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

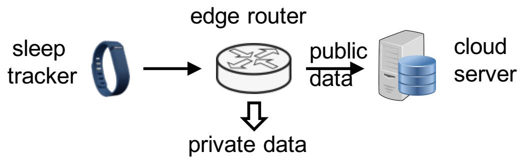
ACM Reference Format:
Christian Skalka, John Ring, David Darais, Minseok Kwon, Sahil Gupta, Kyle Diller, Steffen Smolka, and Nate Foster. 2019. Proof-Carrying Network Code. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3319535.3363214

1 INTRODUCTION

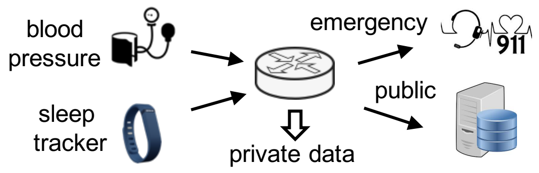
Computer networks play a critical role in implementing security policies, often serving as the first line of defense against malicious attacks. Although there are a growing number of tools for specifying and verifying behavior in software-defined networks (SDNs), most are unable to handle the challenges that arise in networks with multiple administrative domains.

To illustrate, consider the following concrete scenario. Suppose that a health monitoring system is connected to a home network with one or more IoT (Internet of Things) devices—e.g., as shown in Figure 1(a), a fitness tracker monitors the sleep patterns of its residents, using Bluetooth or WiFi to connect to a switch that provides connectivity to other devices on the local network. There is also an edge router that connects the home network to the Internet. To prevent health data from being sent externally, the network is configured as follows. The switch uses VLANs (virtual local area networks) to isolate different segments of the network from each other, while the router uses a firewall to filter incoming and outgoing traffic and a NAT (network address translator) to convert between private addresses used in the home network and public addresses used on the Internet. For example, the switch might classify packets coming from the fitness tracker, adding a tag to indicate if they are private (e.g., fine-grained location information) or public (e.g., aggregate, anonymous sleep data), and the firewall might drop packets tagged as private.

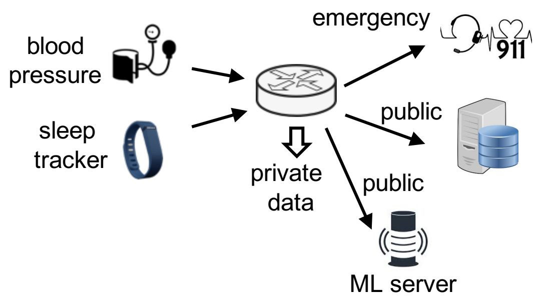
Suppose we add a second device that monitors blood pressure. In order to report information in an emergency—e.g., when blood pressure becomes dangerously high—the network must be reconfigured. In particular, the filtering rules installed on the firewall must be relaxed to allow data to be released from the network even if it is not public during an emergency. One possible approach is to



(a) Initially, only public data generated by the sleep tracker can pass the edge router.



(b) After the blood pressure monitor is added, the emergency data is not filtered at the router reaching the 911 service.



(c) Public data from the sleep tracker is delivered to a federated edge computing service upon request.

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

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 transit 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].

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].

1.2 Contributions and Related Work

In this paper we develop theory and establish an architecture for PCNC, including a prototype implementation and evaluation on realistic use cases both in simulation and on a hardware testbed.

In Section 2, we describe the theoretical foundations of PCNC. We formulate a variant of NAL, called NAL_{light}, that captures *application-level assertions* about NetKAT programs. Thus, judgments in NAL_{light} model an authorization and a behavioral component. The authorization component combines the expressiveness of a higher-order logic extended with modalities for belief ascription and delegation, while properties involving NetKAT programs can be expressed as application-level assertions (though decidable properties, such as equivalence, are of particular interest). For properties that can be reduced to equivalence, we define an algorithm with optimizations for checking equivalence of NetKAT programs that is based on previous work [13].

Also in Section 2, we develop a language for proof representation, called System F_{Says}, a typed term calculus that enjoys a Curry-Howard types-as-formulas correspondence with NAL_{light} as stated in Theorem 2.1. Our approach to proof representation is similar to CDD [2], which demonstrated the benefits of the Curry-Howard approach, including a reduction semantics that can support proof minimization.

In Section 3, we describe how these elements are combined in a system for enforcing security in SDNs. We propose a specific judgment form that can be used to verify requests to either reconfigure or extend the configuration of the network using NetKAT. Verification of requests subsumes System F_{Says} type checking for authorization and decidable equivalence checking for behavioral verification. Classic work on PCC focuses mostly on supporting purely behavioral policies, but previous work has considered Proof-Carrying Authentication [5] and Authorization [6, 15], also known as PCA, to support verification of authorization policies in distributed systems. Thus, PCNC unifies concepts explored in PCC and PCA to obtain a uniform language framework for expressing proofs of authorization and behavioral policy compliance to support SDN network configuration in federated settings.

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_{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 and a signature verification scheme for NAL_{light} credentials. We evaluate the implementation using the case study and report on verification overhead.

2 PCNC FOUNDATIONS

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 Authorization Logic

A number of *authorization logics* have been proposed in previous work that offer features for policy expression [1, 3, 8, 17, 19]. In general, modern authorization logics equate authorization decisions with provability of formulas, where given authorization credentials are modeled as assumptions for the proof derivation. They typically extend an underlying logic (e.g., first-order classical logic) with *SpeaksFor* and *Says* modalities endowed with either a possible worlds semantics [16] or a related semantics of “belief” [19]. The *SpeaksFor* modality allows the authority of one principal to be “handed off” to another, supporting expression of the delegation of authority. The *Says* modality is typically taken to ascribe beliefs to principals and can be used to express authorized credentials among others, but the precise interpretation of *Says* is a subtle matter with significant consequences. We discuss this issue here as it is relevant to our proof representations.

