Proof-Carrying Network Code

Christian Skalka
University of Vermont
Burlington, VT, USA
ceskalka@uvm.edu

John Ring
University of Vermont
Burlington, VT, USA
john.ring@uvm.edu

David Darais
University of Vermont
Burlington, VT, USA
ddarais@uvm.edu

Minseok Kwon
Rochester Institute of Technology
Rochester, NY, USA
jmk@cs.rit.edu

Sahil Gupta
Rochester Institute of Technology
Rochester, NY, USA
sg5414@rit.edu

Kyle Diller
Rochester Institute of Technology
Rochester, NY, USA
kid6584@rit.edu

Steffen Smolka
Cornell University
Ithaca, NY, USA
smolka@cs.cornell.edu

Nate Foster
Cornell University
Ithaca, NY, USA
jnfoster@cs.cornell.edu

ABSTRACT

Computer networks often serve as the first line of defense against malicious attacks. Although there are a growing number of tools for defining and enforcing security policies in software-defined networks (SDNs), most assume a single point of control and are unable to handle the challenges that arise in networks with multiple administrative domains. For example, consumers may want want to allow their home IoT networks to be configured by device vendors, which raises security and privacy concerns. In this paper we propose a framework called Proof-Carrying Network Code (PCNC) for specifying and enforcing security in SDNs with interacting administrative domains. Like Proof-Carrying Authorization (PCA), PCNC provides methods for managing authorization domains, and like Proof-Carrying Code (PCC), PCNC provides methods for enforcing behavioral properties of network programs. We develop theoretical foundations for PCNC and evaluate it in simulated and real network settings, including a case study that considers security in IoT networks for home health monitoring.

CCS CONCEPTS

• Security and privacy → Formal methods and theory of security; • Software and its engineering → Formal software verification; • Networks → Programming interfaces; Network security;

KEYWORDS

Trust Management; Formal Verification; Software-Defined Networks; Nexus Authorization Logic; NetKAT
modify the switch to apply a special emergency tag and relax the filtering rules used by the firewall to forward packets carrying this tag, even if they are also marked as private.

Although this is a simple example, it already raises a number of interesting issues. First, to correctly configure the network, we must specify, verify, and coordinate behavior on multiple devices—e.g., the tagging rules at the switch and the filtering rules at the firewall. Second, we must ensure that requests to reconfigure devices are submitted by an authorized party. For example, the homeowner might only trust device vendors that have been appropriately certified. Unfortunately, existing platforms do not provide adequate mechanisms for specifying and enforcing such properties.

Moreover, challenges related to security and federation arise in more complex and varied systems, such as the following scenarios.

**Campus Network:** The goal is to implement a distributed firewall that isolates different classes of traffic from each other—e.g., faculty, students, and visitors. However, different principals are responsible for managing the devices in the network. For instance, university staff might control the routers at the core of the network while department staff control the switches at the edge.

**Internet Exchange Point (IXP):** The goal is to allow each participant to specify policies that determine how their own traffic is handled. These policies encode intricate preferences and are often inter-dependent, due to complex business relationships and operational concerns—e.g., a large Internet Service Provider (ISP) might be willing to carry traffic generated by its direct customers, but not be willing to provide transist for competitors.

**Federated Edge Computing:** The goal is to push computational tasks to edge devices, which requires using the network to communicate the inputs and outputs of a given computation. For example, the IoT network (B) described above might coordinate with another IoT network (A) that provides a machine learning (ML) service at the edge, allowing A to configure B to forward public data to A as illustrated in Figure 1(c). However, authorization is challenging in this context since A and B may not have a direct trust relationship, and A and B may have independent local policies governing authorization and network behavior.

In each scenario, multiple principals must collaborate to manage different network devices and enforce the intended security policy. However, operators often have no choice but to rely on social mechanisms or even blind faith. This is unfortunate since networks can provide crucial security and privacy defenses. For example, networks can prevent sensitive information from being exfiltrated by monitoring and blocking unauthorized communication. They can ensure that data received from untrusted sources is properly sanitized before it is sent to internal servers. And they can provide strong guarantees about availability, even in the presence of congestion or failures, by setting up multiple, redundant paths connecting each pair of hosts.

A promising approach to these problems is to exploit the programming interface for network devices provided by software-defined networks (SDN). However existing SDN platforms either assume a single administrative domain, or only handle limited forms of federation—e.g., virtualization solutions that enable multiple tenants to control disjoint slices of the network. For instance, languages such as Frenetic [12], Pyretic [28], and NetKAT [4], and data plane verification tools such as Header Space Analysis [22] and Veriflow [23] assume that the network is managed by a single administrator who has global visibility of the network and full authority to control how packets are processed. SDN control platforms also present their own unique security challenges [25, 32].

**Figure 1: PCNC use case: health monitoring edge network.**

1.1 Overview and Foundations

To address these challenges, we introduce an expressive and flexible discipline for reconfiguring SDN controllers in a federated setting that supports both authorization and behavioral compliance of programs, called **Proof-Carrying Network Code (PCNC).** Analogous to **Proof-Carrying Code (PCC)** [29], PCNC allows clients to ship proofs of security compliance along with their code that can be verified before it is installed. We argue that networks are a good application domain for a PCC-style approach for several reasons. First, as discussed above, operators today must often execute programs produced by different parties with varying degrees of trust. Hence, a framework in which rich properties are automatically verified using a trustworthy proof checker could have a significant practical impact. Second, while networks are often large in size, the programs they execute tend to be extremely simple and thus amenable to verification—each device executes a loop-free program that classifies and transforms incoming packets.
We observe that there are two key concerns for allowing network reconfiguration across multiple administrative domains: authorizing that a network update is permitted, and verifying that the update preserves important behavioral properties. To this end, PCNC uses a client-server model in which administrators can submit policy specifications of authorization and behavior and clients can submit authorization credentials and network updates. Upon receiving a client request, the server verifies it against its current authorization and behavioral policies.

PCNC is based on two existing theories, which provide its formal foundation. Nexus Authorization Logic (NAL) [19, 35] for expressing and enforcing authorization policies, and NetKAT for expressing and enforcing behavioral policies. NetKAT is a domain-specific language for programming and reasoning about SDNs developed in previous work [4, 13]. It is based on a solid mathematical foundation, Kleene Algebra with Tests (KAT) [24], and comes equipped with an equational reasoning system that can be used to verify many properties of interest automatically [13].

### 2.1 Authorization Logic

In this section we develop a foundational theory for PCNC. To support authorization in PCNC, we adapt the authorization logic NAL [35]. We define a natural deduction style proof theory for the logic, for which we later develop a proof representation and checking method in Section 2.1.4.

To support behavioral policy specification and enforcement during network reconfiguration, we use the NetKAT language [4, 13]. NetKAT has a decidable equational theory—here, we define an algorithm for checking program equivalence that is incorporated into our PCNC framework to support behavioral verification.

#### 2.1.1 Monadic Interpretation of \textit{Says}

The work most related to our PCNC framework is the FLANC system [18]. However, being based on NAL and NetKAT, PCNC offers effective mechanisms for constructing proofs and deciding behavioral properties that are based on well-studied logical foundations. In particular, since PCNC requests carry System F_{\text{Says}} proof witnesses, authorization is based on proof checking, not proof reconstruction or certificate chain discovery, unlike systems such as SAFE [7].

In Section 4, we develop a case study that illustrates its features and applicability to SDN programming. This case study is based on the IoT health monitoring network example discussed above and illustrated in Figure 1. We propose a specific network topology, configurations, and PCNC requests that embody the example.

In Section 5, we describe an implementation of PCNC on a hardware testbed that combines novel verification components with NetKAT compilation tools from Frenetic [12], as described in Section 5.2. Our implementation includes a JSON wire format schema for PCNC messages and a signature verification scheme for NAL_{light} credentials. We evaluate the implementation using the case study and report on verification overhead.
a non-lax system we can only deduce X ⇒ A Says X if X is a theorem). While the lax interpretation of Says has been assumed by other systems, various authors have suggested that it can lead to dangerous consequences [1, 17, 19]. This includes the authors of a more recent version of Nexus Authorization Logic, called FOCAL, who refer to the lax interpretation of Says in NAL as a “bug” [19]. A thorough formal study of lax vs. non-lax interpretations identifies the core issue as the axiom of escalation [3] which results from extending classical logic with a lax interpretation of Says:

\[ \forall X, Y. (A \text{ Says } X) \Rightarrow (X \land A \text{ Says } Y) \]

However, since NAL is not classical—in particular it lacks negation and the law of excluded middle—it does not exhibit escalation. Furthermore, it enjoys the properties of Says transparency and handoff, which are desirable in any authorization logic. The former means that any principle can be trusted to assert their own worldview, whereas the latter supports delegation of authority in distributed settings. The use of NAL also has implementation benefits—lax logic can be interpreted monadically, and a monadic interpretation of Says readily supports a Curry-Howard isomorphism with typed monadic structures in a functional calculus (as we show in Section 2.1.4).1 Thus terms in the calculus can constitute proof witnesses, which enables optimization techniques (e.g., certain types of reduction).

