pi}'\, \check{t}$ *or* $\breve{\pi} = \breve{\pi}'\, \check{t}\!\cdot\!> \quad$ *or* $\breve{\pi} = \breve{\pi}'\, \check{t} *\!\succ \langle p^{|}_1, \ldots, p^{|}_k \rangle$

The erasure of a concatenation is the concatenation of the erasures:

Lemma 4.21 (Concatenation Erasure) $\lfloor \check{\sigma}_1\, \check{\sigma}_2 \rfloor = \lfloor \check{\sigma}_1 \rfloor\, \lfloor \check{\sigma}_2 \rfloor$

PROOF:
By induction on the length of $\check{\sigma}_2$

□

We can always find an extended execution for an unextended execution, once q goes to zero:

Lemma 4.22 (Liberation) $\Psi; \mathcal{P}; \Phi \rhd\!\rhd \check{\sigma}'$ *for some* $\check{\sigma}'$ *such that* $\lfloor \check{\sigma}' \rfloor = \lfloor \check{\sigma} \rfloor\, s\, \sigma$
if $\Psi; \mathcal{P}; \Phi \rhd\!\rhd \check{\sigma}\, \langle s, \hat{\rho}, 0 \rangle$ *and* $\Psi; \Phi \rhd\!\rhd \lfloor \check{\sigma} \rfloor\, s\, \sigma$

PROOF:
By induction on the length of σ

□

The head of an execution is a prefix if its tail is:

Lemma 4.23 (Continuity) $\mathrm{Head}(\breve{\pi}) \preceq P$ *if* $\Psi; \mathcal{P}; \Phi \,\tilde{\rhd\!\rhd}\, \breve{\pi}$ *and* $\mathrm{Tail}(\breve{\pi}) \preceq P$

PROOF:
By induction on the derivation of $\Psi; \mathcal{P}; \Phi \,\tilde{\rhd\!\rhd}\, \breve{\pi}$

Case: $\breve{\pi} = <\!\cdot \check{t} *\!\succ \langle p^{|}_1, \ldots, p^{|}_k \rangle$
trivial
Case: $\breve{\pi} = \prec\!* \check{t}$
trivial
Case: $\breve{\pi} = \breve{\pi}_1\, \check{t} \to \check{t}'$

$\mathrm{Tail}(\breve{\pi}) = \check{t}'$	Def. 4.10
$\Psi; \mathcal{P}; \Phi \,\tilde{\rhd\!\rhd}\, \breve{\pi}_1\, \check{t} \qquad \Psi; \mathcal{P}; \Phi \,\tilde{\rhd}\, \check{t} \to \check{t}'$	Inv. $\mathsf{ee}^{\sim}_3$
$\check{t} \preceq P$	Lemma 4.19
$\mathrm{Tail}(\breve{\pi}_1\, \check{t}) \preceq P$	Def. 4.10
$\mathrm{Head}(\breve{\pi}_1\, \check{t}) \preceq P$	I.H.

$\mathrm{Head}(\check\pi) \preceq P$	Def. 4.10
Case: $\check\pi = \check\pi_1\, \check t\cdot\!>$	
$\mathrm{Tail}(\check\pi) = \check t$	Def. 4.10
$\Psi; \mathcal{P}; \Phi \mathrel{\widetilde{\rhd\!\rhd}} \check\pi_1\, \check t$	Inv. $\mathsf{ee}_3^{\sim}$
$\mathrm{Tail}(\check\pi_1\, \check t) \preceq P$	Def. 4.10
$\mathrm{Head}(\check\pi_1\, \check t) \preceq P$	I.H.
$\mathrm{Head}(\check\pi) \preceq P$	Def. 4.10
Case: $\check\pi = \check\pi_1\, \check t *\!\succ \langle p_1^{\mid}, \ldots, p_k^{\mid}\rangle, \check\pi_1 \neq <\!\cdot$	
similar to previous case	

$\square$

Given a symbolic execution $\check\pi$ and concrete executions $\check\sigma\, \check s$ and $\lfloor \check\sigma\, \check s \rfloor\, \sigma$, simulation is preserved for an intermediate state provided that the head of $\check\pi$ simulates $\check s$ and the tail of $\check\pi$ is a prefix of the simulation proposition. The tail of $\check\pi$ will simulate the intermediate state in an expanded environment:

Lemma 4.24 (Execution Simulation)

1. $\Psi; \mathcal{P}; \Phi \mathrel{\widetilde{\rhd\!\rhd}} \check\pi$ *such that* $\mathrm{Tail}(\check\pi) \preceq P$ *and* $\vdash \mathcal{P}\ \mathsf{sp}$,
2. $\mathrm{Head}(\check\pi) \sim \check s\ (P, \phi)$, *and*
3. $\Psi; \mathcal{P}; \Phi \rhd\!\rhd \check\sigma\, \check s$ *and* $\Psi; \Phi \rhd\!\rhd \lfloor \check\sigma\, \check s \rfloor\, \sigma$

implies

1. $\mathrm{Int}_{\Psi,\mathcal{P},\Phi}(\lfloor \check\sigma\, \check s \rfloor\, \sigma, \check s')$ *for some* $\check s'$, *and*
2. $\lfloor \check s' \rfloor \in \lfloor \check\sigma\, \check s \rfloor\, \sigma$ *implies* $\mathrm{Tail}(\check\pi) \sim \check s'\ (P, \phi')$ *for some* ϕ' *such that* $\phi \subseteq \phi'$

PROOF:
By induction on the derivation of $\Psi; \mathcal{P}; \Phi \mathrel{\widetilde{\rhd\!\rhd}} \check\pi$

Case: $\check\pi = <\!\cdot\, \check t *\!\succ \langle p_1^{\mid}, \ldots, p_k^{\mid}\rangle$	
$\mathrm{Head}(\check\pi) = \check t = \mathrm{Tail}(\check\pi)$	Def. 4.10
let $\check s' = \check s, \phi' = \phi$	
$\mathrm{Int}_{\Psi,\mathcal{P},\Phi}(\lfloor \check\sigma\, \check s \rfloor\, \sigma, \check s')$	Def. 4.9
Case: $\check\pi = \prec\!*\, \check t$	
similar to previous case	
Case: $\check\pi = \check\pi_1\, \check t_1 \to \check t'$	
$\mathrm{Tail}(\check\pi) = \check t'$	Def. 4.10
$\Psi; \mathcal{P}; \Phi \mathrel{\widetilde{\rhd\!\rhd}} \check\pi_1\, \check t_1 \qquad \Psi; \mathcal{P}; \Phi \mathrel{\tilde\rhd} \check t_1 \to \check t'$	Inv. $\mathsf{ee}_3^{\sim}$
$\check t_1 \preceq P$	Lemma 4.19
$\mathrm{Tail}(\check\pi_1\, \check t_1) \preceq P$	Def. 4.10
$\mathrm{Head}(\check\pi_1\, \check t_1) \sim \check s\ (P, \phi)$	Def. 4.10
$\mathrm{Int}_{\Psi,\mathcal{P},\Phi}(\lfloor \check\sigma\, \check s \rfloor\, \sigma, \check s_1)$	I.H.
$\Psi; \mathcal{P}; \Phi \rhd\!\rhd \check\sigma_1 \qquad \lfloor \check\sigma_1 \rfloor \leq \lfloor \check\sigma\, \check s \rfloor\, \sigma$	Def. 4.9
case: $\lfloor \check\sigma_1 \rfloor = \lfloor \check\sigma\, \check s \rfloor\, \sigma$	
let $\check s'$ be such that $\lfloor \check s' \rfloor \notin \lfloor \check\sigma\, \check s \rfloor\, \sigma$	
$\mathrm{Int}_{\Psi,\mathcal{P},\Phi}(\lfloor \check\sigma\, \check s \rfloor\, \sigma, \check s')$	Def. 4.9

case: $\lfloor \breve{\sigma}_1 \rfloor \neq \lfloor \breve{\sigma}\, \check{s} \rfloor\, \sigma$	
$\breve{\sigma}_1 = \breve{\sigma}_2\, \check{s}_1$	Def. 4.9
$\lfloor \breve{\sigma}\, \check{s} \rfloor\, \sigma = \lfloor \breve{\sigma}_1 \rfloor\, \sigma_1$	Def. 4.8
$\mathrm{Tail}(\breve{\pi}_1\, \check{t}_1) \sim \check{s}_1\ (P, \phi_1) \qquad \phi \subseteq \phi_1$	I.H.
$\check{t}_1 \sim \check{s}_1\ (P, \phi_1)$	Def. 4.10
case: $\sigma_1 = \cdot$	
similar to case $\lfloor \breve{\sigma}_1 \rfloor = \lfloor \breve{\sigma}\, \check{s} \rfloor\, \sigma$	
case: $\sigma_1 = \to s'\, \sigma'$	
$\Psi; \Phi \triangleright \lfloor \check{s}_1 \rfloor \to s'$	Inv. me_2
$\Psi; \mathcal{P}; \Phi \triangleright \check{s}_1 \to \check{s}' \qquad \lfloor \check{s}' \rfloor = s' \qquad \check{t}' \sim \check{s}'\ (P, \phi') \qquad \phi_1 \subseteq \phi'$	Lemma 4.20
$\Psi; \mathcal{P}; \Phi \triangleright\!\!\triangleright \breve{\sigma}_1 \to \check{s}'$	App. ee_2
$\lfloor \breve{\sigma}_1 \rfloor \to s' \leq \lfloor \breve{\sigma}\, \check{s} \rfloor\, \sigma$	Def. 4.8
$\mathrm{Int}_{\Psi, \mathcal{P}, \Phi}(\lfloor \breve{\sigma}\, \check{s} \rfloor\, \sigma, \check{s}')$	Def. 4.9
$\phi \subseteq \phi'$	Def. $\phi_1 \subseteq \phi_2$
case: $\sigma_1 = \cdot\!>$	
(hypothetically)	
$\Psi; \Phi \triangleright \lfloor \check{s}_1 \rfloor \cdot\!>$	Inv. me_3
$\Phi \triangleright \lfloor \check{s}_1 \rfloor \xrightarrow{\mathsf{m}} s'$	Inv. ma_4
let $\langle t_1, \hat{\eta}_1, Q_1 \rangle = \check{t}_1, \langle i_1, \eta_1 \rangle = t_1, \langle t', \hat{\eta}', Q' \rangle = \check{t}', \langle i', \eta' \rangle = t'$	
$\Psi; \Phi\, \tilde{\triangleright}\, t_1 \to t'$	Inv. $\mathsf{ea}_2^{\sim}$
$\Phi\, \tilde{\triangleright}\, t_1 \xrightarrow{\mathsf{m}} t' \qquad i' \in \mathrm{Jump}_{\Phi}(i_1)$	Inv. $\mathsf{ma}_2^{\sim}$
$i' \in \mathrm{Dom}(\Phi)$	Lemma 4.1
$i' \notin \mathrm{Stop}(\Psi)$	Def. $\mathrm{Stop}(\Psi)$
$\mathcal{V}_{\phi_1}(t_1) = \lfloor \check{s}_1 \rfloor$	Def. 4.7
$s' = \mathcal{V}_{\phi_1}(t')$	Lemma 4.5
$i' \in \mathrm{Stop}(\Psi)$	Inv. ma_4
contradiction	
Case: $\breve{\pi} = \breve{\pi}_1\, \check{t}_1 \cdot\!>$	
$\mathrm{Tail}(\breve{\pi}) = \check{t}_1$	Def. 4.10
$\Psi; \mathcal{P}; \Phi\, \widetilde{\triangleright\!\!\triangleright}\, \breve{\pi}_1\, \check{t}_1 \qquad \Psi; \mathcal{P}; \Phi\, \tilde{\triangleright}\, \check{t}_1 \cdot\!>$	Inv. $\mathsf{ee}_3^{\sim}$
$\mathrm{Tail}(\breve{\pi}_1\, \check{t}_1) \preceq P$	Def. 4.10
$\mathrm{Head}(\breve{\pi}_1\, \check{t}_1) \sim \check{s}\ (P, \phi)$	Def. 4.10
$\mathrm{Int}_{\Psi, \mathcal{P}, \Phi}(\lfloor \breve{\sigma}\, \check{s} \rfloor\, \sigma, \check{s}')$	I.H.
$\lfloor \check{s}' \rfloor \in \lfloor \breve{\sigma}\, \check{s} \rfloor\, \sigma$ implies $\mathrm{Tail}(\breve{\pi}_1\, \check{t}_1) \sim \check{s}'\ (P, \phi')$ and $\phi \subseteq \phi'$	I.H.
$\lfloor \check{s}' \rfloor \in \lfloor \breve{\sigma}\, \check{s} \rfloor\, \sigma$ implies $\mathrm{Tail}(\breve{\pi}) \sim \check{s}'\ (P, \phi')$ and $\phi \subseteq \phi'$	Def. 4.10
Case: $\breve{\pi} = \breve{\pi}_1\, \check{t}_1 \ast\!\succ \langle p^{!}_1, \ldots, p^{!}_k \rangle, \breve{\pi}_1 \neq <\!\cdot$	
similar to previous case	

□

4.7 VC Generator

A *junction* is required between VCs P_1 and P_2 when we transition from P_1 to P_2 during abstraction-specification evaluation. If the PreVC is not open, then it must match P_1 and P_2 must be valid; otherwise, there is no transition and P_1 must be P_2:

Definition 4.11 (VC Junction)

1. $\mathrm{Junct}(P, P_1, P_2)$ *iff* $P_1 = P$ *and* $\vDash P_2$

2. $\mathrm{Junct}(\texttt{open}, P_1, P_2)$ *iff* $P_1 = P_2$

A specification is *satisfied* with respect to state $\check{s}'$ if we can show that all its assumptions are valid in $\check{s}'$. Additionally, $\check{s}'$ must agree with a reference state $\check{s}$ for all unabstracted registers. The reference state is useful because sometimes we can only show simulation up to $\check{s}$, but must proceed from $\check{s}'$ (*e.g.*, Lemma 4.28):

Definition 4.12 (Satisfied Specification)

1. $\mathrm{Sat}_{\mathcal{P}}(\langle\langle i,\rho\rangle,\hat{\rho},q\rangle,\langle\langle i,\rho'\rangle,\hat{\rho}',q'\rangle,p^{\mathsf{a}})$
   iff $q' \leq q$ *and* $\vDash_{\phi_{\langle\langle i,\rho'\rangle,\hat{\rho}',q'\rangle}} P'$
   and $\phi_{\langle\langle i,\rho'\rangle,\hat{\rho}',q'\rangle}(x) = \phi_{\langle\langle i,\rho\rangle,\hat{\rho},q\rangle}(x)$ *for all* $x \in X'$
   where $p^{\mathsf{a}} = \texttt{all}\langle P, P'\rangle$ *and* $X' = \emptyset$
   or $p^{\mathsf{a}} = \texttt{some}\langle X, P, P'\rangle$ *and* $X' = (\mathit{Reg} \cup \mathrm{dom}\,\Gamma_{\mathcal{P}}) \smallsetminus X$

2. $\mathrm{Sat}_{\mathcal{P},\Phi}(\langle\langle i,\rho\rangle,\hat{\rho},q\rangle,\check{s}',\texttt{tor}\langle i',p^{\mathsf{a}}\rangle)$
   iff $\Phi;\langle i,\rho\rangle \to \langle i',\rho\rangle \triangleright \langle \mathcal{P},\hat{\rho},q\rangle \overset{\mathsf{s}*}{\to} \langle\cdot,\hat{\rho}',q'\rangle$ *and* $q' \neq 0$
   implies $\mathrm{Sat}_{\mathcal{P}}(\langle\langle i',\rho\rangle,\hat{\rho}',q'\rangle,\check{s}'',p^{\mathsf{a}})$ *for all* $\hat{\rho}'$, q'
   $$\textit{where } \check{s}'' = \begin{cases} \langle\langle i',\rho\rangle,\hat{\rho}',q'\rangle & \textit{if } \check{s}' = \langle\langle i,\rho\rangle,\hat{\rho},q\rangle \\ \check{s}' & \textit{otherwise} \end{cases}$$
   $\mathrm{Sat}_{\mathcal{P},\Phi}(\check{s},\check{s}',p^{\mathsf{t}})$ *iff* $\mathrm{Sat}_{\mathcal{P}}(\check{s},\check{s}',\mathrm{ASp}(p^{\mathsf{t}}))$ *if* $p^{\mathsf{t}} \neq \texttt{tor}\ldots$

3. $\mathrm{Sat}_{\mathcal{P},\Phi}(\langle\langle i,\rho\rangle,\hat{\rho},q\rangle,\check{s}',\langle i',p^{\mathsf{t}}\rangle)$ *iff* $\mathrm{Sat}_{\mathcal{P},\Phi}(\langle\langle i',\rho\rangle,\hat{\rho},q\rangle,\check{s}',p^{\mathsf{t}})$

4. $\mathrm{Sat}_{\mathcal{P},\Phi}(\check{s},\langle p^{\mathsf{t}},p^{\mathsf{c}}_1,\ldots,p^{\mathsf{c}}_k\rangle)$ *iff* $\mathrm{Sat}_{\mathcal{P},\Phi}(\check{s},\check{s},p^{\mathsf{t}})$

A specification is *available* with respect to a state if we can transition to the target address:

Definition 4.13 (Available Specification)

1. $\mathrm{Avail}_{\Psi,\Phi}(s,\texttt{id}\langle p^{\mathsf{a}}\rangle)$ *always*
   $\mathrm{Avail}_{\Psi,\Phi}(\langle i,\rho\rangle,\texttt{tol}\langle p^{\mathsf{a}},i'\rangle)$
   iff $\Psi;\Phi \rhd\!\!\rhd \sigma\,\langle i,\rho\rangle\,\sigma'$ *and* $\sigma' \neq \cdot$ *implies* $\to \langle i',\rho\rangle \leq \sigma'$ *for all* σ,σ'
   $\mathrm{Avail}_{\Psi,\Phi}(s,\texttt{tos}\langle i'\rangle)$
   iff $\Psi;\Phi \rhd\!\!\rhd \sigma\,s\,\sigma'$ *and* $\sigma' \neq \cdot$ *implies* $\to \langle i',\rho'\rangle \leq \sigma'$ *for some* ρ' *for all* σ,σ'
   $\mathrm{Avail}_{\Psi,\Phi}(\langle i,\rho\rangle,\texttt{tor}\langle i',p^{\mathsf{a}}\rangle)$
   iff $\Psi;\Phi \rhd\!\!\rhd \sigma\,\langle i,\rho\rangle\,\sigma'$ *and* $\sigma' \neq \cdot$ *implies* $\to \langle i',\rho\rangle \leq \sigma'$ *for all* σ,σ'

2. $\mathrm{Avail}_{\Psi,\Phi}(s,\langle i,p^{\mathsf{t}}\rangle)$ *iff* $\mathrm{Avail}_{\Psi,\Phi}(s,p^{\mathsf{t}})$

3. $\mathrm{Avail}_{\Psi,\Phi}(s,\langle p^{\mathsf{t}},p^{\mathsf{c}}_1,\ldots,p^{\mathsf{c}}_k\rangle)$ *iff* $\mathrm{Avail}_{\Psi,\Phi}(s,p^{\mathsf{t}})$

A trace context $\mathcal{C}$ is *enabled* with respect to a trace schedule $\mathcal{S}$ if for every state $\check{s}$ that validates a proposition of $\mathcal{C}$, we can find an intermediate state that is simulated by the head state of some $\check{\pi}'$ in $\mathcal{S}$; furthermore, the trace context for $\check{\pi}'$ must also be enabled with respect to $\mathcal{S}$:

Definition 4.14 (Enabled Context) $\mathrm{Enab}_{\Psi,\mathcal{P},\Phi}(\mathcal{C},\mathcal{S},\phi)$ *iff* $\mathrm{CFV}(\mathcal{C}) \subseteq \mathrm{dom}\,\phi$ *and for all* $\check{s}, \check{\sigma}, \sigma$ *such that*