2.1.1 Monadic Interpretation of Says. The *Says* modality has historically been related to classical modalities. The interpretation of *Says* in the original presentation of Nexus Authorization Logic [35] (NAL) can formally be said to be *lax* and in particular can be embedded in the logic S4 [16]. Intuitively, the lax interpretation allows a more liberal ascription of beliefs to principals. Formally, a lax system includes the axiom $\forall X.X \Rightarrow A\ Says\ X$ (whereas in

Syntax

$$\begin{array}{rcl} F, G, H & ::= & f(t, \ldots) \\ & \mid & \text{true} \\ & \mid & \text{false} \\ & \mid & F \textit{ Says } F \\ & \mid & X \\ & \mid & F \wedge F \\ & \mid & F \vee F \\ & \mid & F \Rightarrow F \\ & \mid & \forall X.F \\ & \mid & \exists X.F \end{array}$$

Proof Derivation Rules

$$\text{True}\ \ \Delta \vdash \text{true} \qquad \text{Assume}\ \ \Delta, F \vdash F \qquad \text{Unit}\ \frac{\Delta \vdash F}{\Delta \vdash A \textit{ Says } F} \qquad \text{Bind}\ \frac{\Delta \vdash A \textit{ Says } F \quad \Delta, F \vdash A \textit{ Says } G}{\Delta \vdash A \textit{ Says } G} \qquad \Rightarrow\text{-Intro}\ \frac{\Delta, F \vdash G}{\Delta \vdash F \Rightarrow G} \qquad \Rightarrow\text{-Elim}\ \frac{\Delta \vdash F \Rightarrow G \quad \Delta \vdash F}{\Delta \vdash G}$$

$$\vee\text{-Intro}_L\ \frac{\Delta \vdash G}{\Delta \vdash F \vee G} \qquad \vee\text{-Intro}_R\ \frac{\Delta \vdash F}{\Delta \vdash F \vee G} \qquad \vee\text{-Elim}\ \frac{\Delta \vdash F \vee G \quad \Delta, F \vdash H \quad \Delta, G \vdash H}{\Delta \vdash H} \qquad \wedge\text{-Intro}\ \frac{\Delta \vdash F \quad \Delta \vdash G}{\Delta \vdash F \wedge G} \qquad \wedge\text{-Elim}_L\ \frac{\Delta \vdash F \wedge G}{\Delta \vdash F} \qquad \wedge\text{-Elim}_R\ \frac{\Delta \vdash F \wedge G}{\Delta \vdash G}$$

$$\forall\text{-Intro}\ \frac{\Delta \vdash F \quad X \notin fv(\Delta)}{\Delta \vdash \forall X.F} \qquad \forall\text{-Elim}\ \frac{\Delta \vdash \forall X.F}{\Delta \vdash F[G/X]} \qquad \exists\text{-Intro}\ \frac{\Delta \vdash F[G/X]}{\Delta \vdash \exists X.F} \qquad \exists\text{-Elim}\ \frac{\Delta \vdash \exists X.F \quad \Delta, F \vdash G \quad X \notin (fv(\Delta) \cup fv(G))}{\Delta \vdash G}$$

Figure 2: NAL$_{light}$ syntax and proof derivation rules.

a non-lax system we can only deduce $X \Rightarrow A \textit{ Says } X$ 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*:

$$\forall X, Y.(A \textit{ Says } X) \Rightarrow (X \vee A \textit{ Says } Y)$$

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, in 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 ($\forall$) and existential ($\exists$) quantification, and standard logical connectives ($\wedge$, $\vee$, and $\Rightarrow$). 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 *SpeaksFor* modality is defined as syntactic sugar in terms of higher-order quantification:

$$A \textit{ SpeaksFor } B \quad \triangleq \quad \forall X.(A \textit{ Says } X) \Rightarrow (B \textit{ says } X)$$

A restricted form of delegation is also supported,

$$\begin{gathered} A \textit{ SpeaksFor } B \textit{ on } (X_1 \cdots X_n : F) \\ \triangleq \\ \forall X_1. \ldots .\forall X_n.(A \textit{ Says } F) \Rightarrow (B \textit{ says } F) \end{gathered}$$

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, intuitionistic propositional logic is PSPACE-complete [38], while intuitionistic 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.

[1]Though it should be noted that a Curry-Howard isomorphism with non-lax modalities can be formulated, as demonstrated in [31].

Syntax

$$\begin{array}{llll}
\textit{types} & \tau & ::= & \text{unit} \mid \tau + \tau \mid \tau \times \tau \mid \tau \to \tau \mid X \mid \forall X.\tau \mid \exists X.\tau \mid \\
 & & & \tau\ \textit{Says}\ \tau \mid \text{reconfig}(\tau) \mid \text{extend}(\tau) \mid \tau \preccurlyeq \tau \mid p \mid A \\
\textit{terms} & e & ::= & \text{tt} \mid \\
 & & & \text{inl}(e) \text{ as } \cdot + \tau \mid \text{inr}(e) \text{ as } \tau + \cdot \mid \text{case}(e)\{x.e\}\{x.e\} \mid \\
 & & & \langle e, e \rangle \mid \text{projl}(e) \mid \text{projr}(e) \mid \\
 & & & x \mid \text{let } x \coloneqq e \text{ in } e \mid \lambda(x : \tau).e \mid e(e) \mid \\
 & & & \Lambda X.e \mid e[\tau] \mid \langle^* \tau, e \rangle \text{ as } \exists X.\tau \mid \text{let } \langle^* X, x \rangle \coloneqq e \text{ in } e \mid \\
 & & & \text{ret}^A(e) \mid x \leftarrow e \,;\, e \mid e \preccurlyeq e \mid p
\end{array}$$

Typing

Bind

$$\frac{S, \Gamma \vdash e_1 : A\ \textit{Says}\ \tau_1 \qquad S, \Gamma[x \mapsto \tau_1] \vdash e_2 : A\ \textit{Says}\ \tau_2}{S, \Gamma \vdash x \leftarrow e_1 \,;\, e_2 : A\ \textit{Says}\ \tau_2}$$

Ret

$$\frac{S, \Gamma \vdash e : \tau}{S, \Gamma \vdash \text{ret}^A(e) : A\ \textit{Says}\ \tau}$$

TypeApply

$$\frac{S \vdash \tau' \qquad S, \Gamma \vdash e : \forall X.\tau}{S, \Gamma \vdash e[\tau'] : [\tau'/X]\tau}$$

Figure 3: Selected System F_{Says} syntax and typing rules.

2.1.4 Proof Representation: System F_{Says}. Judgments are of the form $\Delta \vdash F$, where Δ 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 Δ 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_{Says} which enjoys a Curry-Howard types-as-formulas correspondence with NAL_{light}. This approach has been explored previously for a different authorization logic [2].

The syntax of System F_{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 NAL_{light}, and other features to represent NAL_{light} connectives. Types τ of System F_{Says} have a tight correspondence with NAL_{light} formulas F. The System F fragment of System F_{Says} is adequate to represent implication and higher order quantification as in NAL_{light}, and the addition of sum (+) and product (×) 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_{Says} here specifically for PCNC, so we only include three atomic predicate forms: reconfig, extend, and $\preccurlyeq$. 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 $p \preccurlyeq q$ 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, Γ is an environment binding free term variables to types, and S is the set of type variables in scope. Considering the type τ as a NAL formula analogue, the term e is referred to as a *witness*. The type derivation rules have a tight correspondence with the NAL_{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. See the appendix for the full typing rules.