Another important point is that in a proof checking system, beliefs relevant to a judgment are explicitly provided as a component of the judgment. And in the PCNC implementation (see Section 5.1), beliefs provided to support a proof are always cryptographically signed and thus ascribed to a principal (i.e. the signer). Therefore, problems with “importing beliefs” as a consequence of laxity noted by Hirsch et al. [19] are ameliorated in our setting. For these reasons, we focus on PCNC we build on NAL.

### 2.1.2 Syntax and Proof Theory of NAL\_light

In Figure 2 we present the syntax and proof theory of the logic NAL\_light in a judgmental style. The logic NAL\_light is a simplified fragment of NAL with a streamlined set of atomic principals A rather than the more complex principals used by Schneider et al. [35]. These principals are encoded using a subset of nullary atomic formulas (rather than introducing a new kind of term in the grammar)—an approach that simplifies our encoding of formulas as types. In particular, we do not need two forms of universal quantification (i.e., over first- and higher-order constructs) as in the original system [35].

Our proof theory also differs slightly from the original formulation but captures the same principles of deduction. Aside from the judgmental presentation, we define Unit and Bind rules which are known to be inter-derivable with the rule forms in the original definition (including idempotence, distribution, and necessitation). Also, we allow higher-order (vs. first-order) existential quantification. These design choices yield a tighter Curry-Howard correspondence with the system we present in Section 2.1.4. Note that negation and the law of excluded middle is not supported in NAL\_light, hence it is not classical and not subject to escalation [3].

We will use letters A, B, C to refer to principals, while F, G, H refer to formulas. Formulas include universal (∃) and existential (∃) quantification, and standard logical connectives (∧, ∨, and ⇒). Predicates f on terms t are left abstract and represent application-level assertions. In PCNC, we will be concerned with assertions about behavioral policies, so this logic allows verification of authorization and behavioral policy components to be synergized. We only require that variable X are allowed to appear in term position in predicates, thus supporting first-order quantification.

The logic includes a Says modality as a primitive. However the SpeakFor modality is defined as syntactic sugar in terms of higher-order quantification:

\[ A \text{ SpeakFor } B \triangleq \forall X. (A \text{ Says } X) \Rightarrow (B \text{ says } X) \]

A restricted form of delegation is also supported,

\[ A \text{ SpeakFor } B \text{ on } (X_1 \cdots X_n : F) \triangleq \forall X_1, \ldots, X_n. (A \text{ Says } F) \Rightarrow (B \text{ says } F) \]

where \( X_1 \cdots X_n \) only occur in term positions in \( F \)—i.e. they are first-order variables.

### 2.1.3 Complexity of Proof Inference and Verification

Because NAL\_light subsumes a higher-order constructive logic it is highly expressive. At the low end of expressiveness, intuitionist propositional logic is PSPACE-complete [38], while intuitionist predicate logic is undecidable—and these are first-order. Both can be embedded in NAL\_light, which is a higher-order constructive predicate logic, so in general NAL\_light proof inference is also undecidable. But checking NAL\_light proofs is linear in the size of the proof term, since each deduction step involves only simple syntactic checks.

Figure 2: NAL\_light syntax and proof derivation rules.
### Syntax

**Types**
- \( \tau \) ::= unit | \( \tau \times \tau \) | \( \tau \rightarrow \tau \) | \( \forall \tau \).\( \forall \) \( \exists X.\tau \)

**Terms**
- \( e \) ::= \( \text{var} \) | \( \text{proj}_1(e) \) | \( \text{proj}_2(e) \) | \( x \) | \( \lambda X.e \) | \( \text{let } x := e \text{ in } e \) | \( \text{let } x := e \text{ in } \Delta(X) \cdot \tau \cdot e \) | \( \Lambda X.e \cdot e[t] \cdot \langle \tau, e \rangle \)

### Typing Rules

#### Bind
\[
S, \Gamma \vdash \text{let } x := e \text{ in } e \quad : \quad S, \Gamma \vdash e : \forall X.\tau
\]

#### Type Apply
\[
S, \Gamma \vdash r \quad : \quad S, \Gamma \vdash e : X.\tau
\]

#### Ret
\[
S, \Gamma \vdash \text{ret}^h(e) : A \text{ Says } \tau
\]

### Figure 3: Selected System \( F_{\text{Says}} \) syntax and typing rules.

2.1.4 **Proof Representation: System \( F_{\text{Says}} \)**. Judgments are of the form \( \Delta \vdash F \), where \( \Delta \) is a list of assumptions, considered equivalent up to reordering. Validity of judgments is defined as derivability by inductive application of the derivation rules defined in Figure 2. If \( \Delta \) is empty (and hence \( F \) is a tautology) we write \( \vdash F \).

To represent proofs in a compact and verifiable manner, we introduce the language System \( F_{\text{Says}} \) which enjoys a Curry-Howard types-as-formulas correspondence with \( \text{NAL}_{\text{light}} \). This approach has been explored previously for a different authorization logic [2]. The syntax of System \( F_{\text{Says}} \) is presented in Figure 3, where \( x \) and \( X \) are type and term variables respectively. The language is an extension of System \( F \) with a Says monad to represent the Says modality in \( \text{NAL}_{\text{light}} \), and other features to represent \( \text{NAL}_{\text{light}} \) connectives. Types \( \tau \) of System \( F_{\text{Says}} \) have a tight correspondence with \( \text{NAL}_{\text{light}} \) formulas. The System \( F \) fragment of System \( F_{\text{Says}} \) is adequate to represent implication and higher order quantification as in \( \text{NAL}_{\text{light}} \), and the addition of sum (\( + \)) and product (\( \times \)) types are adequate to represent disjunction and conjunction. The Says monad is realized using return and bind terms. Existential quantification is realized using standard pack and unpack terms.

We introduce System \( F_{\text{Says}} \) here specifically for PCNC, so we only include three atomic predicate forms: reconfig, extend, and \( \prec \). The first two of these are parameterized by NetKAT programs and are unary, while the latter is binary. The predicate reconfig(\( p \)) asserts the intent to install configuration program \( p \), extend(\( p \)) asserts the intent to extend the current configuration with program \( p \), and \( p \prec q \) asserts semantic containment of \( p \) in \( q \).

Type validity is defined in terms of judgments are of the form \( S, \Gamma \vdash e : \tau \). Here, \( \Gamma \) is an environment binding free term variables to types, and \( S \) is the type set of variables in scope. Considering the type \( \tau \) as a NAL formula analogue, the term \( e \) is referred to as a \textit{witness}. The type derivation rules have a tight correspondence with the \( \text{NAL}_{\text{light}} \) natural deduction rules shown in Figure 2. These rules are mostly standard and include System \( F \)-style polymorphism, existential quantification, etc. As shown in Figure 3, to support judgments involving the Says modality, we include monadic typing rules for return and bind constructs.

\[
\text{Field} \quad f ::= f_1 | \cdots | f_k
\]

\[
\text{Packet} \quad pk ::= f_1 = v_1, \cdots, f_k = v_k
\]

\[
\text{History} \quad h ::= pk : () | pk : h
\]

**Predicate a, b, 1 = Id**

**Policy a, q ::= a**

**Filter**

\[
[ f = n ] \quad (pk : h) \vdash \{ [pk : h] \} \quad \text{if } pk : f = n
\]

\[
[ f = n ] \quad (pk : h) \vdash \{ [pk : h] \} \quad \text{otherwise}
\]

**Drop**

\[
\vdash a \quad \text{Not}
\]

**Modify**

\[
\vdash a + b \quad \text{Or}
\]

**Union**

\[
\vdash a : b \quad \text{And}
\]

**Sequence**

\[
\vdash p \quad \text{Iterate}
\]

**Duplicate**

\[
\vdash \text{dup} \quad \text{Duplicate}
\]

**Monad**

\[
\vdash h \quad : \quad [\text{dup}] \quad (pk : h) \vdash (pk : pk : h)
\]

**Field**

\[
f ::= f_1 | \cdots | f_k
\]