1. $\vDash_{\phi \cup \phi_{\check{s}}} \mathcal{C}(i)$ *where* $\langle\langle i, \rho\rangle, \hat{\rho}, q\rangle = \check{s}$, *and*
2. $\Psi; \mathcal{P}; \Phi \rhd\!\!\rhd \check{\sigma}\, \check{s}$ *and* $\Psi; \Phi \rhd\!\!\rhd \lfloor \check{\sigma}\, \check{s} \rfloor\, \sigma$

it follows that

1. $\mathrm{Int}_{\Psi,\mathcal{P},\Phi}(\lfloor \check{\sigma}\, \check{s} \rfloor\, \sigma, \check{s}')$ *for some* $\check{s}'$, *and*
2. $\lfloor \check{s}' \rfloor \in \lfloor \check{\sigma}\, \check{s} \rfloor\, \sigma$ *implies*
   (a) $\mathrm{Head}(\check{\pi}') \sim \check{s}'\ (P', \phi')$ *for* $P' = \mathrm{VC}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C}', \check{\pi}'\rangle)$ *and some* ϕ' *and some* $\langle \mathcal{C}', \check{\pi}'\rangle \in \mathcal{S}$ *such that* $\Psi; \mathcal{P}; \Phi\, \widetilde{\rhd\!\!\rhd}\, \check{\pi}'$, *and*
   (b) *either* $\mathcal{C}' = \mathcal{C}$ *and* $\phi'(x) = \phi(x)$ *for all* $x \in \mathrm{CFV}(\mathcal{C})$ *or* $\mathrm{Enab}_{\Psi,\mathcal{P},\Phi}(\mathcal{C}', \mathcal{S}', \phi')$ *for some* $\mathcal{S}' \subseteq \mathcal{S}$

where $\mathrm{CFV}(\mathcal{C}) = \left(\bigcup_{i \in \mathrm{dom}\,\mathcal{C}} \mathrm{FV}(\mathcal{C}(i))\right) \smallsetminus (\mathit{XReg} \cup \mathrm{dom}\,\Gamma_{\mathcal{P}})$

The prefix property is preserved by specification evaluation; the head and tail states of an execution are prefixes of its VC:

Lemma 4.25 (Continuity)

1. $\check{t} \preceq P$ *if* $\check{t}' \preceq P'$ *such that* $\mathrm{Junct}(\mathrm{PreVC}_{\mathcal{P}}(\check{t}, p^{\mathsf{a}}), P, P')$ *where* $\langle \mathcal{C}', \check{t}'\rangle = \mathrm{Eval}_{\mathcal{P}}(\langle \mathcal{C}, \check{t}\rangle, p^{\mathsf{a}})$
2. $\check{t} \preceq P$ *if* $\mathrm{Out}_{\mathcal{P},\Phi}(\check{t}, p^{\mathsf{t}}) \preceq P$
3. $\check{t} \preceq P$ *if* $\check{t}' \preceq P'$ *such that* $\mathrm{Junct}(\mathrm{PreVC}_{\mathcal{P},\Phi}(\check{t}, p^{\mathsf{t}}), P, P')$ *where* $\langle \mathcal{C}', \check{t}'\rangle = \mathrm{Eval}_{\mathcal{P},\Phi}(\langle \mathcal{C}, \check{t}\rangle, p^{\mathsf{t}})$
4. $\check{t} \preceq P$ *if* $\check{t}' \preceq P'$ *such that* $\mathrm{Junct}(\mathrm{PreVC}_{\mathcal{P},\Phi}(\check{t}, p^{\mathsf{c}}), P, P')$ *where* $\langle \mathcal{C}', \check{t}'\rangle = \mathrm{Eval}_{\mathcal{P},\Phi}(\langle \mathcal{C}, \check{t}\rangle, p^{\mathsf{c}})$
5. $\mathrm{Tail}(\check{\pi}) \preceq \mathrm{VC}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C}, \check{\pi}\rangle)$
6. $\mathrm{Head}(\check{\pi}) \preceq \mathrm{VC}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C}, \check{\pi}\rangle)$

PROOF:
Item 1 by cases using Definition 4.11 and def. PreVC/Eval
Item 2 by Lemma 4.13 and by def. Out
Item 3 by cases using items 1 and 2, Lemma 4.13, and def. PreVC/Eval
Item 4 by item 3 and by def. PreVC/Eval
Item 5 by cases using Lemma 4.13 and def. Conj
Item 6 by item 5 and by Lemma 4.23

□

The enabled status of a trace context is preserved by expansion:

Lemma 4.26 (Enabled Context)

1. $\mathrm{Enab}_{\Psi,\mathcal{P},\Phi}(\mathcal{C},\mathcal{S},\phi)$ *and* $\phi'(x) = \phi(x)$ *for all* $x \in \mathrm{CFV}(\mathcal{C})$ *implies* $\mathrm{Enab}_{\Psi,\mathcal{P},\Phi}(\mathcal{C},\mathcal{S},\phi')$

2. $\mathrm{Enab}_{\Psi,\mathcal{P},\Phi}(\mathcal{C},\mathcal{S},\phi)$ *and* $\phi \subseteq \phi'$ *implies* $\mathrm{Enab}_{\Psi,\mathcal{P},\Phi}(\mathcal{C},\mathcal{S},\phi')$

3. $\mathrm{Enab}_{\Psi,\mathcal{P},\Phi}(\mathcal{C},\mathcal{S},\phi)$ *and* $\mathcal{S} \subseteq \mathcal{S}'$ *implies* $\mathrm{Enab}_{\Psi,\mathcal{P},\Phi}(\mathcal{C},\mathcal{S}',\phi)$

PROOF:
Item 1 by Definition 4.14 and Lemma 4.49
Item 2 by Definition 4.14 and item 1
Item 3 by Definition 4.14

□

The fixed point of a trace schedule contains itself; fixed points are closed under reachable executions; fixed points are idempotent:

Lemma 4.27 (Fixed-Point Closure)

1. $\mathcal{S}_1 \cup \mathcal{S}_2 \subseteq \mathrm{Fix}_{\Psi,\mathcal{P},\Phi}(\mathcal{S}_1,\mathcal{S}_2)$,

2. $\langle \mathcal{C}, \breve{\pi}\rangle \in \mathcal{S}_2 \setminus \mathcal{S}_1$ *implies* $\mathrm{Next}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C}, \breve{\pi}\rangle) \subseteq \mathrm{Fix}_{\Psi,\mathcal{P},\Phi}(\mathcal{S}_1,\mathcal{S}_2)$, *and*

3. $\mathrm{Fix}_{\Psi,\mathcal{P},\Phi}(\mathcal{S}_1, \mathrm{Fix}_{\Psi,\mathcal{P},\Phi}(\mathcal{S}_1,\mathcal{S}_2)) \subseteq \mathrm{Fix}_{\Psi,\mathcal{P},\Phi}(\mathcal{S}_1,\mathcal{S}_2)$

if $\mathrm{Fix}_{\Psi,\mathcal{P},\Phi}(\mathcal{S}_1,\mathcal{S}_2)$ *is finite*

PROOF:
Item 1 by induction on the size of $\mathrm{Fix}_{\Psi,\mathcal{P},\Phi}(\mathcal{S}_1,\mathcal{S}_2) \setminus \mathcal{S}_1$
Item 2 by item 1
Item 3 by induction on the size of $\mathrm{Fix}_{\Psi,\mathcal{P},\Phi}(\mathcal{S}_1,\mathcal{S}_2) \setminus \mathcal{S}_1$ using item 2

□

The following lemmas through Lemma 4.33 show that simulation is preserved by specification evaluation:

Lemma 4.28 (Abstraction-Specification Evaluation)

1. $\check{t}' \preceq P'$ *such that* $\mathrm{Junct}(\mathrm{PreVC}_{\mathcal{P}}(\check{t}, p^{\mathsf{a}}), P, P')$ *and* $\vdash \mathcal{P}\ \mathsf{sp}$ *where* $\langle \mathcal{C}', \check{t}'\rangle = \mathrm{Eval}_{\mathcal{P}}(\langle \mathcal{C}, \check{t}\rangle, p^{\mathsf{a}})$, *and*

2. $\check{t} \sim \check{s}\ (P, \phi)$ *such that* $\mathrm{Sat}_{\mathcal{P}}(\check{s}, \check{s}', p^{\mathsf{a}})$

implies $\check{t}' \sim \check{s}'\ (P', \phi')$ *for some* ϕ' *such that* $\mathcal{C}' \neq \emptyset$ *implies* $\mathcal{C}' = \mathcal{C}$ *and* $\phi \subseteq \phi'$

PROOF:
(sketch)
Case: $p^{\mathsf{a}} = \mathtt{all}\langle P_1, P_1'\rangle$
$\vDash (\mathrm{Abs}_{\mathcal{P}}(\eta_*, \hat{\eta}_{\mathcal{P}}))(\theta_{\langle\langle i,\eta_*\rangle, \hat{\eta}_{\mathcal{P}}, \bullet\rangle}(P_1') \supset P_2')$ where P_2' is the suffix of P'
by Definition 4.11 and def. PreVC/Eval
$\vDash_{\phi_{\check{s}'}} P_1'$ by Definition 4.12
let $\mathrm{dom}\,\phi' = \mathrm{FV}(\eta_*) \cup \mathrm{FV}(\hat{\eta}_{\mathcal{P}})$
let $\phi'(\eta_*(r)) = \rho'(r)$ for all $r \in \mathit{Reg}$ and $\phi'(\hat{\eta}_{\mathcal{P}}(\hat{r})) = \hat{\rho}'(\hat{r})$ for all $\hat{r} \in \mathrm{dom}\,\Gamma_{\mathcal{P}}$

$\mathcal{V}_{\phi'}(\eta_*) = \rho'$ and $\mathcal{V}_{\phi'}(\hat{\eta}_{\mathcal{P}}) = \hat{\rho}'$ by def. $\mathcal{V}$
$\models_{\phi'} \theta_{\langle\langle i,\eta_*\rangle,\hat{\eta}_{\mathcal{P}},\bullet\rangle}(P_1') \supset P_2'$ by def. $\models$
$\models_{\phi'} P_2'$ by def. $\models$
$\check{t}' \sim \check{s}'\ (P', \phi')$ by Definition 4.7
Case: $p^{\mathsf{a}} = \texttt{some}\langle X, P_1, P_1'\rangle$
$\models_{\phi}$ $(\mathrm{Abs}_{\mathcal{P}}(\theta_1, \theta_2))(\theta_{\langle t',\hat{\eta}',Q'\rangle}(P_1') \supset P_2')$ if $q \neq 0$ where P_2' is the suffix of P'
by Definition 4.11, Definition 4.7, and def. PreVC/Eval/Gen
$q' \leq q$, $\models_{\phi_{\check{s}'}} P_1'$, and $\phi_{\check{s}'}(x) = \phi_{\check{s}}(x)$ for all $x \in (\mathit{Reg} \cup \mathrm{dom}\,\Gamma_{\mathcal{P}}) \smallsetminus X$
by Definition 4.12
let $\phi' = \phi[y_1 \mapsto \rho'(r_1), \ldots, y_k \mapsto \rho'(r_k), y_1' \mapsto \hat{\rho}'(\hat{r}_1), \ldots, y_{k'}' \mapsto \hat{\rho}'(\hat{r}_{k'})]$
where $\{r_1, \ldots, r_k\} = \mathrm{dom}\,\theta_1$, $\{\hat{r}_1, \ldots, \hat{r}_{k'}\} = \mathrm{dom}\,\theta_2$
and $y_1 = \theta_1(r_1), \ldots, y_k = \theta_1(r_k), y_1' = \theta_2(\hat{r}_1), \ldots, y_{k'}' = \theta_2(\hat{r}_{k'})$
$\mathcal{V}_{\phi'}(\eta') = \rho'$ and $\mathcal{V}_{\phi'}(\hat{\eta}') = \hat{\rho}'$ by def. $\mathcal{V}$
$\models_{\phi'} \theta_{\langle t',\hat{\eta}',Q'\rangle}(P_1') \supset P_2'$ if $q \neq 0$ by def. $\models$
$\models_{\phi'} P_2'$ if $q \neq 0$ by def. $\models$
$\check{t}' \sim \check{s}'\ (P', \phi')$ by Definition 4.7

□

Lemma 4.29 (Incoming-Transition Evaluation)

1. *$\check{t}' \preceq P$ such that $\vdash \mathcal{P}$ $\texttt{sp}$ where $\check{t}' = \mathrm{In}_{\mathcal{P},\Phi}(\check{t}, p^{\mathsf{t}})$,*

2. *$\check{t} \sim \check{s}_0\ (P, \phi)$ such that $\mathrm{Sat}_{\mathcal{P},\Phi}(\check{s}_0, \check{s}, p^{\mathsf{t}})$ and $\mathrm{Avail}_{\Psi,\Phi}(s, p^{\mathsf{t}})$ and $p^{\mathsf{t}} = \texttt{tor}\ldots$ implies $\check{s} = \check{s}_0$ where $\langle s, \hat{\rho}, q\rangle = \check{s}$, and*

3. *$\Psi; \mathcal{P}; \Phi \rhd\!\!\rhd \check{\sigma}\ \check{s}$ and $\Psi; \Phi \rhd\!\!\rhd \lfloor \check{\sigma}\ \check{s} \rfloor\ \sigma$*

implies

1. *$\mathrm{Int}_{\Psi,\mathcal{P},\Phi}(\lfloor \check{\sigma}\ \check{s}\rfloor\ \sigma, \check{s}')$ for some $\check{s}'$ such that $p^{\mathsf{t}} \neq \texttt{tor}\ldots$ implies $\check{s}' = \check{s}$, and*

2. *$\lfloor \check{s}' \rfloor \in \lfloor \check{\sigma}\ \check{s} \rfloor\ \sigma$ implies*

   (a) $\check{t}' \sim \check{s}_1\ (P, \phi')$ for some $\check{s}_1$ such that $\check{s}_0 = \check{s}$ implies $\check{s}_1 = \check{s}'$ and some ϕ' such that $\phi \subseteq \phi'$, and

   (b) $\mathrm{Sat}_{\mathcal{P}}(\check{s}_1, \check{s}', \mathrm{ASp}(p^{\mathsf{t}}))$ where $\check{s}_1$ is a function of $\check{s}_0$, p^{t}, $\mathcal{P}$, and Φ

PROOF:
(sketch)
Case: $p^{\mathsf{t}} = \texttt{tor}\ldots$
$\check{t}' \sim \check{s}'\ (P, \phi')$ by def. In and Lemma 4.15
$q' \neq 0$ implies $\mathrm{Sat}_{\mathcal{P}}(\check{s}', \check{s}', \mathrm{ASp}(p^{\mathsf{t}}))$ by Definition 4.12
$\Psi; \mathcal{P}; \Phi \rhd\!\!\rhd \check{\sigma}\ \check{s} \to \check{s}'$ by applying ea_2 and ee_2 using Definition 4.13
apply Lemma 4.22 if $q' = 0$
otherwise, $\mathrm{Int}_{\Psi,\mathcal{P},\Phi}(\lfloor \check{\sigma}\ \check{s} \rfloor\ \sigma, \check{s}')$ by Definition 4.9
Other cases by Definition 4.9, Definition 4.12, and def. In
(letting $\check{s}_1 = \check{s}_0, \check{s}' = \check{s}, \phi' = \phi$)

□

Lemma 4.30 (Outgoing-Transition Evaluation)

1. *$\check{t}' \preceq P$ such that $\vdash \mathcal{P}$ sp where $\check{t}' = \mathrm{Out}_{\mathcal{P},\Phi}(\check{t}, p^{\mathsf{t}})$,*

2. *$\check{t} \sim \check{s}$ (P, ϕ) such that $p^{\mathsf{t}} \neq \mathtt{tor}\ldots$ implies $\mathrm{Avail}_{\Psi,\Phi}(s, p^{\mathsf{t}})$ where $\langle s, \hat{\rho}, q\rangle = \check{s}$, and*

3. *$\Psi; \mathcal{P}; \Phi \rhd\!\!\rhd \check{\sigma}\ \check{s}$ and $\Psi; \Phi \rhd\!\!\rhd \lfloor \check{\sigma}\ \check{s} \rfloor\ \sigma$*

implies

1. *$\mathrm{Int}_{\Psi,\mathcal{P},\Phi}(\lfloor \check{\sigma}\ \check{s} \rfloor\ \sigma, \check{s}')$ for some $\check{s}'$, and*

2. *$\lfloor \check{s}' \rfloor \in \lfloor \check{\sigma}\ \check{s} \rfloor\ \sigma$ implies $\check{t}' \sim \check{s}'$ (P, ϕ') for some ϕ' such that $\phi \subseteq \phi'$*

PROOF:
(sketch)
Case: $p^{\mathsf{t}} = \mathtt{tol}\ldots$
  $\check{t}' \sim \check{s}'$ (P, ϕ') by def. Out and Lemma 4.15
  $\Psi; \mathcal{P}; \Phi \rhd\!\!\rhd \check{\sigma}\ \check{s} \to \check{s}'$ if $\sigma \neq \cdot$ by applying ea_2 and ee_2 using Definition 4.13
  $\mathrm{Int}_{\Psi,\mathcal{P},\Phi}(\lfloor \check{\sigma}\ \check{s} \rfloor\ \sigma, \check{s}')$ by Definition 4.9
Case: $p^{\mathsf{t}} = \mathtt{tos}\ldots$
  let ϕ_1 be ϕ with new vars mapped to values from ρ' such that $\mathcal{V}_{\phi_1}(\eta') = \rho'$
  $\langle \mathcal{P}, \hat{\eta}, Q(\mathrm{Abs}_{\Gamma_{Reg}}(\theta))\rangle \sim \langle \mathcal{P}, \mathcal{V}_{\phi_1}(\hat{\eta}), q\rangle$ (P, ϕ_1) by def. Out, $\models$, and Abs
  $\check{t}' \sim \check{s}'$ (P, ϕ') by Lemma 4.15
  $\Psi; \mathcal{P}; \Phi \rhd\!\!\rhd \check{\sigma}\ \check{s} \to \check{s}'$ if $\sigma \neq \cdot$ by applying ea_2 and ee_2 using Definition 4.13
  $\mathrm{Int}_{\Psi,\mathcal{P},\Phi}(\lfloor \check{\sigma}\ \check{s} \rfloor\ \sigma, \check{s}')$ by Definition 4.9
Other cases by Definition 4.9, and def. Out (letting $\check{s}' = \check{s}$)

□

Lemma 4.31 (Transition-Specification Evaluation)

1. *$\check{t}' \preceq P'$ such that $\mathrm{Junct}(\mathrm{PreVC}_{\mathcal{P},\Phi}(\check{t}, p^{\mathsf{t}}), P, P')$ and $\vdash \mathcal{P}$ sp where $\langle \mathcal{C}', \check{t}'\rangle = \mathrm{Eval}_{\mathcal{P},\Phi}(\langle \mathcal{C}, \check{t}\rangle, p^{\mathsf{t}})$,*

2. *$\check{t} \sim \check{s}_0$ (P, ϕ) such that $\mathrm{Sat}_{\mathcal{P},\Phi}(\check{s}_0, \check{s}, p^{\mathsf{t}})$ and $\mathrm{Avail}_{\Psi,\Phi}(s, p^{\mathsf{t}})$ and $p^{\mathsf{t}} = \mathtt{tor}\ldots$ implies $\check{s} = \check{s}_0$ where $\langle s, \hat{\rho}, q\rangle = \check{s}$, and*