Syntax

$$\begin{array}{rll}
\text{Field} & f ::= f_1 \mid \cdots \mid f_k & \\
\text{Packet} & pk ::= \{f_1 = v_1, \cdots, f_k = v_k\} & \\
\text{History} & h ::= pk{::}\langle\rangle \mid pk{::}h & \\
\text{Predicate } a, b & ::= 1 & \textit{Id} \\
 & \mid 0 & \textit{Drop} \\
 & \mid f = n & \textit{Test} \\
 & \mid a + b & \textit{Or} \\
 & \mid a \cdot b & \textit{And} \\
 & \mid \neg a & \textit{Not} \\
\text{Policy } p, q & ::= a & \textit{Filter} \\
 & \mid f \leftarrow n & \textit{Modify} \\
 & \mid p + q & \textit{Union} \\
 & \mid p \cdot q & \textit{Sequence} \\
 & \mid p* & \textit{Iterate} \\
 & \mid \text{dup} & \textit{Duplicate}
\end{array}$$

Denotational Semantics

$$\begin{array}{c}
[\![p]\!] \in \mathsf{H} \to \mathcal{P}(\mathsf{H}) \\
[\![1]\!]\ h \triangleq \{h\} \\
[\![0]\!]\ h \triangleq \{\} \\
[\![f = n]\!]\ (pk{::}h) \triangleq \begin{cases} \{pk{::}h\} & \text{if } pk.f = n \\ \{\} & \text{otherwise} \end{cases} \\
[\![\neg a]\!]\ h \triangleq \{h\} \setminus ([\![a]\!]\ h) \\
[\![f \leftarrow n]\!]\ (pk{::}h) \triangleq \{pk[f \coloneqq n]{::}h\} \\
[\![p + q]\!]\ h \triangleq [\![p]\!]\ h \cup [\![q]\!]\ h \\
[\![p \cdot q]\!]\ h \triangleq ([\![p]\!] \bullet [\![q]\!])\ h \\
[\![p*]\!]\ h \triangleq \bigcup_{i \in \mathbb{N}} F^i\ h \\
\text{where } F^0\ h \triangleq \{h\} \\
\text{and } F^{i+1}\ h \triangleq ([\![p]\!] \bullet F^i)\ h \\
[\![\text{dup}]\!]\ (pk{::}h) \triangleq \{pk{::}(pk{::}h)\}
\end{array}$$

Figure 4: NetKAT: syntax and denotational semantics.

2.1.5 Types-as-Formulas Correspondence. Our main result for System F_{Says} is soundness of the representation—i.e., we show that if a System F_{Says} term is typeable at type τ, the formula corresponding to τ 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_{Says} nor an explicit false term.

Since the syntax of types and formulas used in System F_{Says} and NAL_{light} respectively do not match up, we define an interpretation of types as formulas denoted $\langle\tau\rangle$:

$$\begin{array}{rcl}
\langle \tau_1 \preccurlyeq \tau_2 \rangle & = & \langle \tau_1 \rangle \preccurlyeq \langle \tau_2 \rangle \\
\langle \tau_1 + \tau_2 \rangle & = & \langle \tau_1 \rangle \vee \langle \tau_2 \rangle \\
\langle \tau_1 \to \tau_2 \rangle & = & \langle \tau_1 \rangle \Rightarrow \langle \tau_2 \rangle \\
 & \vdots &
\end{array}$$

and so on. We extend this interpretation pointwise to type environments, which translate to lists of assumptions in NAL_{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_{Says} and NAL_{light}.

Theorem 2.1. *If $S, \Gamma \vdash e : \tau$ is derivable then so is $\langle\Gamma\rangle \vdash \langle\tau\rangle$.*

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

$$\forall A.\forall B.(A\ \textit{Says}\ B\ \textit{SpeaksFor}\ A) \Rightarrow (B\ \textit{SpeaksFor}\ A)$$

In System F_{Says} the type representation τ of this formula is:

$$\begin{array}{l}\forall A.\forall B.(A\ \textit{Says}\ (\forall X.B\ \textit{Says}\ X \to A\ \textit{Says}\ X)) \to \\ \qquad (\forall X.B\ \textit{Says}\ X \to A\ \textit{Says}\ X)\end{array}$$

and the following term e serves as a proof witness for this type τ, in the sense that $\varnothing, \varnothing \vdash e : \tau$ is valid:

$$\begin{array}{l}\Lambda A.\Lambda B.\lambda x : (A\ \textit{Says}\ (\forall X.B\ \textit{Says}\ X \to A\ \textit{Says}\ X))\ . \\ \qquad\qquad \Lambda X.\lambda z : B\ \textit{Says}\ X.w \leftarrow x; (w[X])(z)\end{array}$$

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 combinators such as union ($+$), 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 ($[\![\,]\!]$) 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 (*if a then p else q*) using union and sequence ($a \cdot p + \neg a \cdot q$).

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 p returns a subset of the packets returned by q 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 $p \preccurlyeq q$, with the following derivation rule in NAL$_{light}$, and its analogue for term witnessing in System F$_{Says}$:

$$\text{Contains}\ \frac{q \equiv p + q}{\Delta \vdash p \preccurlyeq q} \qquad \text{Contains}\ \frac{q \equiv p + q}{S, \Gamma \vdash p \preccurlyeq q : p \preccurlyeq q}$$

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 δ and ϵ. 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 $\ell \in S$. 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:

$$\begin{array}{rcl} \delta & : & (S \times PK) \to PK \to S \\ \epsilon & : & (S \times PK) \to PK \to 2 \end{array}$$

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 determinization methods [13]). Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be encoded as previously defined, and let δ_1, δ_2, and ϵ_1, ϵ_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 *equiv* can be defined as follows, using an accumulator π_c as a termination condition for previously explored configurations:

$$\begin{array}{l} equiv(c_1, c_2, \pi_c) \iff \\ \quad (c_1, c_2) \in \pi_c \qquad \vee \\ \quad \forall pk\ .\ (\epsilon_1\, c_1\, pk) = (\epsilon_2\, c_2\, pk) \qquad \wedge \\ \quad \forall pk\ .\ equiv((\delta_1\, c_1\, pk,\ pk), (\delta_2\, c_2\, pk,\ pk), \pi_c \cup \{(c_1, c_2)\}) \end{array}$$

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

$$\mathcal{A}_1 \sim \mathcal{A}_2 \iff \forall pk.equiv((0, pk), (0, pk), \varnothing)$$

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_{Says} proof witness e has been provided by the client:

$$\varnothing, \Gamma \vdash \langle e, p \preccurlyeq spec \rangle : \text{Root } Says \text{ reconfig}(p) \times (p \preccurlyeq spec)$$

In this judgment, the program p is the (re)configuration program to be installed and *spec* is the NetKAT behavioral policy specification. The assertion $p \preccurlyeq spec$ 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 conjunct Root *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 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:

$$\varnothing, \Gamma \vdash \langle e, p + q \preccurlyeq spec \rangle : \text{Root } Says \text{ extend}(p) \times (p + q \preccurlyeq spec)$$

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_{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 credentialed approach with public-key signatures allows this to scale to highly distributed settings.