**Packet**

\[
pk ::= f_1 = v_1, \cdots, f_k = v_k
\]

**History**

\[
h ::= pk : () | pk : h
\]

**Predicate a, b, 1 = Id**

**Policy a, q ::= a**

**Filter**

\[
[ f = n ] \quad (pk : h) \vdash \{ [pk : h] \} \quad \text{if } pk : f = n
\]

\[
[ f = n ] \quad (pk : h) \vdash \{ [pk : h] \} \quad \text{otherwise}
\]

**Drop**

\[
\vdash a \quad \text{Not}
\]

**Modify**

\[
\vdash a + b \quad \text{Or}
\]

**Union**

\[
\vdash a : b \quad \text{And}
\]

**Sequence**

\[
\vdash p \quad \text{Iterate}
\]

**Duplicate**

\[
\vdash \text{dup} \quad \text{Duplicate}
\]

2.1.5 **Types-as-Formulas Correspondence**. Our main result for System \( F_{\text{Says}} \) is soundness of the representation—i.e., we show that if a System \( F_{\text{Says}} \) term is typeable at type \( \tau \), the formula corresponding to \( \tau \) is derivable. The issue of completeness is left as an interesting topic for future work—note that we do not support arbitrary atomic formulas in System \( F_{\text{Says}} \) nor an explicit false term.

Since the syntax of types and formulas used in System \( F_{\text{Says}} \) and \( \text{NAL}_{\text{light}} \) respectively do not match up, we define an interpretation of types as formulas denoted \( \langle \cdot \rangle \):

\[
\langle \tau_1 \prec \tau_2 \rangle = \langle \tau_1 \rangle \prec \langle \tau_2 \rangle
\]

\[
\langle \tau_1 + \tau_2 \rangle = \langle \tau_1 \rangle \vee \langle \tau_2 \rangle
\]

\[
\langle \tau_1 \rightarrow \tau_2 \rangle = \langle \tau_1 \rangle \Rightarrow \langle \tau_2 \rangle
\]

and so on. We extend this interpretation pointwise to type environments, which translate to lists of assumptions in \( \text{NAL}_{\text{light}} \). Then we can state the types-as-formulas correspondence isomorphism as follows, using this translation. The results follow by induction on derivations, and is straightforward due to the tight correspondence of derivation rules in System \( F_{\text{Says}} \) and \( \text{NAL}_{\text{light}} \).

**Theorem 2.1.** \( \text{If } S, \Gamma \vdash e : \tau \text{ is derivable then so is } (S, \Gamma) \vdash \langle e \rangle \).

**Example.** To illustrate System \( F_{\text{Says}} \) proof witnessing, consider the following formula that represents the important property of delegation handoff as discussed in [35]:

\[
\forall A.\forall B. (A \text{ Says } B \text{ SpeaksFor } A) \Rightarrow (B \text{ SpeaksFor } A)
\]

In System \( F_{\text{Says}} \), the type representation \( \tau \) of this formula is:

\[
\forall A.\forall B. (A \text{ Says }(X.\text{B Says } X \rightarrow A \text{ Says } X)) \rightarrow (X.\text{B Says } X \rightarrow A \text{ Says } X)
\]

and the following term \( e \) serves as a proof witness for this type \( \tau \), in the sense that \( \emptyset, \emptyset \vdash e : \tau \) is valid:

\[
\Lambda A.\Lambda B. \lambda x : (A \text{ Says }(X.\text{B Says } X \rightarrow A \text{ Says } X)) \cdot \\
\Lambda X.\Lambda z : B \text{ Says } X \cdot x \leftarrow x : (w[X])[z]
\]

2.2 **Network Programming**

The NetKAT language [4, 13] enables programmers to work in terms of functions on packets histories (where a packet is a record of fields and a history is a non-empty list of packets). This is a departure...
from low-level SDN languages such as OpenFlow, which require thinking about hardware-level details such as forwarding tables, matches, actions, priorities, etc. The language offers primitives for matching \((f = n)\) and modifying \((f \leftarrow n)\) packet headers, as well as combinators such as union \(\oplus\), sequence \(\cdot\), and Kleene star \(+\), that form larger programs out of smaller ones. NetKAT is based on a solid mathematical foundation, Kleene Algebra with Tests (KAT) [24], and comes equipped with an equational reasoning system that can be used to verify many properties of interest automatically [13].

Figure 4 defines the syntax and semantics of the language formally. The denotational semantics \(\llbracket \cdot \rrbracket\) models predicates \(a\) and policies \(p\) as functions that take a packet history as input and produce a set of packet histories as output. Most of the constructs in the language are standard, but the dup operator is worth noting: it extends the trajectory recorded in the packet history by one hop, which is useful for encoding paths. Many other constructs can be defined—e.g., it is straightforward to encode conditionals \(\text{if } a \text{ then } p \text{ else } q\) using union and sequence \((a \cdot p + \sim a \cdot q)\).

NetKAT has an equational deductive system that can be used to reason about network-wide properties automatically [4]. This consists of a collection of equational axioms of the form \(p \equiv q\) that capture equivalences between policies. These axioms are sound (every pair of policies that can be proved equivalent behave identically) and complete (every pair of policies that behave identically can be proved equivalent). Moreover, NetKAT has an efficient procedure for deciding policy equivalence [13], which enables automatic verification of rich properties such as reachability, loop freedom, traffic isolation, and many others [4]. In Section 5.3, we develop an extended example of NetKAT programming to realize the application scenario introduced in Section 1.

2.3 Decidable Behavioral Properties

Previous work has identified a number of interesting security properties that are decidable for NetKAT programs, including slicing, isolation, and waypointing policies [4]. A general property of interest for security is containment, where we write \(p \leq q\) if and only if \(q\) returns a subset of the packets returned by \(p\) on all inputs. In a security and privacy context, we can take \(q\) to be a behavioral specification, and by requiring that \(p \leq q\) for any configuration program \(p\), we enforce that \(p\) at most refines or specializes the behavior defined by \(q\). For the case study described in Section 1 this property is appropriate, as we will show in Section 4, and is interesting to adapt for our prototype implementation (though in principal other behavioral properties could also be verified in PCNC). Furthermore, the relation \(p \leq q\) can be considered an abbreviation for \(q \equiv p + q\), so decidability of \(\equiv\) allows specification and enforcement of desired network properties.

As mentioned in Section 2.1, the logic NAL_light admits application-level assertions, which is how PCNC behavioral policy integrates with NAL_light for uniform policy expression. In particular, we introduce the application-level assertion \(p \leq q\), with the following derivation rule in NAL_light, and its analogue for term witnessing in System F_says:

<table>
<thead>
<tr>
<th>Contains (q \equiv p + q)</th>
<th>Contains (q \equiv p + q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Delta + p \leq q)</td>
<td>(S, \Gamma + p \leq q : p \leq q)</td>
</tr>
</tbody>
</table>

2.3.1 Proving Program Equivalence. The technique for proving program equivalence we currently use in PCNC is based on bisimulation of deterministic NetKAT automata, which are obtained via determinization of the Antimorov derivative of NetKAT source programs [13, 37]. Analogous to standard automata for strings, NetKAT automata accept sequences of packets which are isomorphic to packet histories as we have defined for NetKAT. Hence, the language recognized by an automaton is equivalent to the semantics of the source program from which it is derived.