3. *$\Psi; \mathcal{P}; \Phi \rhd\!\!\rhd \check{\sigma}\ \check{s}$ and $\Psi; \Phi \rhd\!\!\rhd \lfloor \check{\sigma}\ \check{s} \rfloor\ \sigma$*

implies

1. *$\mathrm{Int}_{\Psi,\mathcal{P},\Phi}(\lfloor \check{\sigma}\ \check{s} \rfloor\ \sigma, \check{s}')$ for some $\check{s}'$, and*

2. *$\lfloor \check{s}' \rfloor \in \lfloor \check{\sigma}\ \check{s} \rfloor\ \sigma$ implies $\check{t}' \sim \check{s}'$ (P', ϕ') for some ϕ' such that $\mathcal{C}' \neq \emptyset$ implies $\mathcal{C}' = \mathcal{C}$ and $\phi \subseteq \phi'$*

PROOF:
By Lemma 4.29, Lemma 4.28, and Lemma 4.30 using def. PreVC/Eval

□

Lemma 4.32 (Context-Specification Evaluation)

1. $\check{t}' \preceq P'$ *such that* $\mathrm{Junct}(\mathrm{PreVC}_{\mathcal{P},\Phi}(\check{t}, p^{\mathsf{c}}), P, P')$ *and* $\vdash \mathcal{P}\ \mathsf{sp}$
   where $\langle \mathcal{C}', \check{t}' \rangle = \mathrm{Eval}_{\mathcal{P},\Phi}(\langle \mathcal{C}, \check{t} \rangle, p^{\mathsf{c}})$,
2. $\check{t} \sim \check{s}_0\ (P, \phi)$ *such that* $\mathrm{Sat}_{\mathcal{P},\Phi}(\check{s}_0, \check{s}, p^{\mathsf{c}})$ *and* $\mathrm{Avail}_{\Psi,\Phi}(s, p^{\mathsf{c}})$
   and $p^{\mathsf{t}} \neq \mathtt{tor}\ldots$ *where* $p^{\mathsf{c}} = \langle i', p^{\mathsf{t}} \rangle$ *and* $\langle s, \hat{\rho}, q \rangle = \check{s}$, *and*
3. $\Psi; \mathcal{P}; \Phi \rhd\!\!\rhd \check{\sigma}\ \check{s}$ *and* $\Psi; \Phi \rhd\!\!\rhd \lfloor \check{\sigma}\ \check{s} \rfloor\, \sigma$

implies

1. $\mathrm{Int}_{\Psi,\mathcal{P},\Phi}(\lfloor \check{\sigma}\ \check{s} \rfloor\, \sigma, \check{s}')$ *for some* $\check{s}'$, *and*
2. $\lfloor \check{s}' \rfloor \in \lfloor \check{\sigma}\ \check{s} \rfloor\, \sigma$ *implies* $\check{t}' \sim \check{s}'\ (P', \phi')$ *for some* ϕ'
   such that $\mathcal{C}' \neq \emptyset$ *implies* $\mathcal{C}' = \mathcal{C}$ *and* $\phi \subseteq \phi'$

PROOF:
By Lemma 4.31 using Definition 4.12 and def. PreVC/Eval

□

Lemma 4.33 (Link-Specification Evaluation)

1. $\Psi; \mathcal{P}; \Phi \mathrel{\widetilde{\rhd\!\!\rhd}} <\!\cdot \check{t} \ast\!\!\succ \langle p^{\mathsf{l}}_0, \ldots, p^{\mathsf{l}}_k \rangle$ *or* $\Psi; \mathcal{P}; \Phi \mathrel{\tilde{\rhd}} \check{t} \ast\!\!\succ \langle p^{\mathsf{l}}_0, \ldots, p^{\mathsf{l}}_k \rangle$
   for $p^{\mathsf{l}} \in \{p^{\mathsf{l}}_0, \ldots, p^{\mathsf{l}}_k\}$ *and* $\langle \langle \mathcal{C}'_0, \check{t}'_0 \rangle, \ldots, \langle \mathcal{C}'_{k'}, \check{t}'_{k'} \rangle \rangle = \mathrm{Eval}_{\mathcal{P},\Phi}(\langle \mathcal{C}, \check{t} \rangle, p^{\mathsf{l}})$
   and $\vdash \mathcal{P}\ \mathsf{sp}$,
2. $\mathrm{CFV}(\mathcal{C}) \subseteq \mathrm{dom}\, \phi$ *and for all* $j \in \{0, \ldots, k'\}$,
   (a) $\check{t}'_j \preceq P'_j$ *such that* $\mathrm{Junct}((\mathrm{PreVC}_{\mathcal{P},\Phi}(\check{t}, p^{\mathsf{l}}))_j, P_j, P'_j)$, *and*
   (b) $\check{t} \sim \check{s}\ (P_j, \phi)$
3. $\mathrm{Sat}_{\mathcal{P},\Phi}(\check{s}, p^{\mathsf{l}})$ *and* $\mathrm{Avail}_{\Psi,\Phi}(s, p^{\mathsf{l}})$ *where* $\langle s, \hat{\rho}, q \rangle = \check{s}$, *and*
4. $\Psi; \mathcal{P}; \Phi \rhd\!\!\rhd \check{\sigma}\ \check{s}$ *and* $\Psi; \Phi \rhd\!\!\rhd \lfloor \check{\sigma}\ \check{s} \rfloor\, \sigma$

implies

1. $\mathrm{Int}_{\Psi,\mathcal{P},\Phi}(\lfloor \check{\sigma}\ \check{s} \rfloor\, \sigma, \check{s}')$ *for some* $\check{s}'$, *and*
2. $\lfloor \check{s}' \rfloor \in \lfloor \check{\sigma}\ \check{s} \rfloor\, \sigma$ *implies*
   (a) $\check{t}'_0 \sim \check{s}'\ (P'_0, \phi')$ *for some* ϕ' *such that* $\mathrm{CFV}(\mathcal{C}'_0) \subseteq \mathrm{dom}\, \phi'$, *and*
   (b) *either* $\mathrm{Enab}(i'')$ *for all* $i'' \in \mathrm{dom}\, \mathcal{C}'_0$
   or $\mathcal{C} \subseteq \mathcal{C}'_0$ *and* $\phi \subseteq \phi'$ *and* $\mathrm{Enab}(i'')$ *for all* $i'' \in \mathrm{dom}\, \mathcal{C}'_0 \setminus \mathrm{dom}\, \mathcal{C}$

where $\mathrm{Enab}(i'')$ *iff for all* $\check{s}'', \check{\sigma}'', \sigma''$ *such that*

1. $\vDash_{\phi' \cup \phi_{\check{s}''}} \mathcal{C}'_0(i'')$ *where* $\check{s}'' = \langle \langle i'', \rho'' \rangle, \hat{\rho}'', q'' \rangle$, *and*
2. $\Psi; \mathcal{P}; \Phi \rhd\!\!\rhd \check{\sigma}''\ \check{s}''$ *and* $\Psi; \Phi \rhd\!\!\rhd \lfloor \check{\sigma}''\ \check{s}'' \rfloor\, \sigma''$

it follows that

1. $\mathrm{Int}_{\Psi,\mathcal{P},\Phi}(\lfloor\check{\sigma}''\,\check{s}''\rfloor\,\sigma'',\check{s}''')$ *for some* $\check{s}'''$*, and*
2. $\lfloor\check{s}'''\rfloor \in \lfloor\check{\sigma}''\,\check{s}''\rfloor\,\sigma''$ *implies*
   (a) $\check{t}'_j \sim \check{s}'''\ (P'_j,\phi''')$ *for some* $j \in \{0,\ldots,k'\}$ *and some* ϕ'''*, and*
   (b) $j = 0$ *implies* $\phi'''(x) = \phi'(x)$ *for all* $x \in \mathrm{CFV}(\mathcal{C}'_0)$
   $j \neq 0$ *and* $\mathcal{C}'_j \neq \emptyset$ *implies* $\mathcal{C}'_j = \mathcal{C}$ *and* $\phi \subseteq \phi'''$

PROOF:
(sketch)
$\Psi;\mathcal{P};\Phi \rhd\!\!\rhd \check{\sigma}_1$ such that $\lfloor\check{\sigma}_1\rfloor \leq \lfloor\check{\sigma}\,\check{s}\rfloor\,\sigma$
by Lemma 4.29, Definition 4.9, and def. PreVC/Eval/Eval0
Case: $\lfloor\check{\sigma}_1\rfloor \neq \lfloor\check{\sigma}\,\check{s}\rfloor\,\sigma, \check{\sigma}_1 = \check{\sigma}_2\,\check{s}_2$
$\check{t}^1_0 \sim \check{s}_1\ (P_0,\phi_1)$ by Lemma 4.29 (where $\check{t}^1_0 = \mathrm{In}_{\mathcal{P},\Phi}(\check{t},p^{\mathsf{t}})$)
$\check{t}^2_0 \sim \check{s}_2\ (P'_0,\phi_2)$ by Lemma 4.28 (where $\langle\mathcal{C}^2_0,\check{t}^2_0\rangle = \mathrm{Eval}_{\mathcal{P}}(\langle\mathcal{C},\check{t}^1_0\rangle,\mathrm{ASp}(p^{\mathsf{t}}))$)
$\mathrm{Int}_{\Psi,\mathcal{P},\Phi}(\lfloor\check{\sigma}_2\,\check{s}_2\rfloor\,\sigma_1,\check{s}')$ by Lemma 4.30
case: $\lfloor\check{s}'\rfloor \in \lfloor\check{\sigma}_2\,\check{s}_2\rfloor\,\sigma_1, q_2 \neq 0$
$\check{t}'_0 \sim \check{s}'\ (P'_0,\phi')$ by Lemma 4.30
let $\langle i_1,p^{\mathsf{t}}_1\rangle,\ldots,\langle i_{k'},p^{\mathsf{t}}_{k'}\rangle = p^{\mathsf{c}}_1,\ldots,p^{\mathsf{c}}_{k'}$
for each $j \in \{1,...,k'\}$ and all $\check{s}'',\check{\sigma}'',\sigma''$
such that $\vDash_{\phi'\cup\phi_{\check{s}''}} \mathcal{C}'_0(i_j)$, $\Psi;\mathcal{P};\Phi \rhd\!\!\rhd \check{\sigma}''\,\check{s}''$, and $\Psi;\Phi \rhd\!\!\rhd \lfloor\check{\sigma}''\,\check{s}''\rfloor\,\sigma''$
$\mathrm{Sat}_{\mathcal{P},\Phi}(\check{s}_2,\check{s}'',p^{\mathsf{c}}_j)$ and $p^{\mathsf{t}}_j \neq \mathtt{tor}\ldots$ by def. Eval/Eval0 and Lemma 4.34
$\mathrm{Avail}_{\Psi,\Phi}(s'',p^{\mathsf{c}}_j)$ by Lemma 4.35
case: $p^{\mathsf{c}}_j = \langle i,p^{\mathsf{t}}\rangle, k = 1$
$\mathrm{Int}_{\Psi,\mathcal{P},\Phi}(\lfloor\check{\sigma}''\,\check{s}''\rfloor\,\sigma'',\check{s}''')$ by def. In/Eval and Lemma 4.32
case: $\lfloor\check{s}'''\rfloor \in \lfloor\check{\sigma}''\,\check{s}''\rfloor\,\sigma''$
$\check{t}'_0 \sim \check{s}'''\ (P'_j,\phi''')$ by Lemma 4.32
$\phi'''(x) = \phi'(x)$ for all $x \in \mathrm{CFV}(\mathcal{C}'_0)$ by def. Eval/Need/In/Gen
case: $p^{\mathsf{c}}_j \neq \langle i,p^{\mathsf{t}}\rangle$ or $k \neq 1$
$\mathrm{Int}_{\Psi,\mathcal{P},\Phi}(\lfloor\check{\sigma}''\,\check{s}''\rfloor\,\sigma'',\check{s}''')$ by def. Eval, Lemma 4.29, and Lemma 4.32
case: $\lfloor\check{s}'''\rfloor \in \lfloor\check{\sigma}''\,\check{s}''\rfloor\,\sigma''$
$\check{t}'_j \sim \check{s}'''\ (P'_j,\phi''')$ by Lemma 4.32
$\mathrm{Enab}(i_j)$ for each $j \in \{1,...,k'\}$
case: $q_2 = 0$
apply Lemma 4.22 (choosing $\check{s}'$ such that $\lfloor\check{s}'\rfloor \notin \lfloor\check{\sigma}\,\check{s}\rfloor\,\sigma$)

□

Only the following four lemmas depend on our specific choices for tail-step specifications—all other lemmas hold for all possible specifications.

We can satisfy a context specification in a state if we can show that its Need is valid in the state:

Lemma 4.34 (Satisfied Context Specification)

1. $\Psi;\mathcal{P};\Phi \,\tilde{\rhd\!\!\rhd}\, {<\!\cdot}\langle t,\hat{\eta},Q\rangle \ast\!\!\succ \langle p^{\mathsf{l}}_1,\ldots,p^{\mathsf{l}}_k\rangle$ *or* $\Psi;\Phi\,\tilde{\rhd}\,t \ast\!\!\succ \langle p^{\mathsf{l}}_1,\ldots,p^{\mathsf{l}}_k\rangle$
   for $\langle p^{\mathsf{t}},p^{\mathsf{c}}_1,\ldots,p^{\mathsf{c}}_{k'}\rangle \in \{p^{\mathsf{l}}_1,\ldots,p^{\mathsf{l}}_k\}$ *and* $\langle i',p^{\mathsf{t}}\rangle \in \{p^{\mathsf{c}}_1,\ldots,p^{\mathsf{c}}_{k'}\}$*, and*

2. $\models_{\phi\cup\phi_{\langle\langle i',\rho'\rangle,\hat\rho',q'\rangle}} \mathrm{Need}_{\mathcal{P},\Phi}(\check t,\langle i',p^{\mathsf{t}}\rangle)$ *such that* $\mathcal{V}_\phi(\theta_{\check t}) = \phi_{\langle\langle i,\rho\rangle,\hat\rho,q\rangle}$ *and* $\mathrm{dom}\,\phi \cap (\mathit{XReg} \cup \mathit{PReg}) = \emptyset$ *and* $q' \le q$

implies $\mathrm{Sat}_{\mathcal{P},\Phi}(\langle\langle i,\rho\rangle,\hat\rho,q\rangle,\langle\langle i',\rho'\rangle,\hat\rho',q'\rangle,\langle i',p^{\mathsf{t}}\rangle)$ *and* $p^{\mathsf{t}} \neq \mathtt{tor}\ldots$

PROOF:
(sketch)
$p^{\mathsf{t}} = \mathtt{id}\langle\mathtt{some}\langle X,P,P'\rangle\rangle$ or $p^{\mathsf{t}} = \mathtt{tol}\langle\mathtt{some}\langle X,P,P'\rangle,i''\rangle$ by inspection
$\models_{\phi_{\check s'}} P'$ by def. Need/$\models$
for each $x \in (\mathit{Reg} \cup \mathrm{dom}\,\Gamma_{\mathcal{P}}) \smallsetminus X$
$\mathcal{V}_{\phi\cup\phi_{\check s'}}(x) = \mathcal{V}_{\phi\cup\phi_{\check s'}}(\theta_{\check t}(x))$ by def. $\models$
$\phi_{\check s'}(x) = \phi_{\langle\langle i',\rho\rangle,\hat\rho,q\rangle}(x)$ by def. $\mathcal{V}$
$\mathrm{Sat}_{\mathcal{P}}(\langle\langle i',\rho\rangle,\hat\rho,q\rangle,\check s',\mathtt{some}\langle X,P,P'\rangle)$ by Definition 4.12
$\mathrm{Sat}_{\mathcal{P},\Phi}(\langle\langle i,\rho\rangle,\hat\rho,q\rangle,\check s',\langle i',p^{\mathsf{t}}\rangle)$ by Definition 4.12

□

If a context specification is satisfied with respect to a state $\langle s',\hat\rho',q'\rangle$, then it is available with respect to s':

Lemma 4.35 (Available Context Specification)

1. $\Psi;\mathcal{P};\Phi \,\tilde{\triangleright\!\triangleright}\, <\!\cdot\langle\langle i,\eta\rangle,\hat\eta,Q\rangle \ast\!\succ \langle p^{\mathsf{l}}_1,\ldots,p^{\mathsf{l}}_k\rangle$ *or* $\Psi;\Phi\,\tilde{\triangleright}\langle i,\eta\rangle \ast\!\succ \langle p^{\mathsf{l}}_1,\ldots,p^{\mathsf{l}}_k\rangle$ *for* $\langle p^{\mathsf{t}},p^{\mathsf{c}}_1,\ldots,p^{\mathsf{c}}_{k'}\rangle \in \{p^{\mathsf{l}}_1,\ldots,p^{\mathsf{l}}_k\}$ *and* $\langle i',p^{\mathsf{t}}\rangle \in \{p^{\mathsf{c}}_1,\ldots,p^{\mathsf{c}}_{k'}\}$, *and*

2. $\mathrm{Sat}_{\mathcal{P},\Phi}(\langle\langle i_0,\mathcal{V}_\phi(\eta)\rangle,\hat\rho,q\rangle,\langle\langle i',\rho'\rangle,\hat\rho',q'\rangle,\langle i',p^{\mathsf{t}}\rangle)$

implies $\mathrm{Avail}_{\Psi,\Phi}(\langle i',\rho'\rangle,\langle i',p^{\mathsf{t}}\rangle)$

PROOF:
(sketch)
Case: $p^{\mathsf{t}} = \mathtt{id}\langle\mathtt{some}\langle X,\top,P'\rangle\rangle$
trivial
Case: $i' \in \mathrm{Ret}_\Phi(\underline{n}), \underline{n} \in \mathrm{Proc}(\Phi), i \dot+ 1 \in \mathrm{Jump}_\Phi(i), \mathtt{ra} \notin X, \eta(\mathtt{ra}) = \overline{i \dot+ 1}$,
$p^{\mathsf{t}} = \mathtt{tol}\langle\mathtt{some}\langle X,\top,P'\rangle, i \dot+ 1\rangle$
$\mathrm{Sat}_{\mathcal{P}}(\langle\langle i',\rho\rangle,\hat\rho,q\rangle,\check s',\mathtt{some}\langle X,\top,P'\rangle)$ by Definition 4.12
$\phi_{\langle\langle i',\rho\rangle,\hat\rho,q\rangle}(\mathtt{ra}) = i \dot+ 1$ by Definition 4.12 and def. $\mathcal{V}$
for all σ_0,σ such that $\Psi;\Phi \triangleright\!\triangleright \sigma_0\,\langle i',\rho'\rangle\,\sigma$ and $\sigma \neq \cdot$
$\sigma = \rightarrow s''\,\sigma'$ or $\sigma = \cdot\!>$ by inspection
$\Phi \triangleright \langle i',\rho'\rangle \xrightarrow{\mathsf{m}} s''_0$ by inverting me and ma rules
$s''_0 = \langle i \dot+ 1,\rho'\rangle$ by inspecting m rule
$\rightarrow \langle i \dot+ 1,\rho'\rangle \le \sigma$ by def. Skip, Stop, and by inspecting ma rules
$\mathrm{Avail}_{\Psi,\Phi}(\langle i',\rho'\rangle,\langle i',p^{\mathsf{t}}\rangle)$ by Definition 4.13
Case: $\Phi_i = \mathtt{inv}\ P, X, i \dot+ 1 \in \mathrm{Jump}_\Phi(i), p^{\mathsf{t}} = \mathtt{tol}\langle\mathtt{some}\langle X,P,P\rangle, i \dot+ 1\rangle, i' = i$
similar to previous case

□

A link specification is satisfied with respect to any state for which we can show availability and simulation:

Lemma 4.36 (Satisfied Link Specification)