3.1 Authorization in System F_{Says}

On the server side, validity of the submitted judgment is established by checking the validity of $\Gamma \vdash e : Root\ Says$ 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 $A\ Says\ \tau$ in its image. Our approach to this is standard—we represent principals A as public keys K_A, and for every assumption $A\ Says\ \tau$ in the image of Γ, we include signature $s = sig(K_A^{-1}, \tau)$ in the PCNC message, which is τ signed with A's private key. The server can then verify s given K_A and τ 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 could submit the following credential:

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

along with its signature:

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

and the proof term *cred*. Then given these items, on the Server we can verify:

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

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 *cred*.

Crucially, we further observe that there exists no e such that $\varnothing, \varnothing \vdash e : K_{\text{Bob}}\ Says$ reconfig(p) is valid, since reconfig(p) has no direct term witness and must follow from assumptions. Thus any installation request must be appropriately credentialed. 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 $p \preccurlyeq spec$ as a behavioral verification component. As discussed in Section 2.3, this is equivalent to proving $spec \equiv p + spec$ 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:

$$p \preccurlyeq spec \iff (\mathrm{A}(spec) \sim \mathrm{A}(p + spec))$$

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 $p \preccurlyeq spec$ on the Server it is necessary to obtain $\mathrm{A}(p + spec)$ there. While it would be possible to ship $\mathrm{A}(p)$, and then compute $\mathrm{A}(p + spec)$, 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 $\mathrm{A}(p)$, 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 administered 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:

$$\begin{array}{l} K_{\mathrm{Bob}}\ \mathit{Says} \\ \quad K_{\mathrm{NetCo}}\ \mathit{SpeaksFor}\ K_{\mathrm{Bob}}\ \mathit{on}\ (X : \mathrm{reconfig}(X)) \wedge \\ \quad K_{\mathrm{NetCo}}\ \mathit{SpeaksFor}\ K_{\mathrm{Bob}}\ \mathit{on}\ (X : \mathrm{extend}(X)) \end{array}$$

and the corresponding System F_{Says} type form is:

$$\begin{array}{l} \tau_{delegate} \triangleq \\ \quad K_{\mathrm{Bob}}\ \mathit{Says} \\ \qquad (\forall X.K_{\mathrm{NetCo}}\ \mathit{Says}\ \mathrm{reconfig}(X) \longrightarrow K_{\mathrm{Bob}}\ \mathit{Says}\ \mathrm{reconfig}(X)) \times \\ \qquad (\forall X.K_{\mathrm{NetCo}}\ \mathit{Says}\ \mathrm{extend}(X) \longrightarrow K_{\mathrm{Bob}}\ \mathit{Says}\ \mathrm{extend}(X)) \end{array}$$

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

$$sig(K_{\mathrm{Bob}}^{-1}, \tau_{delegate})$$

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

$$\tau_{reconfig} \triangleq K_{\mathrm{NetCo}}\ \mathit{Says}\ \mathrm{reconfig}(p)$$

along with the signature:

$$sig(K_{\mathrm{NetCo}}^{-1}, \tau_{reconfig})$$

Then to install p, NetCo submits Γ containing the following type bindings:

$$delegate : \tau_{delegate} \qquad\qquad reconfig : \tau_{reconfig}$$

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

$$witness_1 \triangleq config \longleftarrow delegate; (((\mathrm{projl}(config))[p])\ reconfig)$$

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

$$\varnothing, \Gamma \vdash witness_1 : K_{\mathrm{Bob}}\ \mathit{Says}\ \mathrm{reconfig}(p)$$

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:

$$K_{\mathrm{Bob}}\ \mathit{Says}\ K_{\mathrm{FedMem}}\ \mathit{SpeaksFor}\ K_{\mathrm{Bob}}\ \mathit{on}\ (X : \mathrm{extend}(X))$$

with the corresponding type form:

$$\begin{array}{l} \tau_{FedMem} \triangleq \\ \quad K_{\mathrm{Bob}}\ \mathit{Says}\ \forall X.K_{\mathrm{FedMem}}\ \mathit{Says}\ \mathrm{extend}(X) \longrightarrow K_{\mathrm{Bob}}\ \mathit{Says}\ \mathrm{extend}(X) \end{array}$$

along with the signature:

$$sig(K_{\mathrm{Bob}}^{-1}, \tau_{FedMem})$$

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

$$K_{\mathrm{FedMem}}\ \mathit{Says}\ K_{\mathrm{Alice}}\ \mathit{SpeaksFor}\ K_{\mathrm{FedMem}}$$

with the corresponding type form:

$$\tau_{Alice} \triangleq K_{\mathrm{FedMem}}\ \mathit{Says}\ \forall X.K_{\mathrm{Alice}}\ \mathit{Says}\ X \longrightarrow K_{\mathrm{FedMem}}\ \mathit{Says}\ X$$

along with the signature:

$$sig(K_{\mathrm{FedMem}}^{-1}, \tau_{Alice})$$

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 Γ and a term witness e such that $\varnothing, \Gamma \vdash e : K_{\text{Bob}}\ \mathit{Says}\ \text{extend}(p)$ 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 NAL_{light} and System F_{Says} forms):

$$\tau_{extend} \triangleq K_{\text{Alice}}\ \mathit{Says}\ \text{extend}(p)$$

along with the signature:

$$sig(K^{-1}_{\text{Alice}}, \tau_{extend})$$

Then Alice submits Γ containing the following type bindings:

$$fedmem : \tau_{FedMem} \qquad alice : \tau_{Alice} \qquad extend : \tau_{extend}$$

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

$$\begin{aligned}&witness_2 \triangleq\\ &\quad bobconfig \longleftarrow fedmem;\\ &\qquad bobconfig[p](config \longleftarrow alice;\ (config[\text{extend}(p)]\ extend))\end{aligned}$$

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

$$\varnothing, \Gamma \vdash witness_2 : K_{\text{Bob}}\ \mathit{Says}\ \text{extend}(p)$$

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 S_1, port P_1, to switch S_2, port P_2:

$$\text{dup} \cdot sw = S_1 \cdot pt = P_1 \cdot sw \leftarrow S_2 \cdot pt \leftarrow P_2 \cdot \text{dup}$$

By convention we will write such a link as follows:

$$[S_1 : P_1] \rightarrow [S_2 : P_2]$$

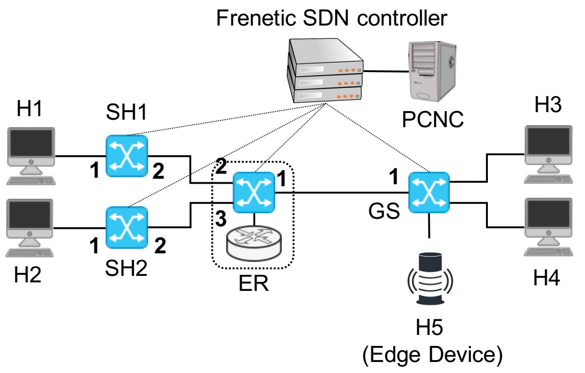


Figure 5: Our Mininet network topology and configuration where the numbers represent port numbers at the switches.