In the following, \(\mathcal{A}\) ranges over deterministic NetKAT automata, \(G(\mathcal{A})\) denotes the language accepted by \(\mathcal{A}\), and \(\sim\) denotes bisimilarity. Formally, if \(\mathcal{A}\) is the automaton computed for a NetKAT program \(p\) using the Antimorov derivative, then \(G(\mathcal{A})\) is related to \([p]\) by a variant of Kleene’s Theorem (see Foster et al. [13] for details). We note that \(\mathcal{A}_1 \sim \mathcal{A}_2\) implies \(G(\mathcal{A}_1) = G(\mathcal{A}_2)\). An automaton can be represented by continuation and observation maps \(\delta\) and \(\epsilon\). Following the formulation due to Smolka et al. [37] for deterministic automata (without loss of generality), both the observation and continuation functions are parameterized by configurations, which are pairs \((pk, l)\) for packets \(pk \in PK\) and states \(l \in S\). Intuitively, these configurations determine the relevant state \(l\), with properties of \(pk\) refining transitions (analogous to automata on guarded strings). The observation map applied to a configuration yields a function that accepts packets, whereas the continuation map applied to a configuration yields a function that transitions to the next state given a packet. Observation and continuation maps thus have the following signatures:

\[
\begin{align*}
\delta &: (S \times PK) \rightarrow PK \rightarrow S \\
\epsilon &: (S \times PK) \rightarrow PK \rightarrow 2
\end{align*}
\]

Given automata \(\mathcal{A}_1\) and \(\mathcal{A}_2\), we can prove their equivalence using bisimulation. Although bisimulation has been informally described in prior work [13, 37], here we give an explicit definition of the algorithm. We assume without loss of generality that automata are deterministic (since prior work has demonstrated sound and complete determination methods [13]). Let \(\mathcal{A}_1\) and \(\mathcal{A}_2\) be encoded as previously defined, and let \(\delta_1, \delta_2,\) and \(e_1, e_2\) be the continuation and observation maps of \(\mathcal{A}_1\) and \(\mathcal{A}_2\) respectively. We can check bisimulation denoted \(\mathcal{A}_1 \sim \mathcal{A}_2\) as follows, based on properties described in [13]. The bisimulation algorithm \(\text{equiv}\) can be defined as follows, using an accumulator \(\pi_c\) as a termination condition for previously explored configurations:

\[
\text{equiv}(c_1, c_2, \pi_c) \iff \\
\begin{align*}
&\forall c_1, c_2 \in \pi_c \lor \\
&\forall pk : (e_1 c_1 pk) = (e_2 c_2 pk) \land \\
&\forall pk : \text{equiv}(\delta_1 c_1 pk, pk), (\delta_2 c_2 pk, pk), \pi_c \cup \{(c_1, c_2)\})
\end{align*}
\]

Then, positing that initial automata states are always identified as 0, we can implement bisimulation as:

\[
\mathcal{A}_1 \sim \mathcal{A}_2 \iff \forall pk. \text{equiv}(0, pk), (0, pk), 0)
\]

Note these definitions quantify over all packets \(pk\). As packets contain all OpenFlow fields, there is clearly potential for combinatorial explosion in a naive implementation. However, a simple optimization is to only consider packets containing fields and values mentioned in the source programs or automata being checked for equivalence. In general, our method for implementing bisimulation
in the PCNC framework depends on our representation of source code and automata, including compiler optimizations. We return to this issue in Section 3.2.

3 THE PCNC FRAMEWORK

We implement PCNC using a client-server architecture. We assume that at least one principal Root has administrative authority on the Server, and that an initial behavioral policy has been defined as a NetKAT program spec. Then to install a configuration program p, the high level goal on the Server is to prove a judgment of the following form, where a credential environment Γ and a System F\textsubscript{says} proof witness e has been provided by the client:

\[ \varnothing, \Gamma \vdash (e, p \prec spec) : \text{Root Says reconfig}(p) \times (p \prec spec) \]

In this judgment, the program p is the (re)configuration program to be installed and spec is the NetKAT behavioral policy specification. The assertion \( p \prec \text{spec} \) is predicated on program equivalence as discussed in Section 2.3, and this component of the conjunction in the judgment covers behavioral policy enforcement. The other component of the conjunction Root \text{Says} reconfig(p) expresses the need to verify that the administrator approves installation, as deduced from the proof witness e provided by the client and the credentials provided in Γ.

We can also imagine as a weaker privilege that rather than a complete reconfiguration of the network, we allow only an extension of the current configuration. That is, assuming that \( q \prec \text{spec} \) is the current network configuration and p is the submitted extension, the network is reconfigured with \( p + q \) following verification. For this purpose we posit a different predicate extend, where the goal judgment to prove on the server is:

\[ \varnothing, \Gamma \vdash (e, p + q \prec \text{spec}) : \text{Root Says extend}(p) \times (p + q \prec \text{spec}) \]

Again in this case, the client would provide Γ and e, in addition to the extension program p, for verification of the request.

Thus, verification of the above judgments on the server, with configuration and proof material provided by the client, comprises both authorization and behavioral verification. Although in this paper we consider just containment policies, we note that the synergy of behavioral and authorization assertions in System F\textsubscript{says} would allow us to constrain principals to only affecting certain kinds of traffic, as in the “flow spaces” of FLANC [18]. This could be accomplished with appropriate application-level (NetKAT) assertions. A credentialized approach with public-key signatures allows this to scale to highly distributed settings.

3.1 Authorization in System F\textsubscript{says}

On the server side, validity of the submitted judgment is established by checking the validity of \( \Gamma \vdash e : \text{Root Says reconfig}(p) \). Since this entails type checking, as long as the latter is implemented correctly, we can trust typability of the given judgment—modulo trust of assumptions in Γ.

Authentication and Integrity of Assumptions. The user will typically submit a non-empty Γ containing credentials (as illustrated below in Section 4.1). The PCNC wire representation of credentials ensures that clients cannot forge false assumptions. Specifically, we restrict Γ so that it can only include credentials of the form A\text{Says} \tau in its image. Our approach to this is standard—we represent principals A as public keys \( K_A \), and for every assumption A\text{Says} \tau in the image of Γ, we include signature s = \text{sig}(K_A^{-1}, \tau) in the PCNC message, which is \tau signed with A’s private key. The server can then verify s given \( K_A \) and \( \tau \) which are provided directly in the credential (so no public key lookup is necessary). This establishes authenticity and integrity for all belief ascriptions in Γ. We provide more detail about algorithms and wire format used to represent proofs in Section 5.1.

As a simple example, if we assume that Bob is the local network administrator, and wants to submit his own reconfiguration program p, then Bob would submit the following credential:

\[ \text{cred} : K_{\text{Bob}} \text{Says reconfig}(p) \]

along with its signature:

\[ \text{sig}(K_{\text{Bob}}^{-1}, K_{\text{Bob}} \text{Says reconfig}(p)) \]

and the proof term \text{cred}. Then given these items, on the Server we can verify:

\[ \varnothing, \text{cred} : K_{\text{Bob}} \text{Says reconfig}(p) \vdash \text{cred} : K_{\text{Bob}} \text{Says reconfig}(p) \]

Note also in this example how verification of the supplied signature verifies authenticity and integrity of the reconfiguration program p, due to the signature of \text{cred}.

Crucially, we further observe that there exists no e such that

\[ \varnothing, \varnothing \vdash e : K_{\text{Bob}} \text{Says reconfig}(p) \]

is valid, since reconfig(p) has no direct term witness and must follow from assumptions. Thus any installation request must be appropriately credentialled. See Section 4.1 for a more extended example with a non-trivial proof term.

3.2 Behavioral Verification in NetKAT

If p is a reconfiguration program and spec is a behavioral policy defined as a NetKAT program, then installation of p requires verification of \( p \prec \text{spec} \) as a behavioral verification component. As discussed in Section 2.3, this is equivalent to proving \( \text{spec} \equiv p + \text{spec} \) by definition, and equivalence of NetKAT programs can be automatically proven via bisimulation of their derived deterministic automata. Thus, writing \( A(p) \) to denote the deterministic automata obtained from any program p, we have:

\[ p \prec \text{spec} \iff (A(\text{spec}) \sim A(p + \text{spec})) \]

Therefore the central technical challenge for PCNC behavioral verification is checking NetKAT program equivalence via bisimulation of derived automata. This approach has the benefit of integrating easily with the current state-of-the-art compiler for NetKAT in Frenetic [12, 37], where determinized NetKAT automata are generated as an intermediate representation. These automata are subsequently provided to a back-end that translates them to OpenFlow tables, but we can “intercept” automata representations, and in our framework, we define bisimulation directly on the intermediate automata representations.

3.2.1 FDDs and Optimizations. An important detail of automata representation in Frenetic is the use of forwarding decision diagrams (FDDs) [37] to represent observation and continuation functions as they are described in Section 2.3. FDDs are a variation of binary decision diagrams (BDDs), but test field names rather than bits. FDDs benefit from optimizing transformations which in turn
support optimizations of network programs as they are deployed in flow tables. Since equality and evaluation of FDDs is decidable, our approach is thus flexible with respect to FDD-based compiler optimizations. Furthermore, to achieve the bisimulation optimization suggested at the end of Section 2.3, in our implementation FDDs are analyzed to extract field-value pairs that are explicitly tested in programs. These pairs are used to generate the strict subset of packets that are relevant to checking bisimulation.

3.2.2 **Program Wire Format.** In the framework it is necessary to maintain the specification spec on the Server for confidence in its definition. Thus, when checking \( p \prec \text{spec} \) on the Server it is necessary to obtain \( A(p + \text{spec}) \) there. While it would be possible to ship \( A(p) \), and then compute \( A(p + \text{spec}) \), the current version of Frenetic does not support a serialized format for automata, and also compilation to automata is highly efficient for the examples we have considered. Shipping the source code \( p \) has the additional advantage of simplicity for this presentation, so in our prototype implementation we take this approach.