1. $\Psi; \mathcal{P}; \Phi \,\widetilde{\vartriangleright\!\!\vartriangleright}\, {<}\!\cdot \check{t} *\!\!\succ \langle p^{\mathsf{l}}_1, \ldots, p^{\mathsf{l}}_k \rangle$ *or* $\Psi; \mathcal{P}; \Phi \,\tilde{\vartriangleright}\, \check{t} *\!\!\succ \langle p^{\mathsf{l}}_1, \ldots, p^{\mathsf{l}}_k \rangle$ *and* $\vdash \mathcal{P}$ sp,

2. $\langle \mathcal{C}', \check{t}' \rangle = (\mathrm{Eval}_{\mathcal{P},\Phi}(\langle \mathcal{C}, \check{t} \rangle, p^{\mathsf{l}}))_0$ *for* $p^{\mathsf{l}} \in \{p^{\mathsf{l}}_1, \ldots, p^{\mathsf{l}}_k\}$ *such that* $\check{t}' \preceq P'$ *and* $\mathrm{Junct}((\mathrm{PreVC}_{\mathcal{P},\Phi}(\check{t}, p^{\mathsf{l}}))_0, P, P')$,

3. $\check{t} \sim \langle s, \hat{\rho}, q \rangle \ (P, \phi)$ *for* $q \neq 0$, *and*

4. $\mathrm{Avail}_{\Psi,\Phi}(s, p^{\mathsf{l}})$

implies $\mathrm{Sat}_{\mathcal{P},\Phi}(\langle s, \hat{\rho}, q \rangle, p^{\mathsf{l}})$

PROOF:
(sketch)
Case: $\Phi_i = \texttt{cond}\ cop\ r, n, \underline{n} \dot{+} i \dot{+} 1, i \dot{+} 1 \in \mathrm{Jump}_\Phi(i)$,
$p^{\mathsf{l}}_1 \ldots p^{\mathsf{l}}_k = \langle \texttt{tol}\langle \texttt{some}\langle \emptyset, \top, cop(r) \rangle, \underline{n} \dot{+} i \dot{+} 1 \rangle \rangle$
$\langle \texttt{tol}\langle \texttt{some}\langle \emptyset, \top, (\neg cop)(r) \rangle, i \dot{+} 1 \rangle \rangle$
case: $p^{\mathsf{l}} = \langle \texttt{tol}\langle \texttt{some}\langle \emptyset, \top, cop(r) \rangle, \underline{n} \dot{+} i \dot{+} 1 \rangle \rangle$
for all σ_0, σ such that $\Psi; \Phi \vartriangleright\!\!\vartriangleright \sigma_0\, s\, \sigma$ and $\sigma \neq \cdot$
$\rightarrow \langle \underline{n} \dot{+} i \dot{+} 1, \mathcal{V}_\phi(\eta) \rangle \leq \sigma$ by Definition 4.13 (where $\check{t} = \langle \langle i, \eta \rangle, \hat{\eta}, Q \rangle$)
$\mathcal{J}(cop)(\mathcal{V}_\phi(\eta)(r)) \neq 0$ by inspecting me, ma, and m rules
$\models_{\phi_{\check{s}}} cop(r)$ by def. $\models$ (where $\check{s} = \langle s, \hat{\rho}, q \rangle$)
$\mathrm{Sat}_{\mathcal{P},\Phi}(\check{s}, p^{\mathsf{l}})$ by Definition 4.12
case: $p^{\mathsf{l}} = \langle \texttt{tol}\langle \texttt{some}\langle \emptyset, \top, (\neg cop)(r) \rangle, i \dot{+} 1 \rangle \rangle$
similar to previous case
Case: $\Phi_i = \texttt{call}\ n, p^{\mathsf{l}} = \langle \texttt{tos}\langle i \dot{+} 1 \rangle \rangle$
$\mathrm{Sat}_{\mathcal{P},\Phi}(\check{s}, p^{\mathsf{l}})$ by Definition 4.12
Case: $\Phi_i = \texttt{call}\ n, \mathcal{P} \vdash P_1\ \mathsf{spec}, p^{\mathsf{l}} = \langle \texttt{tor}\langle \underline{n}, \texttt{all}\langle P_1, P_1 \rangle \rangle, p^{\mathsf{c}}_1, \ldots, p^{\mathsf{c}}_{k'} \rangle$
for all $\hat{\rho}'$, q' s.t. $\Phi; \langle i, \mathcal{V}_\phi(\eta) \rangle \rightarrow \langle \underline{n}, \mathcal{V}_\phi(\eta) \rangle \triangleright \langle \mathcal{P}, \hat{\rho}, q \rangle \stackrel{\mathsf{s}*}{\rightarrow} \langle \cdot, \hat{\rho}', q' \rangle$ and $q' \neq 0$
$P = Q'(\theta_{\langle \langle \underline{n}, \eta \rangle, \hat{\eta}', Q' \rangle}(P_1))$ by def. In/PreVC and Definition 4.11
$\langle \cdot, \hat{\eta}', Q' \rangle \sim \langle \cdot, \hat{\rho}', q' \rangle \ (P, \phi')$ by Lemma 4.15
$\models_{\phi'} \theta_{\langle \langle \underline{n}, \eta \rangle, \hat{\eta}', Q' \rangle}(P_1)$ by Definition 4.5
$\models_{\phi_{\langle \langle \underline{n}, \mathcal{V}_\phi(\eta) \rangle, \hat{\rho}', q' \rangle}} P_1$ by Lemma 4.18 and Lemma 4.16
$\mathrm{Sat}_{\mathcal{P},\Phi}(\check{s}, p^{\mathsf{l}})$ by Definition 4.12
Case: $\Phi_i = \texttt{ret}$ is impossible for $k > 0$
Case: $\Phi_i = \texttt{inv}\ P_1, X, \mathcal{P} \vdash P_1\ \mathsf{spec}, p^{\mathsf{l}} = \langle \texttt{tol}\langle \texttt{some}\langle X, P_1, P_1 \rangle, i \dot{+} 1 \rangle, p^{\mathsf{c}} \rangle$
$P = Q(\theta_{\check{t}}(P_1) \wedge P_2)$ by def. PreVC/Eval/In/Gen and Definition 4.11
$\models_\phi \theta_{\check{t}}(P_1)$ by Definition 4.7 and def. $\models$
$\models_{\phi_{\check{s}}} P_1$ by Lemma 4.16
$\mathrm{Sat}_{\mathcal{P},\Phi}(\check{s}, p^{\mathsf{l}})$ by Definition 4.12
Case: $\mathcal{P} \vdash P_1\ \mathsf{spec}, p^{\mathsf{l}} = \langle \texttt{id}\langle \texttt{all}\langle P_1, P_1 \rangle \rangle, p^{\mathsf{c}}_1, \ldots, p^{\mathsf{c}}_{k'} \rangle$
similar to previous case

□

A link specification is available with respect to the value of an associated symbolic state:

Lemma 4.37 (Available Link Specification)
$\Psi; \mathcal{P}; \Phi \,\widetilde{\vartriangleright\!\!\vartriangleright}\, {<}\!\cdot \langle t, \hat{\eta}, Q \rangle *\!\!\succ \langle p^{\mathsf{l}}_1, \ldots, p^{\mathsf{l}}_k \rangle$ *or* $\Psi; \Phi \,\tilde{\vartriangleright}\, t *\!\!\succ \langle p^{\mathsf{l}}_1, \ldots, p^{\mathsf{l}}_k \rangle$

for $k > 0$ *and* $\vdash \mathcal{P}$ sp
implies $\mathrm{Avail}_{\Psi,\Phi}(\mathcal{V}_\phi(t), p^{\mathsf{l}})$ *for some* $p^{\mathsf{l}} \in \{p^{\mathsf{l}}_1, \ldots, p^{\mathsf{l}}_k\}$

PROOF:
Similar to proof of Lemma 4.35

$\square$

Simulation is preserved by evaluation for all executions of a given VC:

Lemma 4.38 (Trace Simulation)

1. $\Psi; \mathcal{P}; \Phi \mathrel{\widetilde{\triangleright\!\triangleright}} \check{\pi}$ *such that* $\vdash \mathcal{P}$ sp,
2. $\models \mathrm{VC}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C}_1, \check{\pi}_1 \rangle)$ *for all* $\langle \mathcal{C}_1, \check{\pi}_1 \rangle \in \mathcal{S}_1$ *such that* $\mathrm{Next}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C}, \check{\pi} \rangle) \subseteq \mathcal{S}_1$,
3. $\mathrm{Head}(\check{\pi}) \sim \check{s}\ (P, \phi)$ *for* $P = \mathrm{VC}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C}, \check{\pi} \rangle)$ *and* $\mathrm{Enab}_{\Psi,\mathcal{P},\Phi}(\mathcal{C}, \mathcal{S}_2, \phi)$ *and* $\mathrm{Cov}_{\Psi,\mathcal{P},\Phi}(\mathcal{S}_2, \mathcal{S}_1)$, *and*
4. $\Psi; \mathcal{P}; \Phi \mathrel{\triangleright\!\triangleright} \check{\sigma}\, \check{s}$ *and* $\Psi; \Phi \mathrel{\triangleright\!\triangleright} \lfloor \check{\sigma}\, \check{s} \rfloor\, \sigma$

implies

1. $\mathrm{Int}_{\Psi,\mathcal{P},\Phi}(\lfloor \check{\sigma}\, \check{s} \rfloor\, \sigma, \check{s}')$ *for some* $\check{s}'$, *and*
2. $\lfloor \check{s}' \rfloor \in \lfloor \check{\sigma}\, \check{s} \rfloor\, \sigma$ *implies*
   (a) $\mathrm{Head}(\check{\pi}') \sim \check{s}'\ (P', \phi')$ *for* $P' = \mathrm{VC}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C}', \check{\pi}' \rangle)$ *and some* ϕ' *and some* $\langle \mathcal{C}', \check{\pi}' \rangle \in \mathcal{S}_1$ *such that* $\Psi; \mathcal{P}; \Phi \mathrel{\widetilde{\triangleright\!\triangleright}} \check{\pi}'$, *and*
   (b) $\mathrm{Enab}_{\Psi,\mathcal{P},\Phi}(\mathcal{C}', \mathcal{S}'_2, \phi')$ *for some* $\mathcal{S}'_2$ *such that* $\mathrm{Cov}_{\Psi,\mathcal{P},\Phi}(\mathcal{S}'_2, \mathcal{S}_1)$

where $\mathrm{Cov}_{\Psi,\mathcal{P},\Phi}(\mathcal{S}_2, \mathcal{S}_1)$
iff either $\langle \mathcal{C}, \check{\pi} \rangle \in \mathcal{S}_1$ *or* $\mathrm{Next}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C}, \check{\pi} \rangle) \subseteq \mathcal{S}_1$ *for all* $\langle \mathcal{C}, \check{\pi} \rangle \in \mathcal{S}_2$

PROOF:
By induction on the length of σ
Well-foundedness comes from each non-start use of the IH being preceded by the evaluation of a link specification or context specification that makes a transition (the latter indirectly through an enabled context).

(sketch)
$\Psi; \mathcal{P}; \Phi \mathrel{\triangleright\!\triangleright} \check{\sigma}_1$ such that $\lfloor \check{\sigma}_1 \rfloor \leq \lfloor \check{\sigma}\, \check{s} \rfloor\, \sigma$ by Lemma 4.24 and Definition 4.9
Case: $\lfloor \check{\sigma}_1 \rfloor \neq \lfloor \check{\sigma}\, \check{s} \rfloor\, \sigma, \check{\sigma}_1 = \check{\sigma}_2\, \check{s}_1$
  $\mathrm{Tail}(\check{\pi}) \sim \check{s}_1\ (P, \phi_1)$ by Lemma 4.24
  case: $q_1 \neq 0, \check{\pi} = \check{\pi}_1\, \check{t}_1 \cdot\!>$
    $\Psi; \mathcal{P}; \Phi \mathrel{\triangleright\!\triangleright} \check{\sigma}_2\, \check{s}_1 \cdot\!>$ by applying ea_2 and ee_3 using Lemma 4.15 and def. VC
    $\mathrm{Int}_{\Psi,\mathcal{P},\Phi}(\lfloor \check{\sigma}\, \check{s} \rfloor\, \sigma, \check{s}')$ by Definition 4.9 (choosing $\check{s}'$ such that $\lfloor \check{s}' \rfloor \notin \lfloor \check{\sigma}\, \check{s} \rfloor\, \sigma$)
  case: $q_1 \neq 0, \check{\pi} = \check{\pi}_1\, \check{t}_1 \ast\!\succ \langle p^{\mathsf{l}}_1, \ldots, p^{\mathsf{l}}_k \rangle, i_1 \in \mathrm{dom}\, \mathcal{C}$
    $\Psi; \mathcal{P}; \Phi \mathrel{\triangleright\!\triangleright} \check{\sigma}'$ such that $\lfloor \check{\sigma}' \rfloor \leq \lfloor \check{\sigma}_2\, \check{s}_1 \rfloor\, \sigma_1$
      by Definition 4.9 using Definition 4.14, Lemma 4.26, and def. VC
    case: $\lfloor \check{\sigma}' \rfloor \neq \lfloor \check{\sigma}_2\, \check{s}_1 \rfloor\, \sigma_1, \check{\sigma}' = \check{\sigma}'_1\, \check{s}'$
      $\mathrm{Head}(\check{\pi}') \sim \check{s}'\ (P', \phi')$ by Definition 4.14

$\mathrm{Enab}_{\Psi,\mathcal{P},\Phi}(\mathcal{C}',\mathcal{S}_2',\phi')$ by Lemma 4.26 or Definition 4.14
apply IH if $\langle \mathcal{C}',\breve{\pi}'\rangle \notin \mathcal{S}_1$
case: $q_1 \neq 0, \breve{\pi} = \breve{\pi}_1\, \breve{t}_1 \ast\!\succ \langle p_1^{\mathsf{l}},\ldots,p_k^{\mathsf{l}}\rangle, i_1 \notin \mathrm{dom}\,\mathcal{C}, k > 0$
$\mathrm{Avail}_{\Psi,\Phi}(s_1,p^{\mathsf{l}})$ for $p^{\mathsf{l}} \in \{p_1^{\mathsf{l}},\ldots,p_k^{\mathsf{l}}\}$ by Lemma 4.37
let $\langle\langle \mathcal{C}_0',\breve{t}_0'\rangle,\ldots,\langle \mathcal{C}_{k'}',\breve{t}_{k'}'\rangle\rangle = \mathrm{Eval}_{\mathcal{P},\Phi}(\langle \mathcal{C},\breve{t}_1\rangle,p^{\mathsf{l}})$
for each $j \in \{0,\ldots,k'\}$
let $P_j' = \mathrm{VC}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C}_j', \prec\!\ast\, \breve{t}_j'\, \breve{\pi}_j'\rangle)$

$$\text{let } P_j = \begin{cases} (\mathrm{PreVC}_{\mathcal{P},\Phi}(\breve{t}_1,p^{\mathsf{l}}))_j & \text{if } (\mathrm{PreVC}_{\mathcal{P},\Phi}(\breve{t}_1,p^{\mathsf{l}}))_j \neq \texttt{open} \\ P_j' & \text{otherwise} \end{cases}$$

$\breve{t}_1 \sim \breve{s}_1\ (P_j,\phi_1)$ by Definition 4.7 and def. VC/Conj/Open/Closed/$\vDash$
$\mathrm{Sat}_{\mathcal{P},\Phi}(\breve{s}_1,p^{\mathsf{l}})$ by Lemma 4.36
$\Psi;\mathcal{P};\Phi \mathrel{\triangleright\!\triangleright} \breve{\sigma}'$ such that $\lfloor\breve{\sigma}'\rfloor \leq \lfloor\breve{\sigma}_2\, \breve{s}_1\rfloor\, \sigma_1$ by Definition 4.9 using Lemma 4.33
let $\mathcal{S}_2' = \mathcal{S}_2 \cup \{\langle \mathcal{C}_0', \prec\!\ast\, \breve{t}_0'\, \breve{\pi}_0'\rangle,\ldots,\langle \mathcal{C}_{k'}', \prec\!\ast\, \breve{t}_{k'}'\, \breve{\pi}_{k'}'\rangle\}$
case: $\lfloor\breve{\sigma}'\rfloor \neq \lfloor\breve{\sigma}_2\, \breve{s}_1\rfloor\, \sigma_1$
$\breve{t}_0' \sim \breve{s}'\ (P_0',\phi')$ by Lemma 4.33
for all $\breve{s}'',\breve{\sigma}'',\sigma''$ s. t. $\vDash_{\phi'\cup\phi_{\breve{s}''}} \mathcal{C}_0'(i'')$, $\Psi;\mathcal{P};\Phi \mathrel{\triangleright\!\triangleright} \breve{\sigma}''\, \breve{s}''$, and $\Psi;\Phi \mathrel{\triangleright\!\triangleright} \lfloor\breve{\sigma}''\, \breve{s}''\rfloor\, \sigma''$
$\mathrm{Int}_{\Psi,\mathcal{P},\Phi}(\lfloor\breve{\sigma}''\, \breve{s}''\rfloor\, \sigma'', \breve{s}''')$
by def. $\mathrm{Enab}(i'')$ or by Lemma 4.26 and Definition 4.14
$\mathrm{Enab}_{\Psi,\mathcal{P},\Phi}(\mathcal{C}',\mathcal{S}_2',\phi')$ by Definition 4.14
apply IH if $(\mathrm{PreVC}_{\mathcal{P},\Phi}(\breve{t}_1,p^{\mathsf{l}}))_0 = \texttt{open}$
case: $q_1 = 0$
apply Lemma 4.22 (choosing $\breve{s}'$ such that $\lfloor\breve{s}'\rfloor \notin \lfloor\breve{\sigma}\, \breve{s}\rfloor\, \sigma$)

□

We can exhibit an execution $\breve{\sigma}$ for any execution σ for which we have a symbolic execution of a closed trace schedule that simulates a state erasing to a state in σ:

Lemma 4.39 (Trace Completion)

1. $\Psi;\mathcal{P};\Phi \mathrel{\widetilde{\triangleright\!\triangleright}} \breve{\pi}$ *such that* $\vdash \mathcal{P}$ sp,
2. $\vDash \mathrm{VC}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C}_1,\breve{\pi}_1\rangle)$ *for all* $\langle \mathcal{C}_1,\breve{\pi}_1\rangle \in \mathcal{S}_1$ *such that* $\mathrm{Next}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C},\breve{\pi}\rangle) \subseteq \mathcal{S}_1$ *and* $\mathrm{Fix}_{\Psi,\mathcal{P},\Phi}(\emptyset,\mathcal{S}_1) = \mathcal{S}_1$,
3. $\mathrm{Head}(\breve{\pi}) \sim \breve{s}\ (P,\phi)$ *for* $P = \mathrm{VC}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C},\breve{\pi}\rangle)$ *and* $\mathrm{Enab}_{\Psi,\mathcal{P},\Phi}(\mathcal{C},\mathcal{S}_2,\phi)$ *and* $\mathrm{Cov}_{\Psi,\mathcal{P},\Phi}(\mathcal{S}_2,\mathcal{S}_1)$, *and*
4. $\Psi;\mathcal{P};\Phi \mathrel{\triangleright\!\triangleright} \breve{\sigma}\, \breve{s}$ *and* $\Psi;\Phi \mathrel{\triangleright\!\triangleright} \lfloor\breve{\sigma}\, \breve{s}\rfloor\, \sigma$

implies $\Psi;\mathcal{P};\Phi \mathrel{\triangleright\!\triangleright} \breve{\sigma}'$ *for some* $\breve{\sigma}'$ *such that* $\lfloor\breve{\sigma}'\rfloor = \lfloor\breve{\sigma}\, \breve{s}\rfloor\, \sigma$

PROOF:
By induction on the length of σ using Lemma 4.38 and Lemma 4.27
Well-foundedness comes from each use of the IH being preceded by an appeal to Lemma 4.38, which makes at least one transition unless σ is empty.