We can encode the network topology into NetKAT as follows:

$$\begin{aligned} t \quad \triangleq \quad & [SH1 : 2] \rightarrow [ER : 2] + [SH2 : 2] \rightarrow [ER : 3]+\\ & [ER : 2] \rightarrow [SH1 : 2] + [ER : 3] \rightarrow [SH3 : 2]+\\ & [ER : 1] \rightarrow [GS : 1]\end{aligned}$$

Intuitively, this program models the links between ER, $SH1$, and $SH2$, and from ER to GS. To keep the example simple, we will focus on the one-way flow of data from internal devices $H1$ and $H2$ toward the gateway GS—i.e., we will not specify the behavior of GS or of the network that connects it to external hosts. Hence, the local policy forwards from hosts $H1$ and $H2$ toward GS:

$$\begin{aligned} f \quad \triangleq \quad & (\text{sw} = SH1 \cdot \text{pt} \leftarrow 2)+\\ & (\text{sw} = SH2 \cdot \text{pt} \leftarrow 2)+\\ & (\text{sw} = ER \cdot \text{pt} \leftarrow 1)\end{aligned}$$

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

$$\begin{aligned} w \quad \triangleq \quad & \neg(\text{sw} = ER)+\\ & \text{sw} = ER \cdot \text{meta} = \text{PUB}+\\ & \text{sw} = ER \cdot \text{meta} = 911\end{aligned}$$

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

$$spec \triangleq (f \cdot w \cdot t)*$$

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 H_3:

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

and so the initial configuration is $(f \cdot w_{init} \cdot t)*$, and observe it is the case that $(f \cdot w_{init} \cdot t)* \preccurlyeq spec$. 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 $H4$ for emergency data (e.g., a care provider).

$$\begin{aligned} w_{reconfig} \quad \triangleq \quad & \neg(\text{sw} = ER) + \\ & \text{sw} = ER \cdot \text{dst} = H3 \cdot \text{meta} = \text{PUB} + \\ & \text{sw} = ER \cdot \text{dst} = H4 \cdot \text{meta} = 911 \end{aligned}$$

In full detail, the reconfiguration program, that we call *reconfig*, would be defined as:

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

and it is the case that $reconfig \preccurlyeq spec$, since reconfiguration code only *refines 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:

$$\begin{gathered} \varnothing, \Gamma \vdash \langle witness_1, reconfig \preccurlyeq spec \rangle : \\ K_{\text{Bob}}\ Says\ \text{reconfig}(reconfig) \times reconfig \preccurlyeq spec \end{gathered}$$

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 $H5$. 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 $H5$. 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_{extend} as:

$$w_{extend} \quad \triangleq \quad \text{sw} = ER \cdot \text{dst} = H5 \cdot \text{meta} = \text{PUB}$$

and Bob would install the program $extend_1$:

$$extend_1 \triangleq reconfig + (f \cdot w_{extend} \cdot t)*$$

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_{extend}:

$$\begin{aligned} f_{extend} \quad \triangleq \quad & \text{sw} = SH1 \cdot \text{pt} \leftarrow 2) + \\ & (\text{sw} = SH2 \cdot \text{pt} \leftarrow 2) + \\ & (\text{sw} = ER \cdot \text{dst} = H3 \cdot \text{dst} \leftarrow H5 \cdot \text{pt} \leftarrow 1) \end{aligned}$$

and Bob would install the program $extend_2$:

$$extend_2 \triangleq reconfig + (f_{extend} \cdot w \cdot t)*$$

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 judgement:

$$\begin{gathered} \varnothing, \Gamma \vdash \langle witness_2, extend_2 \preccurlyeq spec \rangle : \\ K_{\text{Bob}}\ Says\ \text{extend}((f_{extend} \cdot w \cdot t)*) \times extend_2 \preccurlyeq spec \end{gathered}$$

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:

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

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,

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

since hand-off will not logically apply to establish the delegation—i.e., the verification would fail during System F_{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 τ, and NetKAT programs p, each has an s-expression representation and we denote their serialized format as $\lfloor e \rfloor$, $\lfloor \tau \rfloor$, and $\lfloor p \rfloor$, respectively. In full detail, the client takes as input the following elements:

i. NetKat source code program p.
ii. System F_{Says} source code proof term e.
iii. A list Γ of named credentials each of the form $cred : K_A\ Says\ \tau$.
iv. A list of private key signature $cred : sig(K_A^{-1}, \lfloor \tau \rfloor)$ for each credential in Γ.

```
{
 "title": "PCNC Wire Format"
 "description": "PCNC wire format schema"
 "type": "object"
  "ty": { "type" : "string", "description": "install request" }
 "exp": { "type" : "string", "description": "install request proof"}
 "tenv":
  {
   "type" : "array",
   "items" :
    {
     "type": "object"
     "key": { "type" : "string", "description": "credential id" }
     "ty": { "type" : "string", "description": "credential" }
     "enc": { "type" : "string", "description": "credential signature" }
    }
  }
 "prog": { "type" : "string", "description": "configuration program"}
}
```

Figure 6: PCNC wire format schema.

```
{
 "ty":"(Says (Principal $K_{\mathrm{Bob}}$) Reconfig(Program $\lfloor reconfig \rfloor$))",
 "exp": "Bind config
         (Var delegate_cred)
         (Apply (TyApply Projl(Var config) (Program $\lfloor reconfig \rfloor$))
                        (Var reconfig_cred))"
 "tenv":
 [{"key" : "reconfig_cred",
   "ty" : "Says (Principal $K_{\mathrm{NetCo}}$) (Reconfig (Program $\lfloor reconfig \rfloor$))",
   "enc" : $sig(K^{-1}_{\mathrm{NetCo}}, \lfloor \tau_{reconfig} \rfloor)$},
  {"key" : "delegate_cred",
   "ty" : "Says (Principal $K_{\mathrm{Bob}}$)
                Prod (
                  (Forall X (Fun (Says (Principal $K_{\mathrm{NetCo}}$) Reconfig(TVar X))
                                 (Says (Principal $K_{\mathrm{Bob}}$) Reconfig(TVar X)))),
                  (Forall X (Fun (Says (Principal $K_{\mathrm{NetCo}}$) Extend(TVar X))
                                 (Says (Principal $K_{\mathrm{Bob}}$) Extend(TVar X)))))",
   "enc" : $sig(K^{-1}_{\mathrm{Bob}}, \lfloor \tau_{delegate} \rfloor)$}],
 "prog": $\lfloor reconfig \rfloor$
}
```

Figure 7: PCNC wire format example, with definitions from Section 4 reconfiguration scenario.