However, we observe that in principle it would also be feasible to ship \( A(p) \), which would have the benefit of being adaptable to arbitrary compiler optimizations that clients may desire to apply, and which may not be available on the Server.

4 **PCNC INSTANCE: CASE STUDY**

We now return to the case study introduced in Section 1, and show how it can be implemented as an instance of the PCNC framework. We imagine that the local network is owned and administered by Bob, and that the health monitoring devices Bob adds to his network are provided by the vendor NetCo, which also provides configuration code for these devices submitted as a PCNC request.

We further imagine that another IoT network, owned and administrated by Alice, provides an edge computing service, where Alice and Bob do not have a direct trust relationship. In order to obtain his public data, Alice submits a configuration extension code to Bob as a PCNC request.

4.1 **Authorization**

Because Bob purchases health monitoring devices from NetCo, he also trusts them to install configuration code in his local network, as well as to extend the local configuration. Thus Bob provides them with a credential asserting that NetCo speaks on Bob’s behalf for any installation or extension requests. In NAL_{light} this could be represented as the following formula, with the convention introduced in Section 3.1 that principals are represented by their public keys in credentials:

\[
\begin{align*}
K_{Bob} \text{Says } K_{NetCo} \text{SpeaksFor } K_{Bob} & \text{ on } (X : \text{extend}(X)) \\
& \land \\
& K_{NetCo} \text{SpeaksFor } K_{Bob} \text{ on } (X : \text{reconfig}(X))
\end{align*}
\]

and the corresponding System \( F_{\text{Says}} \) type form is:

\[
\tau_{\text{delegate}} \triangleq K_{Bob} \text{Says} \\
(V X. K_{NetCo} \text{Says reconfig}(X) \rightarrow K_{Bob} \text{Says reconfig}(X)) \times \\
(V X. K_{NetCo} \text{Says extend}(X) \rightarrow K_{Bob} \text{Says extend}(X))
\]

and in addition Bob provides NetCo the appropriate signature to certify the credential:

\[
\text{sig}(K_{Bob}^{-1} \cdot \tau_{\text{delegate}})
\]

Now, when NetCo submits a network configuration update \( p \) to Bob’s network, it is necessary for NetCo to provide credentials \( \Gamma \) and a term witness \( e \) such that \( \emptyset, \Gamma \vdash e : K_{Bob} \text{ Says reconfig}(p) \) is derivable. To accomplish this, NetCo would also generate a credential asserting their intent to install \( p \) (identical in NAL_{light} and System \( F_{\text{Says}} \) forms):

\[
\tau_{\text{reconfig}} \triangleq K_{NetCo} \text{Says reconfig}(p)
\]

along with the signature:

\[
\text{sig}(K_{NetCo}^{-1} \cdot \tau_{\text{reconfig}})
\]

Then to install \( p \), NetCo submits \( \Gamma \) containing the following type bindings:

\[
delegate : \tau_{\text{delegate}} \quad \text{reconfig} : \tau_{\text{reconfig}}
\]

along with the certifying signatures described above, and also submits the following term witness:

\[
witness_1 \downarrow \text{config} \leftarrow \text{delegate};(((\text{proj}(\text{config})|p)) \text{reconfig})
\]

On Bob’s Server, the following authorization judgment can then be verified:

\[
\emptyset, \Gamma \vdash \text{witness}_1 : K_{Bob} \text{ Says reconfig}(p)
\]

4.1.1 **Addressing Federation.** In the federated edge computing scenario where Alice aims to submit a PCNC request to forward Bob’s public data to her, the scenario is complicated by the realistic assumption that Bob and Alice do not have a direct trust relationship. To resolve this, an appealing approach is to assume that members of a network federation will allow each other to extend (not entirely reconfigure) each other’s networks. Membership in this group can be established by agreement on a certification authority (CA), that possesses the private key for the FedMem group. In our scenario this could be NetCo or another trusted entity. Certificates can then be constructed using the public key for FedMem.

Bob can advertise his trust in the FedMem group using the following credential:

\[
K_{Bob} \text{ Says } K_{FedMem} \text{SpeaksFor } K_{Bob} \text{ on } (X : \text{extend}(X))
\]

with the corresponding type form:

\[
\tau_{FedMem} \triangleq K_{Bob} \text{ Says } V X. K_{FedMem} \text{Says extend}(X) \rightarrow K_{Bob} \text{ Says extend}(X)
\]

along with the signature:

\[
\text{sig}(K_{Bob}^{-1} \cdot \tau_{FedMem})
\]

Upon joining the federation, Alice can independently certify her membership via the following certificate, issued by the CA:

\[
K_{FedMem} \text{ Says } K_{Alice} \text{ SpeaksFor } K_{FedMem}
\]

with the corresponding type form:

\[
\tau_{Alice} \triangleq K_{FedMem} \text{ Says } V X. K_{Alice} \text{ Says } X \rightarrow K_{FedMem} \text{ Says } X
\]

along with the signature:

\[
\text{sig}(K_{FedMem}^{-1} \cdot \tau_{Alice})
\]
Now, when Alice submits a network configuration extension $p$ to Bob’s network that forwards his public data to her, it is necessary for Alice to provide credentials $\Gamma$ and a term witness $e$ such that $\emptyset, \Gamma \vdash e : K_B \text{ Says extend}(p)$ is derivable. To accomplish this, Alice would also generate a credential asserting her intent to extend Bob’s current configuration $q$ with $p$ (identical in NALlight and System F$_{\text{Sys}}$ forms):

$$\tau_{\text{extend}} \triangleq K_A \text{ Says extend}(p)$$

along with the signature:

$$\text{sig}(K_A^{-1}, \tau_{\text{extend}})$$

Then Alice submits $\Gamma$ containing the following type bindings:

$$\text{fedmem} : \tau_{\text{fedmem}} \quad \text{alice} : \tau_{\text{Alice}} \quad \text{extend} : \tau_{\text{extend}}$$

along with the certifying signatures described above, and also submits the following term witness:

$$\text{witness}_2 \triangleq$$

$$\text{bobconfig} \leftarrow \text{fedmem}; \quad \text{bobconfig}[p](\text{config} \leftarrow \text{alice}, (\text{config}[\text{extend}(p)] \text{ extend}))$$

On Bob’s Server, the following authorization judgment can then be verified:

$$\emptyset, \Gamma \vdash \text{witness}_2 : K_B \text{ Says extend}(p)$$

### 4.2 Behavioral Verification

The network configuration shown in Figure 5 reifies the network structure for the case study introduced in Section 1. The local network contains two hosts and four switches/routers. Hosts $H1$ and $H2$ represent IoT devices (like the sleep tracker and the blood pressure monitor in the use case). There are also two local network switches (at layer 2) $SH1$ for $H1$ and $SH2$ for $H2$—it is also possible to have the switching capabilities implemented on IoT devices themselves. When we consider a concrete implementation of the case study in Section 5, we will simulate this exact structure in Mininet as described in Section 5.3.

**Behavioral Policy.** We imagine that the local network owner Bob specifies a behavioral policy that any network configuration code must satisfy. The main concern is that this policy only allows public or emergency data from escaping the network. Configuration code submitted by vendors must be a refinement of this policy.