□

We can exhibit an execution $\breve{\sigma}$ for any execution σ, given a valid VC for the program of σ:

Lemma 4.40 (Trace Construction) $\models \mathrm{VC}_{\Psi,\mathcal{P},\Phi}$ *and* $\Psi; \Phi \rhd\!\!\rhd \sigma$ *implies* $\Psi; \mathcal{P}; \Phi \rhd\!\!\rhd \breve{\sigma}$ *for some* $\breve{\sigma}$ *such that* $\lfloor \breve{\sigma} \rfloor = \sigma$

Proof:
(sketch)
$\Psi; \mathcal{P}; \Phi \mathrel{\widetilde{\rhd\!\!\rhd}} <\!\cdot\breve{t}\, \breve{\pi}$ by def. VC
let $\mathrm{dom}\,\phi_0 = \mathrm{FV}(\eta_*) \cup \mathrm{FV}(\hat{\eta}_{\mathcal{P}})$
let $\phi_0(\eta_*(r)) = \rho(r)$ for $r \in \mathit{Reg} \setminus \{\mathtt{ra}\}$ and $\phi_0(\hat{\eta}_{\mathcal{P}}(\hat{r})) = \hat{\rho}_{\mathcal{P}}(\hat{r})$ for $\hat{r} \in \mathrm{dom}\,\Gamma_{\mathcal{P}}$
$\mathcal{V}_{\phi_0}(\eta) = \rho$ and $\mathcal{V}_{\phi_0}(\hat{\eta}_{\mathcal{P}}) = \hat{\rho}_{\mathcal{P}}$ by def. $\mathcal{V}$
$\mathrm{VC}_{\Psi,\mathcal{P},\Phi}(\langle \mathcal{C}, \breve{\pi}\, \breve{t}\cdot\!> \rangle) = (\mathrm{Abs}_{\mathcal{P}}(\eta, \hat{\eta}_{\mathcal{P}}))(P_1)$ by def. VC and rule $\mathtt{ea}_1^{\sim}$
$\models_{\phi_0} P_1$ by def. $\models$
$\breve{t} \sim \breve{s}\ (P, \phi)$ by Definition 4.7 and Lemma 4.15
let $\mathcal{S}_1 = \mathrm{Fix}_{\Psi,\mathcal{P},\Phi}(\emptyset, \{\langle \emptyset, <\!\cdot\langle\langle i_j, \eta_j \rangle, \hat{\eta}_j, Q_j \rangle\, \breve{\pi}_j \rangle \mid 1 \leq j \leq k\})$
$\Psi; \mathcal{P}; \Phi \rhd\!\!\rhd \breve{\sigma}$ and $\lfloor \breve{\sigma} \rfloor = \sigma$ by Lemma 4.39

□

4.8 Technical Properties

The proofs of these lemmas are relatively direct, so they are not reproduced here.

4.8.1 Environments

Lemma 4.41 (Subsumption) $\phi_1 \cup \phi_2 = \phi_2$ *if* $\mathrm{dom}\,\phi_1 \subseteq \mathrm{dom}\,\phi_2$

Lemma 4.42 (Limited Commutativity) $\phi_1 \cup \phi_2 = \phi_2 \cup \phi_1$ *if* $\mathrm{dom}\,\phi_1 \cap \mathrm{dom}\,\phi_2 = \emptyset$

Lemma 4.43 (Associativity) $(\phi_1 \cup \phi_2) \cup \phi_3 = \phi_1 \cup (\phi_2 \cup \phi_3)$

Lemma 4.44 (Hiding) $(\phi_1 \setminus \mathrm{dom}\,\phi_2) \cup \phi_2 = \phi_1 \cup \phi_2$

Lemma 4.45 (Singleton Concatenation) $\phi[x \mapsto v] = \phi \cup \phi_{\emptyset}[x \mapsto v]$

Lemma 4.46 (Left Valuation) $\mathcal{V}_{\phi_1 \cup \phi_2}(E) = \mathcal{V}_{\phi_1}(E)$ *if* $E \in \mathrm{dom}\,\mathcal{V}_{\phi_1}$ *and* $\mathrm{FV}(E) \cap \mathrm{dom}\,\phi_2 = \emptyset$

Lemma 4.47 (Left Validity) $\models_{\phi_1 \cup \phi_2} P$ *iff* $\models_{\phi_1} P$ *if* $\mathrm{FV}(P) \subseteq \mathrm{dom}\,\phi_1$ *and* $\mathrm{FV}(P) \cap \mathrm{dom}\,\phi_2 = \emptyset$

Lemma 4.48 (Right Valuation) $\mathcal{V}_{\phi_1 \cup \phi_2}(E) = \mathcal{V}_{\phi_2}(E)$ *if* $E \in \mathrm{dom}\,\mathcal{V}_{\phi_2}$

Lemma 4.49 (Right Validity) $\models_{\phi_1 \cup \phi_2} P$ *iff* $\models_{\phi_2} P$ *if* $\mathrm{FV}(P) \subseteq \mathrm{dom}\,\phi_2$

4.8.2 Substitutions

Lemma 4.50 (Basic Properties)

1. $\theta_1 \cup \theta_2 = \theta_2$ *if* $\mathrm{dom}\,\theta_1 \subseteq \mathrm{dom}\,\theta_2$,
2. $\theta_1 \cup \theta_2 = \theta_2 \cup \theta_1$ *if* $\mathrm{dom}\,\theta_1 \cap \mathrm{dom}\,\theta_2 = \emptyset$,
3. $(\theta_1 \cup \theta_2) \cup \theta_3 = \theta_1 \cup (\theta_2 \cup \theta_3)$,
4. $(\theta_1 \setminus \mathrm{dom}\,\theta_2) \cup \theta_2 = \theta_1 \cup \theta_2$, *and*
5. $\theta[x \mapsto E] = \theta \cup \theta_\emptyset[x \mapsto E]$

Lemma 4.51 (Extension Free Variables)
$\mathrm{FV}(\theta[x \mapsto E]) \subseteq \mathrm{FV}(\theta) \cup \mathrm{FV}(E)$

Lemma 4.52 (Application Free Variables, Expressions)
$\mathrm{FV}(\theta(E)) \subseteq (\mathrm{FV}(E) \setminus \mathrm{dom}\,\theta) \cup \mathrm{FV}(\theta)$

Lemma 4.53 (Application Free Variables, Propositions)
$\mathrm{FV}(\theta(P)) \subseteq (\mathrm{FV}(P) \setminus \mathrm{dom}\,\theta) \cup \mathrm{FV}(\theta)$

Lemma 4.54 (Valuation Domain) $\theta \in \mathrm{dom}\,\mathcal{V}_\phi$ *iff* $\mathrm{FV}(\theta) \subseteq \mathrm{dom}\,\phi$

Lemma 4.55 (Left Valuation) $\mathcal{V}_{\phi_1 \cup \phi_2}(\theta) = \mathcal{V}_{\phi_1}(\theta)$
if $\theta \in \mathrm{dom}\,\mathcal{V}_{\phi_1}$ *and* $\mathrm{FV}(\theta) \cap \mathrm{dom}\,\phi_2 = \emptyset$

Lemma 4.56 (Right Valuation) $\mathcal{V}_{\phi_1 \cup \phi_2}(\theta) = \mathcal{V}_{\phi_2}(\theta)$ *if* $\theta \in \mathrm{dom}\,\mathcal{V}_{\phi_2}$

Lemma 4.57 (Extension Valuation) $\mathcal{V}_\phi(\theta[x \mapsto E]) = \mathcal{V}_\phi(\theta)[x \mapsto \mathcal{V}_\phi(E)]$
if $\theta[x \mapsto E] \in \mathrm{dom}\,\mathcal{V}_\phi$

Lemma 4.58 (Concatenation Valuation) $\mathcal{V}_\phi(\theta_1 \cup \theta_2) = \mathcal{V}_\phi(\theta_1) \cup \mathcal{V}_\phi(\theta_2)$
if $\theta_1 \cup \theta_2 \in \mathrm{dom}\,\mathcal{V}_\phi$

Lemma 4.59 (Application Valuation) $\mathcal{V}_\phi(\theta(E)) = \mathcal{V}_{\phi \cup \mathcal{V}_\phi(\theta)}(E)$
if $\theta \in \mathrm{dom}\,\mathcal{V}_\phi$ *and* $E \in \mathrm{dom}(\phi \cup \mathcal{V}_\phi(\theta))$

Lemma 4.60 (Application Validity) $\vDash_\phi \theta(P)$ *iff* $\vDash_{\phi \cup \mathcal{V}_\phi(\theta)} P$,
if $\theta \in \mathrm{dom}\,\mathcal{V}_\phi$ *and* $\mathrm{FV}(P) \subseteq \mathrm{dom}(\phi \cup \mathcal{V}_\phi(\theta))$

4.8.3 Proposition Contexts

Lemma 4.61 (Identity) $Q(\bullet) = Q$

Lemma 4.62 (Associativity) $(Q_1(Q_2))(P) = Q_1(Q_2(P))$

Lemma 4.63 (Operand Inference) $P_1 = P_2$ *if* $Q(P_1) = Q(P_2)$

Lemma 4.64 (Application Bound Variables)
$\mathrm{BV}(Q_1(Q_2)) = \mathrm{BV}(Q_1) \cup \mathrm{BV}(Q_2)$

4.8.4 Other

Lemma 4.65 (Expression Free Variables)
$\mathrm{FV}(E) \subseteq \mathrm{dom}\,\Gamma$ *if* $\Gamma \vdash_{\Delta} E : \tau$

Lemma 4.66 (Proposition Free Variables)
$\mathrm{FV}(P) \subseteq \mathrm{dom}\,\Gamma$ *if* $\Gamma \vdash_{\Delta} P \ \mathsf{prop}$

Acknowledgements

We thank Frank Pfenning, Fred B. Schneider, and Lisa Price for their helpful comments on an earlier draft of this report.

References

[ALLW96] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and language-independent mobile programs. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 127–136, Philadelphia, PA, May 1996.

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[AS86] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. Technical Report TR86-727, Cornell University, Computer Science Department, January 1986.

[AS89] Bowen Alpern and Fred B. Schneider. Verifying temporal properties without temporal logic. *ACM Transactions on Programming Languages and Systems*, 11(1):147–167, January 1989.

[CM99] Karl Crary and Greg Morrisett. Type structure for low-level programming languages. In *Automata, Languages and Programming: 26th International Colloquium, Proceedings*, volume 1644 of *Lecture Notes in Computer Science*, Prague, Czech Republic, July 1999.

[CW00] Karl Crary and Stephnie Weirich. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 184–198, Boston, MA, January 2000.

[CWM99] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, TX, January 1999.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[DvH66] Jack B. Dennis and Earl C. van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966. Presented at an ACM Programming Language and Pragmatics Conference, August, 1965, San Dinas, CA.

[Eme90] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier Science Publishers, 1990.

[ES99] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 New Security Paradigms Workshop*, Caledon Hills, Ontario, Canada, September 1999.

[ES00] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *RSP: 21th IEEE Computer Society Symposium on Research in Security and Privacy*, 2000.

[ET99] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Research in Security and Privacy, pages 32–45, Oakland, CA, May 1999.

[FM98] Stephen N. Freund and John C. Mitchell. A type system for object initialization in the Java bytecode language. Technical Note CS-TN-98-62, Stanford University, Department of Computer Science, April 1998.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison Wesley, 1996.

[GMPS97] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[HCC+97] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in java. Technical Report TR97-1660, Cornell University, Computer Science Department, December 1997.

[Kin71] J. C. King. Proving programs to be correct. *IEEE Transactions on Computers*, 20(11):1331–1336, November 1971.

[Koz99] Dexter Kozen. Language-based security. Technical Report TR99-1751, Cornell University, Computer Science Department, June 1999.

[Lam71] B. W. Lampson. Protection. In *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems,*. Princeton University, March 1971. Reprinted in Operating Systems Review 8,1 January 74.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, 1999.

[MCG+99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, May 1999.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 129–142, Saint-Malo, France, October 1997.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, San Diego, CA, January 1998.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 1997.

[Nec98] George Ciprian Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, September 1998. Available as Technical Report CMU-CS-98-154.

[NL96] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *USENIX 2nd Symposium on OS Design and Implementation*, Seattle, Washington, October 1996.

[NL98a] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 333–344, Montreal, Canada, June 1998.

[NL98b] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[O'C99] Robert O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 70–78, San Antonio, TX, January 1999.

[Rus89] John Rushby. Kernels for safety? In T. Anderson, editor, *Safe and Secure Computing Systems*, chapter 13, pages 210–220. Blackwell Scientific Publications, 1989.

[SA99] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, January 1999.

[Sch87] Fred B. Schneider. Decomposing properties into safety and liveness. Technical Report TR87-874, Cornell University, Computer Science Department, October 1987.

[Sch99] Fred B. Schneider. Enforceable security policies. Technical Report TR99-1759, Cornell University, Computer Science Department, July 1999.

[SG98] Abraham Silberschatz and Peter Baer Galvin. *Operating System Concepts*. Addison Wesley, fifth edition, 1998.

[Tho90] Wolfgang Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 135–191. Elsevier Science Publishers, 1990.

[Wal99] David Walker. A type system for expressive security policies. In *FLOC '99 Workshop on Run-time Result Verification*, Trento, Italy, July 1999.

[WBDF97] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architecture for Java. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 116–128, Saint-Malo, France, October 1997.

[WCC+74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.

[WK95] Kevin G. Wika and John C. Knight. On the enforcement of software safety policies. In *Compass '95: 10th Annual Conference on Computer Assurance*, pages 83–94, Gaithersburg, Maryland, 1995.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, December 1993.

A Examples

In this appendix, we exhibit representative Palladium security properties.

A.1 Resource Bounds

A.1.1 Instruction Counting

The following security property is a basis counting instructions:

$$\texttt{reg nInstr} : \mathbf{nat}$$

$$\begin{array}{l} \texttt{eval} \ <\!\cdot\_ \Rightarrow \texttt{nInstr} := 0 \\ \texttt{eval} \ \_ \rightarrow\Rightarrow \texttt{nInstr} := \texttt{nInstr} + 1 \end{array}$$

There is a single property register, `nInstr`, which counts the number of instructions executed. The first rule specifies that `nInstr` is zero when an agent starts executing. The second rule specifies that `nInstr` is incremented for each instruction. `nInstr` is declared as a natural number to avoid machine-word overflow.

The cost of an execution is obtained from `nInstr`. Thus, the rule

$$\texttt{require} \ \_ \rightarrow\Rightarrow \texttt{nInstr} \leq n$$

requires that an agent execute at most $\underline{n}$ instructions.

This technique extends to other cumulative properties of executions (*e.g.*, memory accesses, system calls). Given a valid proof, such a security property can be enforced by PCC with no run-time overhead.

To confirm our intuition about the above security property $\mathcal{P}$, we first define the length of an execution as its transition count:

$$\begin{array}{ll} |<\!\cdot s| = 0 & |<\!\cdot \check{s}| = 0 \\ |\sigma \rightarrow s| = |\sigma| + 1 & |\check{\sigma} \rightarrow \check{s}| = |\check{\sigma}| + 1 \\ |\sigma\!\cdot\!>| = |\sigma| & |\check{\sigma}\!\cdot\!>| = |\check{\sigma}| \end{array}$$

It is easy to show that $|\check{\sigma}| = |\lfloor\check{\sigma}\rfloor|$.

Given the usual model for the natural numbers, we can now show that `nInstr` contains the length of the execution:

Lemma A.1 (Instruction Count) $\Psi;\mathcal{P};\Phi \rhd\!\rhd \check{\sigma}\,\check{s}$ *implies* $\phi_{\check{s}}(\texttt{nInstr}) = |\check{\sigma}\,\check{s}|$

Proof:
By induction on the derivation of $\Psi;\mathcal{P};\Phi \rhd\!\rhd \check{\sigma}\,\check{s}$

let $\langle s, \hat{\rho}, q\rangle = \check{s}$
Case: $\check{\sigma} = <\!\cdot$

$\Psi;\mathcal{P};\Phi \rhd <\!\cdot\check{s}$	Inv. ee_1
$\Phi; <\!\cdot s \rhd \langle\mathcal{P}, \hat{\rho}_{\mathcal{P}}, 1\rangle \xrightarrow{\mathsf{s*}} \langle\cdot, \hat{\rho}, q\rangle$	Inv. ea_1
$\hat{\rho} = \hat{\rho}_{\mathcal{P}}[\texttt{nInstr} \mapsto 0]$	Inv. $\mathsf{s*}, \mathsf{s}$

$\phi_{\check{s}}(\texttt{nInstr}) = (\phi_s \cup \hat{\rho})(\texttt{nInstr}) = 0 = |<\!\cdot\check{s}|$ Def. $\phi_{\check{s}}, \cup, |\check{\sigma}|$

Case: $\check{\sigma} = \check{\sigma}_0\ \check{s}_0 \rightarrow$

$\Psi; \mathcal{P}; \Phi \rhd\!\rhd \check{\sigma}_0\ \check{s}_0 \qquad \Psi; \mathcal{P}; \Phi \rhd \check{s}_0 \rightarrow \check{s}$ Inv. ee_2

$\phi_{\check{s}_0}(\texttt{nInstr}) = |\check{\sigma}_0\ \check{s}_0|$ I.H.

let $\langle s_0, \hat{\rho}_0, q_0\rangle = \check{s}_0$

$\Phi; s_0 \rightarrow s \rhd \langle \mathcal{P}, \hat{\rho}_0, q_0\rangle \xrightarrow{\mathsf{s}*} \langle \cdot, \hat{\rho}, q\rangle$ Inv. ea_2

$\hat{\rho} = \hat{\rho}_0[\texttt{nInstr} \mapsto \mathcal{V}_{\phi\cup\hat{\rho}_0}(\texttt{nInstr} + 1)]$ Inv. $\mathsf{s}*, \mathsf{s}$

$\phi_{\check{s}}(\texttt{nInstr}) = (\phi_s \cup \hat{\rho})(\texttt{nInstr}) = \mathcal{V}_{\phi\cup\hat{\rho}_0}(\texttt{nInstr} + 1)$ Def. $\phi_{\check{s}}, \cup$

$= \hat{\rho}_0(\texttt{nInstr}) + 1 = \phi_{\check{s}_0}(\texttt{nInstr}) + 1 = |\check{\sigma}_0\ \check{s}_0| + 1$ Def. $\mathcal{V}, \cup, \phi_{\check{s}}$

$= |\check{\sigma}_0\ \check{s}_0 \rightarrow \check{s}|$ Def. $|\check{\sigma}|$

□

From this lemma, it follows that no execution exceeds the instruction bound:

Theorem A.1 (Instruction Bound) $\sigma \in \mathcal{P}(\Sigma_{\Psi,\Phi})$ *implies* $|\sigma| \leq \underline{n}$

PROOF:

$\Psi; \mathcal{P}; \Phi \rhd\!\rhd \check{\sigma} \qquad \lfloor\check{\sigma}\rfloor = \sigma$ Def. $\mathcal{P}(\Sigma_{\Psi,\Phi})$

Case: $\check{\sigma} = \check{\sigma}_0\ \check{s}_0 \rightarrow \check{s}$

let $\langle s_0, \hat{\rho}_0, q_0\rangle = \check{s}_0, \langle s, \hat{\rho}, q\rangle = \check{s}$

$q = 1$ Theorem 2.2

$\Phi; s_0 \rightarrow s \rhd \langle \mathcal{P}, \hat{\rho}_0, q_0\rangle \xrightarrow{\mathsf{s}*} \langle \cdot, \hat{\rho}, q\rangle$ Inv. ea_2

$\models_{\phi\cup\hat{\rho}} \texttt{nInstr} \leq n$ Inv. $\mathsf{s}*, \mathsf{s}$

$\mathcal{V}_{\phi\cup\hat{\rho}}(\texttt{nInstr}) \leq \mathcal{V}_{\phi\cup\hat{\rho}}(n)$ Def. $\models$

$\hat{\rho}(\texttt{nInstr}) \leq \underline{n}$ Def. $\mathcal{V}, \cup$

$\phi_{\check{s}}(\texttt{nInstr}) \leq \underline{n}$ Def. $\phi_{\check{s}}$

$|\check{\sigma}| \leq \underline{n}$ Lemma A.1

Case: $\check{\sigma} = <\!\cdot\check{s}$

similar to previous case

Case: $\check{\sigma} = \check{\sigma}_0\ \check{s}_0 \rightarrow \check{s}\cdot\!>$

$|\check{\sigma}_0\ \check{s}_0 \rightarrow \check{s}| \leq \underline{n}$ See prev. case

$|\check{\sigma}| \leq \underline{n}$ Def. $|\check{\sigma}|$

Case: $\check{\sigma} = <\!\cdot\check{s}\cdot\!>$

similar to previous case

□

A.1.2 Mutex Usage

The following security property tracks which locks are held by an agent:

$$\begin{array}{l}\texttt{reg arg} : \mathbf{wd} \\ \texttt{reg locks} : \mathbf{mapw}\end{array}$$

$$\texttt{admit} \;\; <\cdot\_ \Rightarrow \forall \mathtt{n} : \mathbf{wd}.\mathbf{selw}(\mathtt{locks}, \mathtt{n}) = \mathbf{0w}$$

$$\begin{array}{l}\texttt{eval} \;\; \rightarrow \texttt{proc acquire} \Rightarrow \texttt{arg} := \mathtt{r}_0 \\ \texttt{require} \;\; \rightarrow \texttt{proc acquire} \Rightarrow \mathbf{selw}(\mathtt{locks}, \mathtt{arg}) = \mathbf{0w} \\ \texttt{eval proc acquire} \rightarrow\Rightarrow \texttt{locks} := \mathbf{updw}(\mathtt{locks}, \mathtt{arg}, \mathbf{1w})\end{array}$$

$$\begin{array}{l}\texttt{eval} \;\; \rightarrow \texttt{proc release} \Rightarrow \texttt{arg} := \mathtt{r}_0 \\ \texttt{require} \;\; \rightarrow \texttt{proc release} \Rightarrow \mathbf{selw}(\mathtt{locks}, \mathtt{arg}) = \mathbf{1w} \\ \texttt{eval proc release} \rightarrow\Rightarrow \texttt{locks} := \mathbf{updw}(\mathtt{locks}, \mathtt{arg}, \mathbf{0w})\end{array}$$

Locks are identified by machine words (*e.g.*, resource descriptors, pointers to data structures). A lock is in one of two states: *acquired* or *released*. The trusted procedure `acquire` acquires the lock in register $\mathtt{r}_0$, taking it from the released state to the acquired state. `release` is its inverse, taking a lock from acquired to released. We adopt a strict interface: attempts to acquire an acquired lock or release a released lock are invalid.