In our implementation we generate and check signatures using the Ring cryptography library [36]. Note, however, that the client does not require access to private keys, so it can use credentials signed and provided by non-local sources.

The client parses p, e, and Γ to a serialized format, associates signatures with their credentials, and generates a JSON object matching the schema defined in Figure 6. An example PCNC message that instantiates this schema with definitions from Section 4, specifically the reconfiguration scenario embodied in proof term $witness_1$ and program *reconfig*, is given in Figure 7. We show some details of the proof witness and credentials in serialized format to give a flavor of 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 Γ, e, p, and a list of signatures *sigs* indexed by credential name.
ii. An authorization check to prove that the request is allowed given the proof witness e and credentials in Γ.

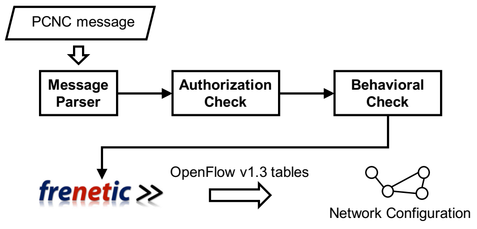


Figure 8: PCNC server workflow.

iii. A behavioral check to verify $p \preccurlyeq spec$ given a policy specification *spec*.
iv. Frenetic compilation of p to Open Flow 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 2.3.1, behavioral 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 $sigs(cred)$ is a valid signature for $\Gamma(cred) = K_A \; Says \; \tau$ for each $cred \in domain(\Gamma)$, using the public key K_A in the credential itself. Following this verification step, we type check $\varnothing, \Gamma \vdash e : \mathrm{reconfig}(p)$. 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 $\mathrm{A}(spec)$ and $\mathrm{A}(p + spec)$, which are the determinized NetKAT automata derived from *spec* and $p + spec$, respectively. Then we check $\mathrm{A}(p + spec) \sim \mathrm{A}(spec)$ using the algorithm described in Section 2.3, leveraging FDD-based optimizations discussed in Section 3.2. This verifies that $p \preccurlyeq spec$ and thus behavioral compliance of p. The authorization and behavioral components together obtain validity of the target System F_{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 `vlanID` field, specifically assigned to values 1000 or 1001 to denote PUB and 911 flags, respectively. The NetKAT policies

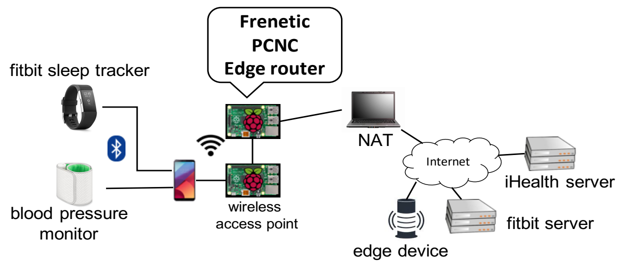


Figure 9: Real network testbed with a Fitbit Charge 3 for sleep tracking and an iHealth Feel wireless blood pressure monitor.

	Network Packets		Type	Initial	Reconfig.
No.	Destination	TCP port			
1	104.16.65.50	443	PUB	✓	✓
2	104.16.66.50	443	PUB	✓	✓
3	89.30.121.150	443	911	✗	✓
4	151.101.146.217	443	911	✗	✓
5	129.21.62.150	443	PUB	✓	✓

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

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 via *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.

[2]https://github.com/uvm-plaid/PCNC_CCS_2019

Match			Action
IP4Dst = 104.16.65.50	Vlan = 1000	EthType = 0x800 (ip)	Output(1)
IP4Dst = 104.16.66.50	Vlan = 1000	EthType = 0x800 (ip)	Output(1)

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.

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 `vlanID` 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:

```
(filter (not (switch = 346653522121)) or
  (switch = 346653522121 and
   ((vlanId=1000 and ip4Dst=104.16.65.50 and tcpDstPort=443) or
    (vlanId=1000 and ip4Dst=104.16.66.50 and tcpDstPort=443) or
     ipProto = 17)))
```

Emergency data (911) from the iHealth app is sent specifically to either 89.30.121.150:443 or 151.101.146.217:443, which are the Withings cloud servers, while public data (PUB) from the fitbit is sent to the fitbit servers at 104.16.65.50:443 and 104.16.66.50:443. The public

Match			Action
IP4Dst = 104.16.65.50	Vlan = 1000	EthType = 0x800 (ip)	SetField(ipDst, 129.21.62.150) Output(1) + Output(1)
IP4Dst = 104.16.66.50	Vlan = 1000	EthType = 0x800 (ip)	SetField(ipDst, 129.21.62.150) Output(1) + Output(1)
IP4Dst = 151.101.146.217	Vlan = 1001	EthType = 0x800 (ip)	Output(1)
IP4Dst = 89.30.121.150	Vlan = 1001	EthType = 0x800 (ip)	Output(1)

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.

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:

```
(filter (not (switch = 346653522121)) or
  (switch = 346653522121 and
    ((vlanId=1000 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=89.30.121.150 and tcpDstPort=443) or
     (vlanId=1001 and ip4Dst=151.101.146.217 and tcpDstPort=443) or
       ipProto = 17)))
```

Additionally, the forwarding policy is revised to duplicate packets from H_3 to H_5 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:

```
(if switch = 346653522121
 then (ip4Dst := 104.16.66.50; port := 1
       + ip4Dst := 129.21.61.113; port := 1)
 else port := 55555 (* packet drop *) )
```

where the IP address of H_5 is 129.21.61.113.

5.4.3 Results. As for the Mininet example, the case study PCNC messages for the testbed 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.[3]

[3] https://github.com/uvm-plaid/PCNC_CCS_2019

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 F_{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

[1] M. Abadi. 2003. Logic in access control. In *IEEE Symposium of Logic in Computer Science (LICS)*. 228–233. https://doi.org/10.1109/LICS.2003.1210062
[2] Martín Abadi. 2006. Access Control in a Core Calculus of Dependency. In *ACM International Conference on Functional Programming (ICFP)*. 263–273. https://doi.org/10.1145/1159803.1159839
[3] Martín Abadi. 2008. Variations in Access Control Logic. In *International Conference on Deontic Logic in Computer Science (DEON)*. 96–109. https://doi.org/10.1007/978-3-540-70525-3_9

[4] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *ACM Symposium on Principles of Programming Languages (POPL)*.