The policy includes specification of the network topology $t$, a local forwarding policy $f$ for the topology, and crucially a firewall policy $w$ that indicates what sort of traffic is allowed to escape the network. Beginning with the topology, the local switches are linked to the edge router denoted by $ER$, which is then connected to the gateway switch $GS$ for external communication. In NetKAT, it is standard to use the following notation to represent a link from switch $S1$, port $P1$, to switch $S2$, port $P2$:

$$\text{dup} \cdot \text{sw} = S1 \cdot pt = P1 \cdot \text{sw} \leftarrow S2 \cdot pt \leftarrow P2 \cdot \text{dup}$$

By convention we will write such a link as follows:

$$[S1 : P1] \rightarrow [S2 : P2]$$

We can encode the network topology into NetKAT as follows:

$$f \triangleq (\text{sw} = S1 \cdot pt \leftarrow 2) + (\text{sw} = SH2 \cdot pt \leftarrow 2) + (\text{sw} = ER \cdot pt \leftarrow 1)$$

The firewall policy specifies that only data marked with the $\text{PUB}$ flag, indicating public data, or with the 911 flag, indicating emergency data, may be forwarded by the edge router:

$$w \triangleq \neg(\text{sw} = \text{ER}) + \text{sw} = \text{ER} \cdot \text{meta} = \text{PUB} + \text{sw} = \text{ER} \cdot \text{meta} = 911$$

The complete policy is then defined as follows, which is submitted by Bob to the PCNC server:

$$\text{spec} \triangleq (f \cdot w \cdot t)$$

#### 4.2.1 Configuration and Reconfiguration. Also depicted in Figure 5 are two external hosts $H3$ and $H4$. Initially, when Bob’s home network only includes the sleep tracker, we imagine that $H3$ is the external data repository used by the vendor for data storage and analysis. Thus, the initial network configuration uses the same topology and forwarding policies $t$ and $f$ as spec, but refines the firewall to only allow the release of public data destined for $H3$:

$$w_{\text{init}} \triangleq \neg(\text{sw} = \text{ER}) + \text{sw} = \text{ER} \cdot \text{dst} = H3 \cdot \text{meta} = \text{PUB}$$

and so the initial configuration is $(f \cdot w_{\text{init}} \cdot t)$+, and observe it is the case that $(f \cdot w_{\text{init}} \cdot t)$+ $\ll$ spec. Subsequently, when Bob’s network is augmented to include the blood pressure monitor, NetCo provides a reconfiguration that allows public and emergency data to be
released, while identifying a specific destination $H_4$ for emergency data (e.g., a care provider).

$$w_{\text{reconfig}} \triangleq \neg(sw = ER) +$$
$$sw = ER \cdot dst = H_3 \cdot meta = \text{PUB} +$$
$$sw = ER \cdot dst = H_4 \cdot meta = 911$$

In full detail, the reconfiguration program, that we call $\text{reconfig}$, would be defined as:

$$\text{reconfig} \triangleq (f \cdot w_{\text{reconfig}} \cdot t)^*$$

and it is the case that $\text{reconfig} \preceq \text{spec}$, since reconfiguration code only refines $\text{spec}$ with specific data endpoints.

Putting together these definitions in the PCNC framework, successful verification for the described reconfiguration request establishes validity of the following judgment:

$$\emptyset, \Gamma \vdash \langle \text{witness}_2, \text{reconfig}, \text{spec} \rangle :$$
$$K_{\text{Bob}} \text{Says } \text{reconfig}(\text{reconfig}) \times \text{reconfig} \preceq \text{spec}$$

4.2.2 Extension. Finally, in the scenario where Alice aims to extends Bob’s network configuration to also forward his public data to her, we imagine that Alice’s edge router address is $H_5$. In order to receive Bob’s public data, we can envision two possible scenarios. In the first, we imagine that Bob is contacted by Alice via some application-level process, and updates Bob’s sleep monitor to send public data to $H_5$. Then Alice can request to extend Bob’s existing firewall with a new rule allowing public data out of the network. That is, we can define $w_{\text{extend}}$ as:

$$w_{\text{extend}} \triangleq sw = ER \cdot dst = H_5 \cdot meta = \text{PUB} +$$

and Bob would install the program $\text{extend}_1$:

$$\text{extend}_1 \triangleq \text{reconfig} + (f \cdot w_{\text{extend}} \cdot t)^*$$

Another option is for Alice to duplicate traffic intended for $H_3$ and address it to $H_5$ entirely at the network level. In this case she could request to extend the forwarding behavior at $ER$ via $f_{\text{extend}}$:

$$f_{\text{extend}} \triangleq sw = SH_1 \cdot pt \leftarrow 2 +$$
$$\quad (sw = SH_2 \cdot pt \leftarrow 2 +)$$
$$\quad (sw = ER \cdot dst = H_3 \cdot dst \leftarrow H_5 \cdot pt \leftarrow 1)$$

and Bob would install the program $\text{extend}_2$:

$$\text{extend}_2 \triangleq \text{reconfig} + (f_{\text{extend}} \cdot w \cdot t)^*$$

In our implementation we explore the second option as described in Section 5, because the sleep monitor application we used allows only one data destination address to be specified.

Putting together these definitions in the PCNC framework, successful verification of the second sort of extension request establishes validity of the following judgment:

$$\emptyset, \Gamma \vdash \langle \text{witness}_2, \text{extend}_2, \text{spec} \rangle :$$
$$K_{\text{Bob}} \text{Says } \text{extend}_1(f_{\text{extend}} \cdot w \cdot t^*) \times \text{extend}_2 \preceq \text{spec}$$

4.3 Additional Concerns

It is important to note that multiple PCNC servers can exist in the same federated network that support their own policies. For example, in the above scenario Bob is only the administrator of his own network. Alice could also support reprogramming of her own network with an edge controller running a PCNC server, and by expressing trust in FedMem members in a similar manner as Bob, for example via the credential:

$$K_{\text{Alice}} \text{Says } K_{\text{FedMem}} \text{ SpeaksFor } K_{\text{Alice}} \text{ on } (X : \text{extend}(X))$$

Of course, Bob or other principals would still need to be properly credentialed as FedMem members by the FedMem CA. In particular, Bob could not spoof membership using a credential like,

$$K_{\text{Bob}} \text{Says } \text{BX}. K_{\text{Bob}} \text{ Says } X \rightarrow K_{\text{FedMem}} \text{ Says } X$$

since hand-off will not logically apply to establish the delegation—i.e., the verification would fail during System $F_{\text{Says}}$ type checking.

Another practical concern is certificate revocation—the credentials discussed above are effectively irrevocable, which is problematic if certain principals turn out to be untrustworthy. However, credential revocation can be defined in NAL—e.g., via timestamps and a notion of trusted local time, as explored in previous work [35]. We omit consideration of revocation here for brevity.

Finally, we note that our scenario assumes that trust domains (e.g., Bob’s network) have not been compromised. Otherwise, private data could easily be leaked—e.g., by internally relabeling private data packets as public. One method to address the problem of device compromise is to use PCNC itself to segment the network into different trust domains, and validate packets upon ingress. For example, if the network contains an untrusted device, we could use the switches at the trust boundary to drop packets that originate from that device and carry the PUB tag, because only trusted devices should add that tag. Moreover, we can use PCNC to ensure that this behavioral property is always enforced, even when the switches are reconfigured by other principals. As for individual device compromise, while defense against such threats falls out of the scope of this paper, we could mitigate such risks by employing solutions for device authentication [10, 33, 34], and/or for authorizing firmware updates to avoid malware installation [26, 27].

5 PCNC IMPLEMENTATION

We have implemented the PCNC client-server framework described in Section 3 in both a Mininet [39] simulation, and in a real Raspberry Pi-based network with sleep tracking and blood pressure monitoring devices. We extend the existing Frenetic [12] framework with features for communicating and verifying PCNC messages. In this Section we describe important details of the implementation, and evaluate its performance using the PCNC instance formulated in Section 4.

5.1 Client and Wire Format

The PCNC client accepts source code of a proof term, configuration program, and credentials as input. It generates a JSON object intended for communication to the server over HTTPS. For expressions $e$, types $t$, and NetKAT programs $p$, each has an $e$-expression representation and we denote their serialized format as $[e]$, $[\tau]$, and $[p]$, respectively. In full detail, the client takes as input the following elements:

i. NetKAT source code program $p$.

ii. System $F_{\text{Says}}$ source code proof term $e$.

iii. A list $\Gamma$ of named credentials each of the form $\text{cred} : K_A \text{ Says } \tau$.

iv. A list of private key signature $\text{cred} : \text{sig}(K_A^{-1}, [\tau])$ for each credential in $\Gamma$. 
The PCNC server receives messages and processes them—it verifies that requests are authorized, that submitted reconfiguration program in proof term instantiates this schema with definitions from Section 4, specifically assigned to values 1000 or 1001 to denote PUB and VLAN 1001, respectively. The NetKAT policies are defined in Section 4.2, except there we made some simplifications that define in Section 4.2, except there we made some simplifications that fine-tune the syntax, while we elide details of the cryptographic material and the serialized format of NetKAT programs.

5.2 Server and Configuration Workflow

The PCNC server receives messages and processes them—it verifies that requests are authorized, that submitted reconfiguration programs adhere to behavioral policies, and then compiles and deploys derived tables to network components. The server has the following components:

i. A parser that converts JSON-formatted PCNC messages to elements \( \Gamma, e, p \), and a list of signatures \( \text{sigs} \) indexed by credential name.

ii. An authorization check to prove that the request is allowed given the proof witness \( \epsilon \) and credentials in \( \Gamma \).

iii. A behavioral check to verify \( p \triangleleft \text{spec} \) given a policy specification \( \text{spec} \).

iv. Frenetic compilation of \( p \) to OpenFlow v1.3 tables.

v. Deployment of tables to network devices.