This security property defines two property registers: `locks` maps lock identifiers to one or zero, according to whether a lock is acquired or released, respectively. `arg` saves the argument lock so that it is available in procedure postconditions.

The first and third evaluation rules specify that `arg` saves $\mathtt{r}_0$ for trusted-procedure calls. The requirement rules formalize the preconditions of the trusted procedures: a lock to be acquired must be released, but a lock to be released must be acquired[9]. The second and fourth evaluation rules formalize the state changes that result from calls to the trusted procedures.

To require that a program eventually release all locks, use the rule

$$\texttt{require} \;\_\cdot\!>\Rightarrow \forall \mathtt{n} : \mathbf{wd}.\mathbf{selw}(\mathtt{locks}, \mathtt{n}) = \mathbf{0w}$$

This rule requires the `locks` register to be returned to its original state—note that this requirement may be difficult to prove due to aliasing. We might also want to specify that the agent not abort with unreleased locks.

A.2 Essential Safety

Necula [Nec98] specifies a generic safety policy for his abstract machine based on three kinds of constraints: instruction safety, system-call safety, and partial correctness.

- *Instruction safety* limits the use of dangerous instructions based on a machine-state predicate.

[9] We assume that a type-safety property checks for invalid locks.

- *System-call safety* constrains calls to the trusted system.
- *Partial correctness* specifies the input/output behavior of the agent.

In this section, we show how this safety policy is encoded in Palladium[10].

Instruction safety is defined in terms of the relations $\texttt{Safe}_{eop}$ (one for each *eop*), $\texttt{Safe}_{cop}$ (one for each *cop*), $\texttt{SafeRd}$, and $\texttt{SafeWr}$. An operator relation holds when it is safe to perform the corresponding operation; the memory relations hold when it is safe to read or write memory. Using these relations, we encode instruction safety as follows:

$$\begin{array}{ll}
\texttt{require}\ _ \leftarrow \texttt{n1}\ eop\ \texttt{n2} \rightarrow\!\Rightarrow \texttt{Safe}_{eop}(\texttt{n1},\texttt{n2}) & \text{for each } eop \\
\texttt{require}\ \texttt{cond}\ cop\ \texttt{n1}, _ \rightarrow\!\Rightarrow \texttt{Safe}_{cop}(\texttt{n1}) & \text{for each } cop \\
\texttt{require}\ _ \leftarrow M[\texttt{n1}] \rightarrow\!\Rightarrow \texttt{SafeRd}(\texttt{mem},\texttt{n1}) & \\
\texttt{require}\ M[\texttt{n1}] \leftarrow \texttt{n2} \rightarrow\!\Rightarrow \texttt{SafeWr}(\texttt{mem},\texttt{n1},\texttt{n2}) &
\end{array}$$

There is one rule for each *eop* and *cop*.

Each trusted system call i has a precondition Pre_i, a postcondition $Post_i$, and a set of callee-save registers CS_i. We encode system-call safety as follows:

$$\begin{array}{ll}
\texttt{require}\ \rightarrow \texttt{proc}\ \bar{i} \Rightarrow Pre_i & \text{for each } i \in \text{Skip}(\Psi) \\
\texttt{scs}\ \bar{i} \Rightarrow r & \text{for each } i \in \text{Skip}(\Psi) \text{ and } r \in CS_i \\
\texttt{new}\ \texttt{proc}\ \bar{i} \rightarrow\!\Rightarrow \hat{r} : \tau_{\hat{r}} & \text{for each } i \in \text{Skip}(\Psi) \text{ and } \hat{r} \notin CS_i \\
\texttt{admit}\ \texttt{proc}\ \bar{i} \rightarrow\!\Rightarrow Post_i & \text{for each } i \in \text{Skip}(\Psi)
\end{array}$$

where $\tau_{\hat{r}}$ is the type of property register $\hat{r}$.

Similarly, each agent entry point i has a precondition Pre_i, a postcondition $Post_i$, and a set of callee-save registers CS_i. We encode partial correctness as follows:

$$\begin{array}{ll}
\texttt{admit}\ <\!\cdot\texttt{proc}\ \bar{i} \Rightarrow Pre_i & \text{for each } i \in \text{Start}(\Psi) \\
\texttt{ucs}\ \bar{i} \Rightarrow r & \text{for each } i \in \text{Start}(\Psi) \text{ and } r \in CS_i \\
\texttt{require}\ \texttt{proc}\ \bar{i}\cdot\!>\Rightarrow \hat{r} = E_{\hat{r}} & \text{for each } i \in \text{Start}(\Psi) \text{ and } \hat{r} \in CS_i \\
\texttt{require}\ \texttt{proc}\ \bar{i}\cdot\!>\Rightarrow Post_i & \text{for each } i \in \text{Start}(\Psi)
\end{array}$$

where $E_{\hat{r}}$ is the initial value of property register $\hat{r}$.

A.3 Security Automata

We can encode the simplest kind of security automata [Sch99] directly in Palladium. Note that Palladium security properties are themselves encodings of more general security automata (see Section 1.6).

Let a finite deterministic security automaton[11] $\mathcal{A}$ be a triple

$$\langle Q, q_0, \delta \rangle$$

[10] Note that we do not encode control-flow safety, because it is built into our VC generator; additionally, we do not enforce a stack discipline.

[11] In general, security automata can be nondeterministic over a countable set of states.

where

$$
\begin{array}{ll}
Q \subset \mathbb{N} \setminus \{0\} & \text{is the finite set of states} \\
q_0 \in Q & \text{is the initial state} \\
\delta \in Q \times \mathit{State} \rightarrow Q \cup \{0\} & \text{is the transition function}
\end{array}
$$

The states of an automaton are identified with a finite subset of the natural numbers; zero is reserved for the "bad" state. If an execution would cause a transition to the bad state, then it is aborted at that point.

We assume that δ can be represented as follows:

$$
\begin{array}{l}
s_1^{\square} \Rightarrow q_1 \stackrel{P_1}{\rightarrow} q_1' \\
\vdots \\
s_k^{\square} \Rightarrow q_k \stackrel{P_k}{\rightarrow} q_k'
\end{array}
$$

Rule j specifies that the automaton changes from state q_j to state q_j' when entering a machine state matching pattern $s_j^{\square}$, but only if P_j holds. Exactly one rule must be applicable for any given automaton state/machine state pair. For example, the following rule specifies a transition from state 1 to state 2 if anything is loaded from address 20:

$$r \leftarrow M[r'] \Rightarrow 1 \stackrel{r'=20}{\rightarrow} 2$$

δ is thus defined by the following equation:

$$
\delta(q, s) = \begin{cases}
q_1' & \text{if } q = q_1 \text{ and } \Phi \triangleright s_1^{\square}; s \stackrel{\mathsf{ps}}{\rightarrow} \theta_1; \phi_1 \text{ and } \models_{\phi_1} \theta_1(P_1) \\
\vdots & \\
q_k' & \text{if } q = q_k \text{ and } \Phi \triangleright s_k^{\square}; s \stackrel{\mathsf{ps}}{\rightarrow} \theta_k; \phi_k \text{ and } \models_{\phi_k} \theta_k(P_k)
\end{cases}
$$

Given this representation of δ, we encode the security property $\mathcal{P}_{\mathcal{A}}$ as follows:

$$
\begin{array}{l}
\texttt{reg}\ \mathsf{q} : \mathbf{nat} \\
\texttt{reg}\ \mathsf{qp} : \mathbf{nat} \\
\texttt{eval}\ <\!\cdot\,\_ \Rightarrow \mathsf{qp} := q_0 \\
\texttt{eval}\ \rightarrow \_ \Rightarrow \mathsf{qp} := \mathsf{q} \\
\texttt{new}\ \stackrel{<\cdot}{\rightarrow} \_ \Rightarrow \mathsf{q} : \mathbf{nat} \\
\texttt{admit}\ \stackrel{<\cdot}{\rightarrow} s_1^{\square} \Rightarrow (\mathsf{qp} = q_1 \wedge P_1) \supset \mathsf{q} = q_1' \\
\vdots \\
\texttt{admit}\ \stackrel{<\cdot}{\rightarrow} s_k^{\square} \Rightarrow (\mathsf{qp} = q_k \wedge P_k) \supset \mathsf{q} = q_k' \\
\texttt{require}\ \stackrel{<\cdot}{\rightarrow} \_ \Rightarrow \mathsf{q} \neq 0
\end{array}
$$

A.4 Liveness Properties

Borrowing a technique from Alpern and Schneider [AS89], we can use our VC generator to enforce certain liveness properties. Note that this example may be of only theoretical interest, because in practice we can use a real-time property to approximate a liveness property. Informally, a *real-time property* asserts that a specific "good thing" will happen after a finite time bound: these properties are prefix closed and therefore safety properties.

To enforce a liveness property of the form "eventually P", we require the agent to supply a *variant function* from states to integers whose value is less than one only after the target condition is satisfied; the variant function must decrease across each procedure call and branch. The *target condition* is a predicate on extended states that holds when the last state of an execution satisfies the property. The agent encodes the variant function as an expression E with free variables among the registers; P is constructed similarly and holds once the target condition is satisfied.

To require that the variant decreases, we use the following security property:

$$
\begin{array}{lll}
\texttt{reg v} : \textbf{int} & & \\
\texttt{eval} \;\; <\cdot\_ \Rightarrow \texttt{v} := E & & \\
\texttt{require} \;\; \rightarrow \texttt{cond}\ cop\ \_,\_ \Rightarrow E < \texttt{v} & & \text{for each } cop \\
\texttt{require} \;\; \rightarrow \texttt{call}\ \_ \Rightarrow E < \texttt{v} & & \\
\texttt{eval} \;\; \rightarrow \texttt{cond}\ cop\ \_,\_ \Rightarrow \texttt{v} := E & & \text{for each } cop \\
\texttt{eval} \;\; \rightarrow \texttt{call}\ \_ \Rightarrow \texttt{v} := E & &
\end{array}
$$

There are two rules for each *cop*. We save the most recent variant in the property register `v` and check that it decreases by the time we reach the next branch.

The following rules ensure that the property is eventually satisfied:

$$
\begin{array}{ll}
\texttt{require} \;\; \rightarrow \texttt{cond}\ cop\ \_,\_ \Rightarrow E \leq 0 \supset P & \text{for each } cop \\
\texttt{require} \;\; \rightarrow \texttt{call}\ \_ \Rightarrow E \leq 0 \supset P & \\
\texttt{require} \;\; \_\cdot> \Rightarrow P & \\
\texttt{require} \;\; \rightarrow \texttt{proc abort} \Rightarrow P & \text{for each such procedure}
\end{array}
$$

`abort` is a trusted procedure that always aborts the agent; this procedure is invoked to signal that the agent is unable to continue: for example, if the agent is instrumented to check array bounds, `abort` is called when a check fails. To ensure that the agent satisfies the target condition, we prohibit it from calling any aborting procedure until the condition is satisfied. Note that this requirement makes automatic certification very difficult.

Note that this technique depends on specific properties of the enforcement mechanism: the usual dynamic enforcement mechanism would abort the agent at a zero variant and leave the target condition unsatisfied. Although the execution set of the *security property* is prefix closed and therefore a safety property, our *enforcement mechanism* only accepts agents whose actual executions are a strict subset of this set. We argue this as follows: prior to satisfying the target condition, infinite loops are disallowed by the variant function and aborts are disallowed by precondition; because the agent cannot loop forever and cannot abort, it must eventually terminate normally, and thus must satisfy the target condition. The set of normally terminating agent executions is not prefix closed, and is thus not a safety property.

Note also that we can extend this technique to more general classes of security properties if we can encode a given property as one or more Büchi automata [Tho90] in Palladium. Given a set of pairs of positive and negative automata (see Section 4.3 of Alpern and Schneider [AS89]), we assume that one or more property registers represent the state of each automaton. Let P_i^+ hold when positive automaton i is in an accepting state, let P_i^- hold when negative automaton i is in an accepting state, and let E_i^- be the agent-supplied variant for negative automaton i. We derive a partial security property for each i:

$$\texttt{reg } \texttt{v}_i : \mathbf{int}$$

$$\begin{array}{l}\texttt{eval } <\!\cdot\_ \Rightarrow \texttt{v}_i := E_i^- \\ \texttt{require } \rightarrow \_ \Rightarrow P_i^+ \vee (\texttt{v}_i \geq E_i^- \wedge (P_i^- \supset \texttt{v}_i > E_i^-)) \\ \texttt{eval } \rightarrow \_ \Rightarrow \texttt{v}_i := E_i^-\end{array}$$

$$\begin{array}{l}\texttt{require } \_\cdot\!> \Rightarrow P_i^- \supset P_i^+ \\ \texttt{require } \rightarrow \texttt{proc abort} \Rightarrow P_i^- \supset P_i^+\end{array}$$

We use a disjunction instead of the equivalent implication for clarity.

It is possible to extend Palladium to encode execution sets that are not prefix closed and thereby increase its expressive power. Such a language could encode some liveness properties directly, but we have not extended Palladium because it is not clear if such a language would be practical.

B Static Semantics

This appendix contains a simple type system for checking the consistency of Palladium security properties. The type system catches simple compositional errors and should be a front-end filter for enforcement mechanisms.

B.1 First-Order Logic

Figures 28 through 31 contain a standard type system for the first-order logic. Because the logic is a sublanguage of Palladium, this type system is a component of the Palladium type system.

A signature Δ assigns types to constants (including functions and relations); Figure 28 contains inference rules for well-formed signatures ($\vdash \Delta\ \mathsf{sig}$). A context Γ assigns types to variables (*e.g.*, the free variables of a term); Figure 29 contains rules for well-formed contexts ($\vdash_\Delta \Gamma\ \mathsf{cont}$).

We also define signatures for our machine model:

$$\begin{array}{ll}\Delta_{\mathbf{mapw}} & = \mathbf{selw} : \mathbf{mapw} \times \mathbf{wd} \rightarrow \mathbf{wd}, \mathbf{updw} : \mathbf{mapw} \times \mathbf{wd} \times \mathbf{wd} \rightarrow \mathbf{mapw} \\ \Delta_{Lit} & = \mathbf{0w} : \mathbf{wd}, \mathbf{1w} : \mathbf{wd}, \ldots \\ \Delta_{Eop} & = \mathbf{addw} : \mathbf{wd} \times \mathbf{wd} \rightarrow \mathbf{wd}, \mathbf{subw} : \mathbf{wd} \times \mathbf{wd} \rightarrow \mathbf{wd}, \ldots \\ \Delta_{Cop} & = \mathbf{eq0w} : 2^{\mathbf{wd}}, \mathbf{neq0w} : 2^{\mathbf{wd}}, \ldots\end{array}$$

as well as contexts for the machine registers:

$$\begin{array}{ll} \Gamma_{UReg} & = \mathtt{r}_0 : \mathbf{wd}, \ldots, \mathtt{r}_{UMax} : \mathbf{wd}, \mathtt{ra} : \mathbf{wd} \\ \Gamma_{Reg} & = \Gamma_{UReg}, \mathtt{mem} : \mathbf{mapw} \\ \Gamma_{XReg} & = \Gamma_{Reg}, \mathtt{pc} : \mathbf{wd} \end{array}$$

To ensure that each register environment ρ is well typed, we require the initial ρ and the skip relation of Ψ to be well typed in Γ_{Reg}.

In Figure 30, we assign types to expressions; the judgment $\Gamma \vdash_\Delta E : \tau$ is read "expression E has type τ in context Γ and signature Δ." Finally, Figure 31 contains rules for well-formed propositions ($\Gamma \vdash_\Delta P$ prop).

B.2 Patterns

Figures 32 through 34 contain rules for well-formed patterns. For example, the judgment $\Gamma \vdash A^\square$ pat asserts that the step pattern $A^\square$ is well-formed with respect to Γ. Γ assigns types to the pattern variables of $A^\square$.

B.3 Palladium

Figure 35 contains rules for well-formed security properties. The judgment $\Gamma \vdash_\Delta \mathcal{P}$ sp asserts that $\mathcal{P}$ is well formed with respect to Γ and Δ. Γ assigns types to the registers appearing free in component terms of $\mathcal{P}$.

To type check specifications, we need the types of property registers. The context $\Gamma_\mathcal{P}$ contains the register declarations of $\mathcal{P}$:

$$\begin{array}{llll} \Gamma_{\cdot} & = \cdot & \Gamma_{\mathtt{eval}\ A^\square \Rightarrow \hat{r}:=E;\mathcal{P}} & = \Gamma_\mathcal{P} \\ \Gamma_{\mathtt{reg}\ \hat{r}:\tau;\mathcal{P}} & = \Gamma_\mathcal{P}, \hat{r} : \tau & \Gamma_{\mathtt{new}\ A^\square \Rightarrow \hat{r}:\tau;\mathcal{P}} & = \Gamma_\mathcal{P} \\ \Gamma_{\mathtt{require}\ A^\square \Rightarrow P;\mathcal{P}} & = \Gamma_\mathcal{P} & \Gamma_{\mathtt{scs}\ n^\square \Rightarrow r;\mathcal{P}} & = \Gamma_\mathcal{P} \\ \Gamma_{\mathtt{admit}\ A^\square \Rightarrow P;\mathcal{P}} & = \Gamma_\mathcal{P} & \Gamma_{\mathtt{ucs}\ n^\square \Rightarrow r;\mathcal{P}} & = \Gamma_\mathcal{P} \end{array}$$

Thus, $\operatorname{dom}\Gamma_\mathcal{P}$ is the property registers of $\mathcal{P}$.