[5] Andrew W. Appel and Edward W. Felten. 1999. Proof-carrying Authentication. In *ACM Conference on Computer and Communications Security (CCS)*. 52–62. https://doi.org/10.1145/319709.319718

[6] Ljudevit Bauer. 2003. *Access Control for the Web via Proof Carrying Authorization*. Ph.D. Dissertation. Princeton University.

[7] Qiang Cao, Vamsi Thummala, Jeffrey S. Chase, Yuanjun Yao, and Bing Xie. 2017. Certificate Linking and Caching for Logical Trust. *CoRR* abs/1701.06562 (2017). http://arxiv.org/abs/1701.06562

[8] Peter Chapin, Christian Skalka, and X. Sean Wang. 2008. Authorization in Trust Management: Features and Foundations. *Comput. Surveys* 40, 3 (2008), 1–48.

[9] CloudFlare. 2019. Cloudflare. (2019). https://www.cloudflare.com/.

[10] Heather Crawford, Karen Renaud, and Tim Storer. 2013. A framework for continuous, transparent mobile device authentication. *Elsevier Computers & Security* 39 (2013), 127–136.

[11] fitbit. 2019. Fitbit Charge 3. (2019). https://www.fitbit.com/home.

[12] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: A Network Programming Language. In *ACM International Conference on Functional Programming (ICFP)*. 279–291.

[13] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. 2015. A Coalgebraic Decision Procedure for NetKAT. In *ACM Symposium on Principles of Programming Languages (POPL)*. 343–355. https://doi.org/10.1145/2676726.2677011

[14] Raspberry Pi Foundation. 2019. Raspberry Pi 3 Model B+. (2019). https://www.raspberrypi.org/.

[15] Deepak Garg. 2007. An Introduction to Proof-Carrying Authorization. (2007). https://people.mpi-sws.org/~dg/papers/intro-pca.pdf Course notes for CMU 18-739: Foundations of Security and Privacy.

[16] Deepak Garg and Martín Abadi. 2008. A Modal Deconstruction of Access Control Logics. In *Proceedings of the Theory and Practice of Software, 11th International Conference on Foundations of Software Science and Computational Structures (FOSSACS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 216–230. http://dl.acm.org/citation.cfm?id=1792803.1792819

[17] Deepak Garg and Frank Pfenning. 2012. Stateful Authorization Logic—Proof Theory and a Case Study. *Journal of Computer Security* 20, 4 (July 2012), 353–391. http://dl.acm.org/citation.cfm?id=2590602.2590605

[18] Arpit Gupta, Nick Feamster, and Laurent Vanbever. 2016. Authorizing Network Control at Software Defined Internet Exchange Points. In *Proceedings of the Symposium on SDN Research (SOSR '16)*. ACM, New York, NY, USA, Article 16, 6 pages. https://doi.org/10.1145/2890955.2890956

[19] Andrew K. Hirsch and Michael R. Clarkson. 2013. Belief Semantics of Authorization Logic. In *ACM Conference on Computer and Communications Security (CCS)*. 561–572. https://doi.org/10.1145/2508859.2516667

[20] iHealth. 2019. iHealth. https://ihealthlabs.com/. (2019).

[21] INRIA. 2019. OCaml. (2019). https://ocaml.org/.

[22] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *USENIX Symposium on Network Systems Design and Implementation (NSDI)*.

[23] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2013. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *USENIX Symposium on Network Systems Design and Implementation (NSDI)*.

[24] Dexter Kozen. 1997. Kleene algebra with tests. *Transactions on Programming Languages and Systems* 19, 3 (May 1997), 427–443.

[25] Diego Kreutz, Fernando M.V. Ramos, and Paulo Verissimo. 2013. Towards Secure and Dependable Software-defined Networks. In *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*. 55–60. https://doi.org/10.1145/2491185.2491199

[26] Dan Mihai, James Martucci, and Kenneth Kohler. 2004. System and method for medical device authentication. (Aug. 2004). US Patent App. 10/748,762.

[27] Takayuki Miura, Tsuyoshi Ono, Naoshi Suzuki, and Kouji Miyata. 2010. Device authentication system. (Mar. 2010). US Patent 7,681,033.

[28] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. 2013. Composing Software Defined Networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI), Lombard, IL*.

[29] George C. Necula. 1997. Proof-carrying Code. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 106–119. https://doi.org/10.1145/263699.263712

[30] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. 2015. The Design and Implementation of Open vSwitch. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 117–130.

[31] Frank Pfenning and Rowan Davies. 2001. A Judgmental Reconstruction of Modal Logic. *Mathematical. Structures in Comp. Sci.* 11, 4 (Aug. 2001), 511–540. https://doi.org/10.1017/S0960129501003322

[32] Philip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. 2012. A Security Enforcement Kernel for OpenFlow Networks. In *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networks (HotSDN)*. 121–126. https://doi.org/10.1145/2342441.2342466

[33] Yue Qiu and Maode Ma. 2016. A mutual authentication and key establishment scheme for m2m communication in 6lowpan networks. *IEEE transactions on industrial informatics* 12, 6 (2016), 2074–2085.

[34] Freddy K Santoso and Nicholas CH Vun. 2015. Securing IoT for smart home system. In *IEEE International Symposium on Consumer Electronics (ISCE)*. 1–2.

[35] Fred B. Schneider, Kevin Walsh, and Emin Gün Sirer. 2011. Nexus Authorization Logic (NAL): Design Rationale and Applications. *ACM Trans. Inf. Syst. Secur.* 14, 1, Article 8 (June 2011), 28 pages. https://doi.org/10.1145/1952982.1952990

[36] Brian Smith. 2019. ring cryptography API for Rust. (2019). https://github.com/briansmith/ring

[37] Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. 2015. A Fast Compiler for NetKAT. In *ACM International Conference on Functional Programming (ICFP)*. 328–341. https://doi.org/10.1145/2858949.2784761

[38] Richard Statman. 1979. Intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science* 9, 1 (1979), 67 – 72. https://doi.org/10.1016/0304-3975(79)90006-9

[39] Mininet Team. 2019. Mininet. (2019). http://mininet.org/.

[40] withings. 2019. Withings BPM Core. (2019). https://www.withings.com/.

A System F_{Says} DEFINITION

Section 3 presented a summary of System F_{Says}. This appendix provides a more complete definition.