We describe authorization and behavioral checks in more detail as follows. Compilation and deployment to OpenFlow tables use existing Frenetic technology. As discussed in Section 2.3.2, Frenetic verification (item iii) is based on an existing bisimulation algorithm, but our implementation includes some new optimizations.

5.2.1 Authorization Check.

In the authorization check, we first verify that \( \text{sig}(\text{cred}) \) is a valid signature for \( \Gamma(\text{cred}) = K_A \text{Says} \tau \) for each \( \text{cred} \in \text{domain}(\Gamma) \), using the public key \( K_A \) in the credential itself. Following this verification step, we type check \( \Gamma \vdash \epsilon : \text{reconfig}(p) \). This type check verifies authorization, as well as the authenticity and integrity of \( p \), as discussed in Section 3.1.

5.2.2 Behavioral Check.

To perform the behavioral check, we obtain \( A(\text{spec}) \) and \( A(p + \text{spec}) \), which are the determined NetKAT automata derived from \( \text{spec} \) and \( p + \text{spec} \), respectively. Then we check \( A(p + \text{spec}) \sim A(\text{spec}) \) using the algorithm described in Section 2.3, leveraging FDD-based optimizations discussed in Section 3.2. This verifies that \( p \triangleleft \text{spec} \) and thus behavioral compliance of \( p \). The authorization and behavioral components together obtain validity of the target System F\text{Says} judgment for PCNC verification as described at the very end of Section 3.

5.3 Virtual Network Experiment

We first used a virtual network environment to evaluate the efficacy and usefulness of PCNC proof for authorization and NetKAT for defining policy on the data plane. Specifically, we used Mininet [39] with Frenetic [12] to create a virtual network topology with routers in an SDN environment, that is endowed with PCNC server capabilities. Our network configuration shown in Figure 5 reflects the case study introduced in Section 1 and formalized in the PCNC framework in Section 4.

The topology specifications and configuration, reconfiguration, and extension programs we tested are essentially the same as defined in Section 4.2, except there we made some simplifications that need to be fleshed out for testing in Mininet (and on real platforms). For example addresses \( H3, H4, \) and \( H5 \) need to be replaced with real IP addresses. And unfortunately Frenetic uses OpenFlow 1.0 that does not provide the metadata field (OpenFlow 1.1 or later versions support the metadata field), but we circumvent this hurdle by using the vланID field, specifically assigned to values 1000 or 1001 to denote PUB and 911 flags, respectively. The NetKAT policies
were implemented in OCaml [21]. The policies were tested with the Linux utility `nc` to facilitate communication channels between the destination hosts and the device hosts.

5.3.1 Results. The size of the tested PCNC messages were 2.6KB including all cryptographic material which constitutes the bulk of the encoding. Authorization and behavioral verification on an x86-64 laptop computer as implemented for the Mininet version of the server takes 0.172 seconds. The size of the PCNC server binary for x86-64 is 19.3MB for the verification components and cryptography library, and 47MB for the Frenetic codebase.

The results from this testing also show that the edge router allows packets with proper header information to pass—i.e., packets with PUB go to H3 while packets with 911 are forwarded to H4. The router also successfully drops packets with improper header information. All of the source code for the policies and documents on how to run them with Frenetic and Mininet is available online.²

5.4 Real Network Testbed

We also tested PCNC in a real network. In this section we discuss configuration with hardware and software components, and evaluation of the use case. Notably, we show that PCNC can run effectively on embedded devices such as Raspberry Pis, an attractive platform for adding security and privacy protections in home IoT systems.

5.4.1 Testbed Configuration. As shown in Figure 9, the testbed consists of two IoT devices connected to their remotely-located servers through an edge router. The two IoT devices are a Fitbit Charge 3 [11] for tracking sleep data and an iHealth Feel wireless blood pressure monitor [20]. The Fitbit Charge tracks the sleep activities of the user, and periodically sends data to the Fitbit servers running on Cloudflare [9] via its app on an Android phone. Similarly, the blood pressure monitor aggregates blood pressure data and communicates with the Withings servers [40] via the iHealth app on the phone. The phone connects to a wireless access point, then to the edge router that runs Open VSwitch (OVS version 2.7) [30] for external communication. We used a Raspberry Pi 3 Model B+ with a 1.4GHz 64-bit quad-core processor [14], wireless LAN, Bluetooth, and Ethernet for both the switch and access points. We used the Linux host access point daemon (`hostapd`) to turn the Pi to a virtual access point and to create a virtual LAN, and `dnsmasq` as a lightweight DHCP and caching DNS server. We used a laptop with a 2.2 GHz Intel Core i7 and 16GB RAM as the NAT server that is connected to the switch through an Ethernet interface, and another laptop for the edge device that also receives the same data from the fitbit. We bridge the wireless and Ethernet interfaces using the (`bridge-utils`) available in Ubuntu. The OVS data plane is controlled by a Frenetic controller component of the PCNC server which is located on the Raspberry Pi, that performs verification and deployment of programs via OpenFlow. In fact it would be easy to also incorporate the NAT server on the Raspberry Pi, indicating the feasibility of a low-cost embedded platform to support PCNC functionality.

5.4.2 Authorization and Behavioral Policies. The PCNC server runs on the Raspberry Pi and uses the same implementation as developed for the Mininet experiment. The wire format is as defined previously, and for this experiment we assume the same authorization proof target, credentials, etc. as for the Mininet experiment. The PCNC evaluation example for the Raspberry Pi-based server is therefore the same as in Figure 7, except with modifications to the reconfiguration program and specification.

Again, we use `vlantid` values of 1000 and 1001 to denote PUB and 911 data, respectively. The firewall component of the behavioral policy specification for the testbed server is as follows. A technical detail is that this specification allows for messages sent via UDP protocol for DNS lookups (via the condition `ipProto = 17`) as required by the testbed for initialization:

<table>
<thead>
<tr>
<th>Match</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP4Dst = 104.16.65.50</td>
<td>Vlan = 1000</td>
</tr>
<tr>
<td>IP4Dst = 104.16.66.50</td>
<td>Vlan = 1000</td>
</tr>
</tbody>
</table>

Table 2: The flow table at the edge router before the reconfiguration where the destination IP addresses are the two fitbit servers, VLAN is used as the metadata field (VLAN = 1000 represents PUB), and the outgoing port is 1.

Figure 9: Real network testbed with a Fitbit Charge 3 for sleep tracking and an iHealth Feel wireless blood pressure monitor.

![Figure 9: Real network testbed with a Fitbit Charge 3 for sleep tracking and an iHealth Feel wireless blood pressure monitor.](image)

Table 1: Packets that are allowed (√) or blocked (✗) for the initial policy and after the policy reconfiguration.

<table>
<thead>
<tr>
<th>No.</th>
<th>Network</th>
<th>Destination</th>
<th>TCP port</th>
<th>Type</th>
<th>Initial</th>
<th>Reconfig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>104.16.65.50</td>
<td>443</td>
<td>PUB</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td>104.16.66.50</td>
<td>443</td>
<td>PUB</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td>89.30.121.150</td>
<td>443</td>
<td>911</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>151.101.146.217</td>
<td>443</td>
<td>911</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>5</td>
<td>129.21.62.150</td>
<td>443</td>
<td>PUB</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

²https://github.com/uvm-plaid/PCNC_CCS_2019
Table 3: The flow table at the edge router after the reconfiguration where two new entries are added with VLAN = 1001, which denotes 911, destined to the iHealth servers for blood pressure data.