A security property is well formed if it is well formed in the machine signatures and context:

$$\frac{\Gamma_{XReg} \vdash_{\Delta_{\mathbf{mapw}}, \Delta_{Lit}, \Delta_{Eop}, \Delta_{Cop}} \mathcal{P}\ \mathsf{sp}}{\vdash \mathcal{P}\ \mathsf{sp}}$$

Programs are annotated with specifications for procedures and loops. A specification is well formed if it is well formed in the machine signatures and context, as well as the property-register context:

$$\frac{\Gamma_{XReg}, \Gamma_\mathcal{P} \vdash_{\Delta_{\mathbf{mapw}}, \Delta_{Lit}, \Delta_{Eop}, \Delta_{Cop}} P\ \mathsf{prop}}{\mathcal{P} \vdash P\ \mathsf{spec}}$$

$$\frac{}{\vdash \cdot\ \mathsf{sig}} \qquad \frac{\vdash \Delta\ \mathsf{sig} \quad c \notin \mathrm{dom}\,\Delta}{\vdash \Delta, c : \tau\ \mathsf{sig}}$$

$$\frac{\vdash \Delta\ \mathsf{sig} \quad f \notin \mathrm{dom}\,\Delta}{\vdash \Delta, f : \tau_1 \times \cdots \times \tau_k \to \tau\ \mathsf{sig}} \qquad \frac{\vdash \Delta\ \mathsf{sig} \quad R \notin \mathrm{dom}\,\Delta}{\vdash \Delta, R : 2^{\tau_1 \times \cdots \times \tau_k}\ \mathsf{sig}}$$

Figure 28: Well-Formed Signatures ($\vdash \Delta\ \mathsf{sig}$)

$$\frac{\vdash \Delta\ \mathsf{sig}}{\vdash_\Delta \cdot\ \mathsf{cont}} \qquad \frac{\vdash_\Delta \Gamma\ \mathsf{cont} \quad x \notin \mathrm{dom}\,\Gamma}{\vdash_\Delta \Gamma, x : \tau\ \mathsf{cont}}$$

Figure 29: Well-Formed Contexts ($\vdash_\Delta \Gamma\ \mathsf{cont}$)

$$\frac{\vdash_\Delta \Gamma\ \mathsf{cont} \quad c : \tau \in \Delta}{\Gamma \vdash_\Delta c : \tau} \qquad \frac{\vdash_\Delta \Gamma\ \mathsf{cont} \quad x : \tau \in \Gamma}{\Gamma \vdash_\Delta x : \tau}$$

$$\frac{\vdash_\Delta \Gamma\ \mathsf{cont} \quad f : \tau_1 \times \cdots \times \tau_k \to \tau \in \Delta \quad \Gamma \vdash_\Delta E_1 : \tau_1 \quad \ldots \quad \Gamma \vdash_\Delta E_k : \tau_k}{\Gamma \vdash_\Delta f(E_1, \ldots, E_k) : \tau}$$

Figure 30: Expression Types ($\Gamma \vdash_\Delta E : \tau$)

$$\frac{\vdash_\Delta \Gamma\ \mathsf{cont} \quad R : 2^{\tau_1 \times \cdots \times \tau_k} \in \Delta \quad \Gamma \vdash_\Delta E_1 : \tau_1 \quad \ldots \quad \Gamma \vdash_\Delta E_k : \tau_k}{\Gamma \vdash_\Delta R(E_1, \ldots, E_k)\ \mathsf{prop}}$$

$$\frac{\vdash_\Delta \Gamma\ \mathsf{cont}}{\Gamma \vdash_\Delta \top\ \mathsf{prop}} \qquad \frac{\Gamma \vdash_\Delta E_1 : \tau \quad \Gamma \vdash_\Delta E_2 : \tau}{\Gamma \vdash_\Delta E_1 = E_2\ \mathsf{prop}} \qquad \frac{\Gamma \vdash_\Delta E_1 : \tau \quad \Gamma \vdash_\Delta E_2 : \tau}{\Gamma \vdash_\Delta E_1 \neq E_2\ \mathsf{prop}}$$

$$\frac{\Gamma \vdash_\Delta P_1\ \mathsf{prop} \quad \Gamma \vdash_\Delta P_2\ \mathsf{prop}}{\Gamma \vdash_\Delta P_1 \wedge P_2\ \mathsf{prop}} \qquad \frac{\Gamma \vdash_\Delta P_1\ \mathsf{prop} \quad \Gamma \vdash_\Delta P_2\ \mathsf{prop}}{\Gamma \vdash_\Delta P_1 \supset P_2\ \mathsf{prop}} \qquad \frac{\Gamma, x : \tau \vdash_\Delta P_1\ \mathsf{prop}}{\Gamma \vdash_\Delta \forall x : \tau. P_1\ \mathsf{prop}}$$

Figure 31: Well-Formed Propositions ($\Gamma \vdash_\Delta P\ \mathsf{prop}$)

$$\frac{}{\cdot \vdash r\ \mathsf{pat}} \qquad \frac{}{\cdot \vdash n\ \mathsf{pat}} \qquad \frac{}{x : \mathbf{wd} \vdash x\ \mathsf{pat}} \qquad \frac{}{\cdot \vdash \_\ \mathsf{pat}}$$

Figure 32: Well-Formed Atomic Patterns ($\Gamma \vdash r^{\square}\ \mathsf{pat}$ and $\Gamma \vdash n^{\square}\ \mathsf{pat}$)

$$\begin{array}{|l|}\hline \dfrac{\vdash_{.}\ \Gamma_1, \Gamma_2\ \mathsf{cont} \quad \Gamma_1 \vdash r^{\square}\ \mathsf{pat} \quad \Gamma_2 \vdash n^{\square}\ \mathsf{pat}}{\Gamma_1, \Gamma_2 \vdash I^{\square}\ \mathsf{pat}} \\ \hline I^{\square} \\ \hline r^{\square} \leftarrow n^{\square} \\ \hline \mathtt{cond}\ cop\ r^{\square}, n^{\square} \\ \hline \end{array} \qquad \begin{array}{|l|}\hline \dfrac{\vdash_{.}\ \Gamma_1, \Gamma_2\ \mathsf{cont} \quad \Gamma_1 \vdash r_1^{\square}\ \mathsf{pat} \quad \Gamma_2 \vdash r_1^{\square}\ \mathsf{pat}}{\Gamma_1, \Gamma_2 \vdash I^{\square}\ \mathsf{pat}} \\ \hline I^{\square} \\ \hline r_1^{\square} \leftarrow M[r_2^{\square}] \\ \hline M[r_1^{\square}] \leftarrow r_2^{\square} \\ \hline \end{array}$$

$$\begin{array}{|l|}\hline \dfrac{\Gamma \vdash n^{\square}\ \mathsf{pat}}{\Gamma \vdash I^{\square}\ \mathsf{pat}} \\ \hline I^{\square} \\ \hline \mathtt{ra} \leftarrow \mathtt{pc}\ \mathbf{addw}\ n^{\square} \\ \hline \mathtt{call}\ n^{\square} \\ \hline \end{array} \qquad \frac{\vdash_{.}\ \Gamma_1, \Gamma_2, \Gamma_3\ \mathsf{cont} \quad \Gamma_1 \vdash r_1^{\square}\ \mathsf{pat} \quad \Gamma_2 \vdash r_2^{\square}\ \mathsf{pat} \quad \Gamma_3 \vdash r_3^{\square}\ \mathsf{pat}}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash r_1^{\square} \leftarrow r_2^{\square}\ eop\ r_3^{\square}\ \mathsf{pat}}$$

$$\frac{}{\cdot \vdash \mathtt{ret}\ \mathsf{pat}}$$

Figure 33: Well-Formed Instruction Patterns ($\Gamma \vdash I^{\square}\ \mathsf{pat}$)

$$\frac{\Gamma \vdash n^{\square}\ \mathsf{pat}}{\Gamma \vdash \mathtt{proc}\ n^{\square}\ \mathsf{pat}} \qquad \frac{\Gamma \vdash n^{\square}\ \mathsf{pat}}{\Gamma \vdash \&n^{\square}\ \mathsf{pat}} \qquad \begin{array}{|l|}\hline \dfrac{\Gamma \vdash s^{\square}\ \mathsf{pat}}{\Gamma \vdash A^{\square}\ \mathsf{pat}} \\ \hline A^{\square} \\ \hline <\!\cdot s^{\square} \\ \hline s^{\square}\cdot\!> \\ \hline \rightarrow s^{\square} \\ \hline s^{\square} \rightarrow \\ \hline \overset{<}{\rightarrow} s^{\square} \\ \hline s^{\square} \overset{>}{\rightarrow} \\ \hline \end{array}$$

Figure 34: Well-Formed State and Step Patterns ($\Gamma \vdash s^{\square}\ \mathsf{pat}$ and $\Gamma \vdash A^{\square}\ \mathsf{pat}$)

$$\frac{\vdash_\Delta \Gamma \ \mathsf{cont}}{\Gamma \vdash_\Delta \cdot \ \mathsf{sp}} \qquad \frac{\Gamma, \hat{r} : \tau \vdash_\Delta \mathcal{P} \ \mathsf{sp}}{\Gamma \vdash_\Delta \mathtt{reg}\ \hat{r} : \tau; \mathcal{P} \ \mathsf{sp}}$$

$$\frac{\Gamma_2 \vdash A^{\square} \ \mathsf{pat} \quad \Gamma_1, \Gamma_2 \vdash_\Delta P \ \mathsf{prop} \quad \Gamma_1 \vdash_\Delta \mathcal{P} \ \mathsf{sp}}{\Gamma_1 \vdash_\Delta \mathtt{require}\ A^{\square} \Rightarrow P; \mathcal{P} \ \mathsf{sp}} \qquad \frac{\Gamma_2 \vdash A^{\square} \ \mathsf{pat} \quad \Gamma_1, \Gamma_2 \vdash_\Delta P \ \mathsf{prop} \quad \Gamma_1 \vdash_\Delta \mathcal{P} \ \mathsf{sp}}{\Gamma_1 \vdash_\Delta \mathtt{admit}\ A^{\square} \Rightarrow P; \mathcal{P} \ \mathsf{sp}}$$

$$\frac{\Gamma_2 \vdash A^{\square} \ \mathsf{pat} \quad \hat{r} : \tau \in \Gamma_1 \quad \Gamma_1, \Gamma_2 \vdash_\Delta E : \tau \quad \Gamma_1 \vdash_\Delta \mathcal{P} \ \mathsf{sp}}{\Gamma_1 \vdash_\Delta \mathtt{eval}\ A^{\square} \Rightarrow \hat{r} := E; \mathcal{P} \ \mathsf{sp}} \qquad \frac{\Gamma_2 \vdash A^{\square} \ \mathsf{pat} \quad \hat{r} : \tau \in \Gamma_1 \quad \Gamma_1 \vdash_\Delta \mathcal{P} \ \mathsf{sp}}{\Gamma_1 \vdash_\Delta \mathtt{new}\ A^{\square} \Rightarrow \hat{r} : \tau; \mathcal{P} \ \mathsf{sp}}$$

$$\frac{\Gamma_2 \vdash n^{\square} \ \mathsf{pat} \quad \Gamma_1 \vdash_\Delta \mathcal{P} \ \mathsf{sp}}{\Gamma_1 \vdash_\Delta \mathtt{scs}\ n^{\square} \Rightarrow r; \mathcal{P} \ \mathsf{sp}} \qquad \frac{\Gamma_2 \vdash n^{\square} \ \mathsf{pat} \quad \Gamma_1 \vdash_\Delta \mathcal{P} \ \mathsf{sp}}{\Gamma_1 \vdash_\Delta \mathtt{ucs}\ n^{\square} \Rightarrow r; \mathcal{P} \ \mathsf{sp}}$$

Figure 35: Well-Formed Security Properties ($\Gamma \vdash_\Delta \mathcal{P}$ sp)

C Pattern Matching and Derived Forms

In this appendix, we define pattern matching and derived forms.

C.1 Semantics

Figures 36 through 39 contain inference rules for pattern matching. For example, the judgment $\triangleright I^{\square}; I \overset{\mathsf{pi}}{\to} \theta$ asserts that instruction I matches instruction pattern $I^{\square}$ with substitution θ binding the pattern variables of $I^{\square}$. The judgment $\Phi \triangleright s^{\square}; s \overset{\mathsf{ps}}{\to} \theta; \phi$ is similar, except that the environment ϕ binds the registers of s. In the step judgment $\Phi \triangleright A^{\square}; A \overset{\mathsf{pa}}{\to} \theta; \phi$, ϕ is selected according to whether $A^{\square}$ is an "entering" or "leaving" pattern. Judgments are labeled pn for atomic patterns, pi for instruction patterns, ps for state patterns, and pa for step patterns.

We define substitution operations that are analogous to environments:

$$
\begin{array}{lll}
\theta_1 = \theta_2 & \text{iff} & \operatorname{dom}\theta_1 = \operatorname{dom}\theta_2 \text{ and} \\
 & & \theta_1(x) = \theta_2(x) \text{ for any } x \in \operatorname{dom}\theta_1 \\[4pt]
\theta_1 \subseteq \theta_2 & \text{iff} & \operatorname{dom}\theta_1 \subseteq \operatorname{dom}\theta_2 \text{ and} \\
 & & \theta_1(x) = \theta_2(x) \text{ for any } x \in \operatorname{dom}\theta_1
\end{array}
$$

$$
\begin{array}{ll}
\operatorname{dom}(\theta[x \mapsto E]) & = \operatorname{dom}\theta \cup \{x\} \\
\operatorname{dom}(\theta_1 \cup \theta_2) & = \operatorname{dom}\theta_1 \cup \operatorname{dom}\theta_2 \\
\operatorname{dom}(\theta \setminus X) & = \operatorname{dom}\theta \setminus X \\
\operatorname{dom}\theta_\emptyset & = \emptyset
\end{array}
$$

$$
\begin{array}{ll}
(\theta[x \mapsto E])(y) & = \begin{cases} E & \text{if } y = x \\ \theta(y) & \text{otherwise} \end{cases} \\[8pt]
(\theta_1 \cup \theta_2)(x) & = \begin{cases} \theta_2(x) & \text{if } x \in \operatorname{dom}\theta_2 \\ \theta_1(x) & \text{otherwise} \end{cases} \\[8pt]
(\theta \setminus X)(x) & = \theta(x)
\end{array}
$$

The free variables of a substitution are the free variables of its expressions:

$$
\mathrm{FV}(\theta) = \bigcup_{x \in \operatorname{dom}\theta} \mathrm{FV}(\theta(x))
$$

We apply a substitution to an expression or a proposition by substituting expressions for (free) variables:

$$
\begin{array}{ll}
\theta(c) & = c \\[6pt]
\theta(x) & = \begin{cases} \theta(x) & \text{if } x \in \operatorname{dom}\theta \\ x & \text{otherwise} \end{cases} \\[8pt]
\theta(f(E_1, \ldots, E_k)) & = f(\theta(E_1), \ldots, \theta(E_k))
\end{array}
$$

$$
\begin{array}{ll}
\theta(\top) & \equiv \top \\
\theta(R(E_1, \ldots, E_k)) & \equiv R(\theta(E_1), \ldots, \theta(E_k)) \\
\theta(E_1 = E_2) & \equiv \theta(E_1) = \theta(E_2) \\
\theta(E_1 \neq E_2) & \equiv \theta(E_1) \neq \theta(E_2) \\
\theta(P_1 \wedge P_2) & \equiv \theta(P_1) \wedge \theta(P_2) \\
\theta(P_1 \supset P_2) & \equiv \theta(P_1) \supset \theta(P_2) \\
\theta(\forall x : \tau . P_1) & \equiv \forall y : \tau . (\theta[x \mapsto y])(P_1) \\
 & \quad \text{where } y \notin \mathrm{FV}(\theta) \cup \mathrm{FV}(P_1)
\end{array}
$$

C.2 Symbolic Evaluation

Figures 40 and 41 contain pattern-matching rules for symbolic states and steps. These rules are similar to the corresponding concrete rules.

C.3 Derived Forms

Two security-property rules are defined by rewriting. Figure 42 contains the definition of these derived forms. We assume that the skip relation of Ψ does not make reentrant calls to Φ; a more sophisticated mechanism is needed to handle reentrant calls. Note that the calling convention of the symbolic evaluator (see Section 3) implies that skips are not reentrant.

$$\frac{}{\rhd r; r \overset{\mathsf{pn}}{\to} \theta_\emptyset}\ \mathsf{pn}_1 \qquad \frac{}{\rhd x; r \overset{\mathsf{pn}}{\to} \theta_\emptyset[x \mapsto r]}\ \mathsf{pn}_2 \qquad \frac{}{\rhd \_; r \overset{\mathsf{pn}}{\to} \theta_\emptyset}\ \mathsf{pn}_3$$

$$\frac{}{\rhd n; n \overset{\mathsf{pn}}{\to} \theta_\emptyset}\ \mathsf{pn}_4 \qquad \frac{}{\rhd x; n \overset{\mathsf{pn}}{\to} \theta_\emptyset[x \mapsto n]}\ \mathsf{pn}_5 \qquad \frac{}{\rhd \_; n \overset{\mathsf{pn}}{\to} \theta_\emptyset}\ \mathsf{pn}_6$$

Figure 36: Atomic Pattern Matching ($\rhd n^\square; n \overset{\mathsf{pn}}{\to} \theta$ and $\rhd r^\square; r \overset{\mathsf{pn}}{\to} \theta$)

$$\frac{\rhd r^\square; r \overset{\mathsf{pn}}{\to} \theta_1 \quad \rhd n^\square; n \overset{\mathsf{pn}}{\to} \theta_2}{\rhd I^\square; I \overset{\mathsf{pi}}{\to} \theta_1 \cup \theta_2}\ \mathsf{pi}_2$$

$I^\square$	I
$r^\square \leftarrow n^\square$	$r \leftarrow n$
`cond` $cop\ r^\square, n^\square$	`cond` $cop\ r, n$

$$\frac{\rhd r_1^\square; r_1 \overset{\mathsf{pn}}{\to} \theta_1 \quad \rhd r_2^\square; r_2 \overset{\mathsf{pn}}{\to} \theta_2}{\rhd I^\square; I \overset{\mathsf{pi}}{\to} \theta_1 \cup \theta_2}\ \mathsf{pi}_1$$

$I^\square$	I
$r_1^\square \leftarrow M[r_2^\square]$	$r_1 \leftarrow M[r_2]$
$M[r_1^\square] \leftarrow r_2^\square$	$M[r_1] \leftarrow r_2$

$$\frac{\rhd n^\square; n \overset{\mathsf{pn}}{\to} \theta}{\rhd I^\square; I \overset{\mathsf{pi}}{\to} \theta}\ \mathsf{pi}_3$$

$I^\square$	I
`ra` $\leftarrow$ `pc` **addw** $n^\square$	`ra` $\leftarrow$ `pc` **addw** n
`call` $n^\square$	`call` n

$$\frac{\rhd r_1^\square; r_1 \overset{\mathsf{pn}}{\to} \theta_1 \quad \rhd r_2^\square; r_2 \overset{\mathsf{pn}}{\to} \theta_2 \quad \rhd r_3^\square; r_3 \overset{\mathsf{pn}}{\to} \theta_3}{\rhd r_1^\square \leftarrow r_2^\square\ eop\ r_3^\square; r_1 \leftarrow r_2\ eop\ r_3 \overset{\mathsf{pi}}{\to} \theta_1 \cup \theta_3 \cup \theta_3}\ \mathsf{pi}_4$$

$$\frac{}{\rhd \mathtt{ret}; \mathtt{ret} \overset{\mathsf{pi}}{\to} \theta_\emptyset}\ \mathsf{pi}_5$$