<table>
<thead>
<tr>
<th>IP4Dst</th>
<th>Vlan</th>
<th>EthType</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>104.16.65.50</td>
<td>1000</td>
<td>ip/800</td>
<td>SetField(ipDst, 129.21.62.150) Output(1) + Output(1)</td>
</tr>
<tr>
<td>104.16.66.50</td>
<td>1000</td>
<td>ip/800</td>
<td>SetField(ipDst, 129.21.62.150) Output(1) + Output(1)</td>
</tr>
<tr>
<td>151.101.146.217</td>
<td>1001</td>
<td>ip/800</td>
<td>Output(1)</td>
</tr>
<tr>
<td>89.30.121.150</td>
<td>1001</td>
<td>ip/800</td>
<td>Output(1)</td>
</tr>
</tbody>
</table>

Table data (PUB) is also replicated to the edge device at 129.21.62.150. In Table 1, the types of packets are displayed with their destination before and after the reconfiguration. The reconfiguration firewall also specifies an additional well-formedness condition that TCP destination port 443 should be used for all communications:

\[
\text{Filter (not (switch = 346653522121)) or (switch = 346653522121 and (vlanId=1000 and ip4Dst=89.30.121.150 and tcpDstPort=443) or (vlanId=1001 and ip4Dst=89.30.121.150 and tcpDstPort=443) or (vlanId=1000 and ip4Dst=151.101.146.217 and tcpDstPort=443) or (vlanId=1001 and ip4Dst=104.16.66.50 and tcpDstPort=443) or (vlanId=1001 and ip4Dst=104.16.65.50 and tcpDstPort=443) or (vlanId=1000 and ip4Dst=104.16.66.50 and tcpDstPort=443) or (vlanId=1001 and ip4Dst=104.16.65.50 and tcpDstPort=443) or (vlanId=1000 and ip4Dst=104.16.66.50 and tcpDstPort=443) or (vlanId=1001 and ip4Dst=104.16.65.50 and tcpDstPort=443) or (vlanId=1000 and ip4Dst=104.16.66.50 and tcpDstPort=443)) or (ipProto = 17))}
\]

Additionally, the forwarding policy is revised to duplicate packets from H3 to H5 for the extended network presented in Section 4.2.2. The firewall policy is not changed as we use the second approach (see Section 4.2.2) since the iHealth device allows only one destination address to be specified. The revised forwarding policy is:

\[
\text{(if switch = 346653522121)} \quad \text{then (ip4Dst := 104.16.66.50; port := 1)} \quad \text{else port := 55555 (* packet drop *) )}
\]

where the IP address of H5 is 129.21.61.113.

5.4.3 Results. As for the Mininet example, the case study PCNC messages for the tested are 2.6KB including all cryptographic material which constitutes the bulk of the encoding. Authorization and behavioral verification on the Raspberry Pi server takes 0.444 seconds. The size of the PCNC server binary for Raspberry Pi is 13MB for the verification components and cryptography library, and 27MB for the Frenetic codebase.

In Tables 2 and 3, part of the flow tables is shown, specifically the entries for forwarding from the devices to the servers, before and after the reconfiguration in the OVS switch. In the initial flow table, the packets destined to either 151.101.146.217 or 89.30.121.150 (with VLAN = 1001, i.e., emergency data from iHealth) are blocked, while other traffic (data from fitbit, i.e., VLAN = 1000) is forwarded to outgoing port 1, where Output(n) means that traffic is to be replicated to port n. After the reconfiguration and extension, the flow table (Table 3) is changed to forward both the PUB and 911 traffic to the outgoing port, i.e., the emergency data is no longer blocked, and the data is replicated to another destination set by SetField(ipDst, 129.21.62.150). We also captured packets from the ingress and egress ports of the switch to ensure that the installed policy takes effect. The flow tables and entire .pcap file containing the captured packets are available online.\[^2\]

6 CONCLUSION

In this paper we developed the Proof Carrying Network Code (PCNC) framework, that allows software defined network (SDN) programming by multiple, possibly non-local administrative domains in a secure manner. PCNC provides features for checking authorization of administrative domains for network programming, and allows programs themselves to be verified with respect to behavioral policy specifications.

PCNC is based on mathematically well-founded theories, specifically Nexus Authorization Logic and the NetKAT programming language. We developed a method for verifying behavioral properties that leverages the decidable equational theory of NetKAT. We also introduced a new language and type theory called System $\text{F}_{\text{ Says}}$ that provides proof terms for an authorization logic via a types-as-formulas correspondence.

To evaluate the practicality of PCNC, we implemented it in both simulated and real network settings, and considered the use-case scenario of an home health monitoring network. The latter incorporated a sleep monitor, where privacy concerns are relevant, and a heart rate monitor that may report emergency data that must be allowed to escape the network. We showed how PCNC can be used to support these sorts of applications and security concerns, while allowing external users to reconfigure local networks, even when direct trust relationships do not exist with those users as in a federated setting. Our results show that the PCNC system can support authorization, verification, and deployment of network programs efficiently and with a small binary footprint, even when installed on embedded devices. PCNC messages themselves are shown to be of manageable size in our case study.

ACKNOWLEDGMENTS

The authors wish to thank the CCS reviewers, Jonathan Dilorenzo, Ryan Doenges, and Fred Schneider for helpful comments and suggestions. Hossein Hojjat, Mark Reitblatt, and Lindsey Stuntz contributed many early ideas related to PCNC. This work was supported in part by the NSF under grants 1413972, 1717581, 1718036, and 1718083, and gifts from Fujitsu and InfoSys.

REFERENCES


\[^2\]https://github.com/uwm-plaid/PCNC_CCS_2019
A System $\text{F}_{\text{Says}}$ DEFINITION

Section 3 presented a summary of System $\text{F}_{\text{Says}}$. This appendix provides a more complete definition.

The syntax of System $\text{F}_{\text{Says}}$ is given in Figure 10. The language is an extension of System $F$ with a Says modal type to represent the Says modality in NAL-light, and other features to represent NAL-light connectives. Types $\tau$ of System $\text{F}_{\text{Says}}$ have a tight correspondence with NAL-light formulas $F$. The System $F$ fragment of System $\text{F}_{\text{Says}}$ is adequate to represent implication and higher order quantification as in NAL-light, and the addition of sum ($+$) and product ($\times$) types are adequate to represent disjunction and conjunction. The Says modality is realized using the return and bind style. Existential quantification is realized using standard pack and unpack terms. The predicate reconfig($p$) asserts the intent to install configuration program $p$, while $p < q$ asserts semantic containment of $p$ in $q$. The language could be endowed with a type preserving reduction semantics [31], but we leave this as future work.

The System $\text{F}_{\text{Says}}$ type derivation rules are given in Figure 11. There are two main judgment forms, well-formedness, $S \vdash \tau$ where $S$ is a set of variables assumed to be in scope and $\tau$ is required to have no free variables outside of $S$, and typing judgments $S, \Gamma \vdash e : \tau$ where $\Gamma$ is a free variable typing environment. These rules are fairly standard and include System $F$-style polymorphism, existential quantification for pack and unpack, and monadic typing rules for return and bind. Most notable is the Equiv rule, which is predicated on the NetKAT equivalence $p \equiv q$. [32] George C. Necula. 1997. Proof-carrying Code. In ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). 106–119. https://doi.org/10.1145/263649.263712
\[ p \in \text{prog} \quad \vdash \quad \ldots \] NetKAT Programs

\[ X \in \text{tvar} \quad \vdash \quad \ldots \] type variables

\[ A \in \text{pals} \quad \vdash \quad \ldots \] principles

\[ \pi \in \text{type} \quad \vdash \quad \text{unit} \mid \pi \times \pi \mid \pi \to \pi \mid X \mid \forall X.\pi \mid \exists X.\pi \] type variables

\[ x \in \text{var} \quad \vdash \quad \ldots \] term variables

\[ e \in \text{exp} \quad \vdash \quad \text{tt} \mid \text{inl}(e) \mid \text{inr}(e) \mid \text{lam}(e) \mid \text{proj}(e) \mid \text{projl}(e) \mid \text{projr}(e) \mid \text{let} \] unit

\[ v \in \text{val} \quad \vdash \quad \text{tt} \mid \text{inl}(v) \mid \text{inr}(v) \mid \text{lam}(v) \mid \text{proj}(v) \mid \text{projl}(v) \mid \text{projr}(v) \mid \text{let} \] sums

\[ \tau \in \text{type} \; \text{val} \quad \vdash \quad \text{tt} \mid \text{inl}(\tau) \mid \text{inr}(\tau) \mid \text{lam}(\tau) \mid \text{proj}(\tau) \mid \text{projl}(\tau) \mid \text{projr}(\tau) \mid \text{let} \] products

\[ \Gamma \in \text{tenv} \quad \vdash \quad \text{var} \to \text{type} \] type environment

\[ S \in \text{scope} \quad \vdash \quad \phi(tenv) \] type scope

---

### Figure 10: System F_{\text{Says}} Syntax

<table>
<thead>
<tr>
<th>Rule</th>
<th>Syntactic Formulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ S + r ]</td>
<td>Equiv: [ S + p \equiv p' ]</td>
</tr>
<tr>
<td>[ S, \Gamma \vdash e : \tau ]</td>
<td>Type Well-formedness: [ S, \Gamma \vdash e : \tau ]</td>
</tr>
</tbody>
</table>

### Figure 11: System F_{\text{Says}} Type Derivation Rules.