Figure 37: Instruction Pattern Matching ($\rhd I^\square; I \overset{\mathsf{pi}}{\to} \theta$)

$$\frac{\rhd n^\square; n \overset{\mathsf{pn}}{\to} \theta \quad i \in \{\underline{n}\} \cup \mathrm{Dom}_\Phi(\underline{n})}{\Phi \rhd \mathtt{proc}\ n^\square; \langle i, \rho\rangle \overset{\mathsf{ps}}{\to} \theta; \phi_{\langle i,\rho\rangle}}\ \mathsf{ps}_1 \qquad \frac{\rhd I^\square; \Phi_i \overset{\mathsf{pi}}{\to} \theta}{\Phi \rhd I^\square; \langle i, \rho\rangle \overset{\mathsf{ps}}{\to} \theta; \phi_{\langle i,\rho\rangle}}\ \mathsf{ps}_2$$

$$\frac{\rhd n^\square; n \overset{\mathsf{pn}}{\to} \theta}{\Phi \rhd \& n^\square; \langle \underline{n}, \rho\rangle \overset{\mathsf{ps}}{\to} \theta; \phi_{\langle \underline{n},\rho\rangle}}\ \mathsf{ps}_3 \qquad \frac{}{\Phi \rhd \_; s \overset{\mathsf{ps}}{\to} \theta_\emptyset; \phi_s}\ \mathsf{ps}_4$$

Figure 38: State Pattern Matching ($\Phi \rhd s^\square; s \overset{\mathsf{ps}}{\to} \theta; \phi$)

$$\frac{\Phi \triangleright s^{\Box}; s \overset{\mathsf{ps}}{\to} \theta; \phi}{\Phi \triangleright A^{\Box}; A \overset{\mathsf{pa}}{\to} \theta; \phi}\ \mathsf{pa}_1$$

$A^{\Box}$	A	Case
$<\!\cdot s^{\Box}$	$<\!\cdot s$	
$\to s^{\Box}$	$s' \to s$	$s^{\Box} \neq \texttt{proc} \ \ldots$
$s^{\Box} \to$	$s \to s'$	$s^{\Box} \neq \texttt{proc} \ \ldots$
$s^{\Box}\!\cdot\!>$	$s\!\cdot\!>$	

$$\frac{\Phi \triangleright A_1^{\Box}; A \overset{\mathsf{pa}}{\to} \theta; \phi}{\Phi \triangleright A_2^{\Box}; A \overset{\mathsf{pa}}{\to} \theta; \phi}\ \mathsf{pa}_3$$

$A_1^{\Box}$	$A_2^{\Box}$
$<\!\cdot s^{\Box}$	$\overset{<}{\to} s^{\Box}$
$\to s^{\Box}$	$\overset{<}{\to} s^{\Box}$
$s^{\Box}\!\cdot\!>$	$s^{\Box} \overset{>}{\to}$
$s^{\Box} \to$	$s^{\Box} \overset{>}{\to}$

$$\frac{\Phi \triangleright \texttt{proc}\ n^{\Box}; s \overset{\mathsf{ps}}{\to} \theta; \phi}{\Phi \triangleright A^{\Box}; A \overset{\mathsf{pa}}{\to} \theta; \phi}\ \mathsf{pa}_2$$

$A^{\Box}$	A	Φ_i	s	Case
$\to \texttt{proc}\ n^{\Box}$	$\langle i, \rho\rangle \to s'$	$\texttt{call}\ n$	$\langle \underline{n}, \rho\rangle$	
$\texttt{proc}\ n^{\Box} \to$	$\langle i, \rho\rangle \to \langle i', \rho'\rangle$	$\texttt{ret}$	$\langle i, \rho'\rangle$	
$\texttt{proc}\ n^{\Box} \to$	$\langle i, \rho\rangle \to \langle i', \rho'\rangle$	$\texttt{call}\ n$	$\langle \underline{n}, \rho'\rangle$	$i' \neq \underline{n}$

Figure 39: Step Pattern Matching ($\Phi \triangleright A^{\Box}; A \overset{\mathsf{pa}}{\to} \theta; \phi$)

$$\frac{\triangleright n^{\Box}; n \overset{\mathsf{pn}}{\to} \theta \quad i \in \{\underline{n}\} \cup \mathrm{Dom}_{\Phi}(\underline{n})}{\Phi \,\tilde{\triangleright}\, \texttt{proc}\ n^{\Box}; \langle i, \eta\rangle \overset{\mathsf{ps}}{\to} \theta; \theta_{\langle i, \eta\rangle}}\ \mathsf{ps}_1^{\sim} \qquad \frac{\triangleright I^{\Box}; \Phi_i \overset{\mathsf{pi}}{\to} \theta}{\Phi \,\tilde{\triangleright}\, I^{\Box}; \langle i, \eta\rangle \overset{\mathsf{ps}}{\to} \theta; \theta_{\langle i, \eta\rangle}}\ \mathsf{ps}_2^{\sim}$$

$$\frac{\triangleright n^{\Box}; n \overset{\mathsf{pn}}{\to} \theta}{\Phi \,\tilde{\triangleright}\, \&n^{\Box}; \langle \underline{n}, \eta\rangle \overset{\mathsf{ps}}{\to} \theta; \theta_{\langle \underline{n}, \eta\rangle}}\ \mathsf{ps}_3^{\sim} \qquad \frac{}{\Phi \,\tilde{\triangleright}\, \_; t \overset{\mathsf{ps}}{\to} \theta_{\emptyset}; \theta_t}\ \mathsf{ps}_4^{\sim}$$

Figure 40: Symbolic State Pattern Matching ($\Phi \,\tilde{\triangleright}\, s^{\Box}; t \overset{\mathsf{ps}}{\to} \theta_1; \theta_2$)

$$\frac{\Phi \,\tilde{\triangleright}\, s^{\square}; t \xrightarrow{\mathsf{ps}} \theta_1;\theta_2}{\Phi \,\tilde{\triangleright}\, A^{\square}; B \xrightarrow{\mathsf{pa}} \theta_1;\theta_2}\ \mathsf{pa}_1^{\sim}$$

$A^{\square}$	B	Case
$<\!\cdot s^{\square}$	$<\!\cdot t$	
$\rightarrow s^{\square}$	$t' \rightarrow t$	$s^{\square} \neq \texttt{proc} \ \ldots$
$s^{\square} \rightarrow$	$t \rightarrow t'$	$s^{\square} \neq \texttt{proc} \ \ldots$
$s^{\square}\cdot\!>$	$t\cdot\!>$	

$$\frac{\Phi \,\tilde{\triangleright}\, A_1^{\square}; B \xrightarrow{\mathsf{pa}} \theta_1;\theta_2}{\Phi \,\tilde{\triangleright}\, A_2^{\square}; B \xrightarrow{\mathsf{pa}} \theta_1;\theta_2}\ \mathsf{pa}_3^{\sim}$$

$A_1^{\square}$	$A_2^{\square}$
$<\!\cdot s^{\square}$	$\xrightarrow{<} s^{\square}$
$\rightarrow s^{\square}$	$\xrightarrow{<} s^{\square}$
$s^{\square}\cdot\!>$	$s^{\square} \xrightarrow{>}$
$s^{\square} \rightarrow$	$s^{\square} \xrightarrow{>}$

$$\frac{\Phi \,\tilde{\triangleright}\, \texttt{proc}\ n^{\square}; t \xrightarrow{\mathsf{ps}} \theta_1;\theta_2}{\Phi \,\tilde{\triangleright}\, A^{\square}; B \xrightarrow{\mathsf{pa}} \theta_1;\theta_2}\ \mathsf{pa}_2^{\sim}$$

$A^{\square}$	B	Φ_i	t	Case
$\rightarrow \texttt{proc}\ n^{\square}$	$\langle i,\eta\rangle \rightarrow t'$	$\texttt{call}\ n$	$\langle \underline{n},\eta\rangle$	
$\texttt{proc}\ n^{\square} \rightarrow$	$\langle i,\eta\rangle \rightarrow \langle i',\eta'\rangle$	$\texttt{ret}$	$\langle i,\eta'\rangle$	
$\texttt{proc}\ n^{\square} \rightarrow$	$\langle i,\eta\rangle \rightarrow \langle i',\eta'\rangle$	$\texttt{call}\ n$	$\langle \underline{n},\eta'\rangle$	$i' \neq \underline{n}$

Figure 41: Symbolic Step Pattern Matching ($\Phi \,\tilde{\triangleright}\, A^{\square}; B \xrightarrow{\mathsf{pa}} \theta_1;\theta_2$)

$$\begin{array}{lcll}
 & & \mathcal{P}_1; & \\
\mathcal{P}_1; & & \texttt{reg}\ \hat{r}_s : \tau; & \\
\texttt{scs}\ n^{\square} \Rightarrow r; & \Longrightarrow & \texttt{eval}\ \ \rightarrow \texttt{proc}\ n^{\square} \Rightarrow \hat{r}_s := r; & \text{where } r : \tau \in \Gamma_{Reg} \\
\mathcal{P}_2 & & \texttt{admit}\ \texttt{proc}\ n^{\square} \rightarrow\Rightarrow r = \hat{r}_s; & \text{and } \hat{r}_s \notin \operatorname{dom}\Gamma_{\mathcal{P}_1} \cup \operatorname{dom}\Gamma_{\mathcal{P}_2} \\
 & & \mathcal{P}_2 &
\end{array}$$

$$\begin{array}{lcll}
 & & \mathcal{P}_1; & \\
\mathcal{P}_1; & & \texttt{reg}\ \hat{r}_s : \tau; & \\
\texttt{ucs}\ n^{\square} \Rightarrow r; & \Longrightarrow & \texttt{eval}\ \ <\!\cdot\texttt{proc}\ n^{\square} \Rightarrow \hat{r}_s := r; & \text{where } r : \tau \in \Gamma_{Reg} \\
\mathcal{P}_2 & & \texttt{require}\ \texttt{proc}\ n^{\square}\cdot\!>\Rightarrow r = \hat{r}_s; & \text{and } \hat{r}_s \notin \operatorname{dom}\Gamma_{\mathcal{P}_1} \cup \operatorname{dom}\Gamma_{\mathcal{P}_2} \\
 & & \mathcal{P}_2 &
\end{array}$$

Figure 42: Security-Property Derived Forms

D Notation

D.1 First-Order Logic

Types	Type	is finite	τ	$\in \mathit{Type}$
Constants	Con	is finite	c	$\in \mathit{Con}$
	Fun	$\subseteq \mathit{Con}$	f	$\in \mathit{Fun}$
	Rel	$\subseteq \mathit{Con}$	R	$\in \mathit{Rel}$
Variables	Var	is countable	x, y	$\in \mathit{Var}$
Variable Sets	VarSet	$= \mathbf{2}^{\mathit{Var}}$	X	$\in \mathit{VarSet}$
Values	$\mathcal{U}$	$= \bigcup_\tau \mathcal{U}_\tau$	v	$\in \mathcal{U}$
Environments	Env	$= \mathit{Var} \rightharpoonup \mathcal{U}$	ϕ	$\in \mathit{Env}$

Signatures $\Delta ::= \cdot \mid \Delta, c : \tau \mid \Delta, f : \tau_1 \times \cdots \times \tau_k \to \tau \mid \Delta, R : 2^{\tau_1 \times \cdots \times \tau_k}$

Contexts $\Gamma ::= \cdot \mid \Gamma, x : \tau$

$$\mathit{Var} \cap \mathit{Con} = \emptyset$$

D.2 PAL

Literals	Lit	$\subseteq \mathit{Con}$	n	$\in \mathit{Lit}$
Operators	Eop	$\subseteq \mathit{Fun}$	eop	$\in \mathit{Eop}$
	Cop	$\subseteq \mathit{Rel}$	cop	$\in \mathit{Cop}$
Registers	UReg	$= \{\mathtt{r}_0, \ldots, \mathtt{r}_{\mathit{UMax}}, \mathtt{ra}\}$	$\dot{r}$	$\in \mathit{UReg}$
	Reg	$= \mathit{UReg} \cup \{\mathtt{mem}\}$	r	$\in \mathit{Reg}$
	XReg	$= \mathit{Reg} \cup \{\mathtt{pc}\} \subseteq \mathit{Var}$		
Values	$\mathcal{U}_{\mathbf{wd}}$	$= \{i \in \mathbb{N} \mid 0 \leq i < 2^{\mathit{WdBit}}\}$	i	$\in \mathcal{U}_{\mathbf{wd}}$
	$\mathcal{U}_{\mathbf{mapw}}$	$= \mathcal{U}_{\mathbf{wd}} \to \mathcal{U}_{\mathbf{wd}}$		
Register Env.	REnv	$= \mathit{Reg} \to \mathcal{U}$	ρ	$\in \mathit{REnv}$
States	State	$= \mathcal{U}_{\mathbf{wd}} \times \mathit{REnv}$	s	$\in \mathit{State}$

Procedures $F ::= \cdot \mid F, I$

Agent Programs $\Phi ::= \cdot \mid \Phi, \langle i_0, F \rangle$

Steps $A ::= <\!\cdot s \mid s \to s' \mid s \cdot\!>$

Executions $\sigma ::= \cdot \mid <\!\cdot \mid \sigma \to \mid \sigma \cdot\!> \mid \sigma s$

$$
\begin{array}{ll}
\mathbf{wd}, \mathbf{mapw} & \in \mathit{Type} \\
\mathbf{selw}, \mathbf{updw} & \in \mathit{Fun} \\
\mathbf{0w}, \mathbf{1w}, \ldots & \in \mathit{Lit} \\
\mathbf{addw}, \mathbf{subw} & \in \mathit{Eop} \\
\mathbf{eq0w}, \mathbf{neq0w} & \in \mathit{Cop}
\end{array}
$$

$$
\begin{array}{lll}
i \in \mathrm{Start}(\Psi) & \text{iff} & \mathrm{Agent}_\Psi(\langle i, \rho\rangle, s) \text{ for some } \rho, s \\
i \in \mathrm{Skip}(\Psi) & \text{iff} & \mathrm{Skip}_\Psi(\langle i, \rho\rangle, s') \text{ for some } \rho, s' \\
i \in \mathrm{Stop}(\Psi) & \text{iff} & \mathrm{Agent}_\Psi(s, \langle i, \rho\rangle) \text{ for some } s, \rho
\end{array}
$$

$$
\begin{array}{lll}
i_0 \in \mathrm{Proc}(\Phi) & \text{iff} & \langle i_0, F\rangle \in \Phi \text{ for some } F \\
i_0 + i \in \mathrm{Dom}(\Phi) & \text{iff} & \langle i_0, F\rangle \in \Phi \text{ and } i < |F| \text{ for some } F \\
i_0 + i \in \mathrm{Dom}_\Phi(i_0) & \text{iff} & \langle i_0, F\rangle \in \Phi \text{ and } i < |F| \text{ for some } F \\
\Phi_{i_0+i} = I & \text{iff} & \langle i_0, F\rangle \in \Phi \text{ and } F_i = I \text{ for some } F
\end{array}
$$

$$
\begin{array}{rl}
\mathrm{Dom}(\Phi) \cap \mathrm{Skip}(\Psi) & = \emptyset \\
\mathrm{Dom}(\Phi) \cap \mathrm{Stop}(\Psi) & = \emptyset \\
\mathrm{Skip}(\Psi) \cap \mathrm{Stop}(\Psi) & = \emptyset
\end{array}
$$

D.3 Palladium

Property Registers	$\mathit{PReg} \subseteq \mathit{Var}, \quad \mathit{PReg} \cap \mathit{Reg} = \emptyset$	$\hat{r} \in \mathit{PReg}$
Property-Reg. Env.	$\mathit{PEnv} = \mathit{PReg} \to \mathcal{U}$	$\hat{\rho} \in \mathit{PEnv}$
Sec.-Prop. States	$\mathit{PState} = \mathit{SecProp} \times \mathit{PEnv} \times \{0, 1\}$	$\hat{s} \in \mathit{PState}$
Extended States	$\mathit{EState} = \mathit{State} \times \mathit{PEnv} \times \{0, 1\}$	$\check{s} \in \mathit{EState}$
Substitutions	$\mathit{Sub} = \mathit{Var} \rightharpoonup \mathit{Exp}$	$\theta \in \mathit{Sub}$

Extended Steps	$\check{A} ::= <\!\cdot\check{s} \mid \check{s} \to \check{s}' \mid \check{s}\cdot\!>$
Extended Executions	$\check{\sigma} ::= \cdot \mid <\!\cdot \mid \check{\sigma} \to \mid \check{\sigma}\cdot\!> \mid \check{\sigma}\check{s}$

D.4 Symbolic Evaluator/VC Generator

Register Sub.	$RSub$	$= Reg \to Exp$	$\eta \in RSub$
Prop.-Reg. Sub.	$PRSub$	$= PReg \to Exp$	$\hat{\eta} \in PRSub$
Symbolic States	$SState$	$= \mathcal{U}_{\mathbf{wd}} \times RSub$	$t \in SState$
Sym. S.-P. States	$PSState$	$= SecProp \times PRSub \times CProp$	$\hat{t} \in PSState$
Sym. Ext. States	$ESState$	$= SState \times PRSub \times CProp$	$\check{t} \in ESState$
Trace Contexts	$TCont$	$= \mathcal{U}_{\mathbf{wd}} \rightharpoonup Prop$	$\mathcal{C} \in TCont$
Trace Schedules	$TSched$	$= \mathbf{2}^{TCont \times ESExec}$	$\mathcal{S} \in TSched$

Abstraction Spec.	$p^{\mathsf{a}} ::= \mathtt{all}\langle P, P' \rangle \mid \mathtt{some}\langle X, P, P' \rangle$	
Transition Spec.	$p^{\mathsf{t}} ::= \mathtt{id}\langle p^{\mathsf{a}} \rangle$ $\mid \mathtt{tol}\langle p^{\mathsf{a}}, i \rangle \mid \mathtt{tos}\langle i \rangle \mid \mathtt{tor}\langle i, p^{\mathsf{a}} \rangle$	
Context Specifications	$p^{\mathsf{c}} ::= \langle i, p^{\mathsf{t}} \rangle$	
Link Specifications	$p^{\mathsf{l}} ::= \langle p^{\mathsf{t}}, p^{\mathsf{c}}_1, \ldots, p^{\mathsf{c}}_k \rangle$	
Symbolic Steps	$B ::= <\!\cdot t \mid t \to t' \mid t \cdot\!>$ $\mid \prec\!\ast\, t \mid t \ast\!\succ \langle p^{\mathsf{l}}_1, \ldots, p^{\mathsf{l}}_k \rangle$	
Proposition Contexts	$Q ::= \bullet \mid P \wedge Q \mid P \supset Q \mid \forall x : \tau . Q$	$Q \in CProp$
Extended Sym. Steps	$\check{B} ::= <\!\cdot \check{t} \mid \check{t} \to \check{t}' \mid \check{t} \cdot\!>$ $\mid \prec\!\ast\, \check{t} \mid \check{t} \ast\!\succ \langle p^{\mathsf{l}}_1, \ldots, p^{\mathsf{l}}_k \rangle$	
Ext. Sym. Executions	$\check{\pi} ::= \cdot$ $\mid <\!\cdot \mid \check{\pi} \to \ \mid \check{\pi} \cdot\!>$ $\mid \prec\!\ast \ \mid \check{\pi} \ast\!\succ \langle p^{\mathsf{l}}_1, \ldots, p^{\mathsf{l}}_k \rangle$ $\mid \check{\pi}\, \check{t}$	$\check{\pi} \in ESExec$