The syntax of System F_{Says} is given in Figure 10. The language is an extension of System F with a *Says* monad to represent the *Says* modality in NAL_{light}, and other features to represent NAL_{light} connectives. Types τ of System F_{Says} have a tight correspondence with NAL_{light} formulas F. The System F fragment of System F_{Says} is adequate to represent implication and higher order quantification as in NAL_{light}, and the addition of sum (+) and product (×) types are adequate to represent disjunction and conjunction. The *Says* monad 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 $p \preccurlyeq q$ 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 F_{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 τ is required to have no free variables outside of S, and typing judgments $S, \Gamma \vdash e : \tau$ where Γ 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 $p \equiv q$.

p	$\in$	*prog*	$:=$	$\ldots$	*NetKAT Programs*
X	$\in$	*tvar*	$:=$	$\ldots$	*type variables*
A	$\in$	*pals*	$:=$	$\ldots$	*principles*
τ	$\in$	*type*	$::=$	$\text{unit} \mid \tau + \tau \mid \tau \times \tau \mid \tau \to \tau \mid X \mid \forall X.\tau \mid \exists X.\tau \mid$	
				$\tau \textit{ Says } \tau \mid \text{install}(\tau) \mid \tau \preccurlyeq \tau \mid p \mid A$	
x	$\in$	*var*	$:=$	$\ldots$	*term variables*
e	$\in$	*exp*	$::=$	$\text{tt} \mid$	*unit*
				$\text{inl}(e) \text{ as } \cdot + \tau \mid \text{inr}(e) \text{ as } \tau + \cdot \mid \text{case}(e)\{x.e\}\{x.e\} \mid$	*sums*
				$\langle e, e\rangle \mid \text{projl}(e) \mid \text{projr}(e) \mid$	*products*
				$x \mid \text{let } x := e \text{ in } e \mid \lambda(x : \tau).e \mid e(e) \mid$	*variables, let and functions*
				$\Lambda X.e \mid e[\tau] \mid \langle^{*}\tau, e\rangle \text{ as } \exists X.\tau \mid \text{let } \langle^{*}X, x\rangle := e \text{ in } e \mid$	*existential quantification*
				$\text{ret}^{A}(e) \mid x \leftarrow e \,;\, e \mid e \preccurlyeq e \mid p$	
v	$\in$	*val*	$::=$	$\text{tt} \mid$	*unit*
				$\text{inl}(v) \text{ as } \cdot + \tau \mid \text{inr}(v) \text{ as } \tau + \cdot \mid$	*sums*
				$\langle v, v\rangle \mid$	*products*
				$\lambda(x : \tau).e \mid$	
				$\Lambda X.e \mid \langle^{*}\tau, v\rangle \text{ as } \exists X.\tau \mid$	*existential quantification*
				$\text{ret}^{A}(v) \mid p \preccurlyeq p$	*program equivalence*
Γ	$\in$	*tenv*	$:=$	$\textit{var} \rightharpoonup \textit{type}$	*type environment*
S	$\in$	*scope*	$:=$	$\wp(\textit{tvar})$	*type scope*

Figure 10: System F$_{Says}$ Syntax

Scope Well-formedness $\boxed{S \vdash \tau}$

Unit
$$S \vdash \text{unit}$$

Sum
$$\frac{S \vdash \tau_1 \qquad S \vdash \tau_2}{S \vdash \tau_1 + \tau_2}$$

Prod
$$\frac{S \vdash \tau_1 \qquad S \vdash \tau_2}{S \vdash \tau_1 \times \tau_2}$$

Fun
$$\frac{S \vdash \tau_1 \qquad S \vdash \tau_2}{S \vdash \tau_1 \to \tau_2}$$

TVar
$$\frac{X \in S}{S \vdash X}$$

Forall
$$\frac{S \cup \{X\} \vdash \tau}{S \vdash \forall X.\tau}$$

Exists
$$\frac{S \cup \{X\} \vdash \tau}{S \vdash \exists X.\tau}$$

Says
$$\frac{S \vdash \tau}{S \vdash A \textit{ Says } \tau}$$

Equiv
$$S \vdash p \preccurlyeq p'$$

Type Well-formedness $\boxed{S, \Gamma \vdash e : \tau}$

TT
$$S, \Gamma \vdash \text{tt} : \text{unit}$$

Inl
$$\frac{S \vdash \tau_2 \qquad S, \Gamma \vdash e : \tau_1}{S, \Gamma \vdash \text{inl}(e) \text{ as } \cdot + \tau_2 : \tau_1 + \tau_2}$$

Inr
$$\frac{S \vdash \tau_1 \qquad S, \Gamma \vdash e : \tau_2}{S, \Gamma \vdash \text{inr}(e) \text{ as } \tau_1 + \cdot : \tau_1 + \tau_2}$$

Case
$$\frac{S, \Gamma \vdash e_1 : \tau_1 + \tau_2 \qquad S, \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau \qquad S, \Gamma[y \mapsto \tau_2] \vdash e_3 : \tau}{S, \Gamma \vdash \text{case}(e_1)\{x.e_2\}\{y.e_3\} : \tau}$$

Pair
$$\frac{S, \Gamma \vdash e_1 : \tau_1 \qquad S, \Gamma \vdash e_2 : \tau_2}{S, \Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2}$$

Projl
$$\frac{S, \Gamma \vdash e : \tau_1 \times \tau_2}{S, \Gamma \vdash \text{projl}(e) : \tau_1}$$

Projr
$$\frac{S, \Gamma \vdash e : \tau_1 \times \tau_2}{S, \Gamma \vdash \text{projr}(e) : \tau_2}$$

Var
$$\frac{\Gamma(x) = \tau}{S, \Gamma \vdash x : \tau}$$

Let
$$\frac{S, \Gamma \vdash e_1 : \tau_1 \qquad S, \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{S, \Gamma \vdash \text{let } x := e_1 \text{ in } e_2 : \tau_2}$$

Lambda
$$\frac{S \vdash \tau_1 \qquad S, \Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{S, \Gamma \vdash \lambda(x : \tau_1).e : \tau_1 \to \tau_2}$$

Apply
$$\frac{S, \Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad S, \Gamma \vdash e_2 : \tau_1}{S, \Gamma \vdash e_1(e_2) : \tau_2}$$

TypeLambda
$$\frac{S \cup \{X\}, \Gamma \vdash e : \tau}{S, \Gamma \vdash \Lambda X.e : \forall X.\tau}$$

TypeApply
$$\frac{S \vdash \tau' \qquad S, \Gamma \vdash e : \forall X.\tau}{S, \Gamma \vdash e[\tau'] : [\tau'/X]\tau}$$

Pack
$$\frac{S, \Gamma \vdash e : [\tau'/X]\tau}{S, \Gamma \vdash \langle^{*}\tau', e\rangle \text{ as } \exists X.\tau : \exists X.\tau}$$

Unpack
$$\frac{S \setminus \{X\} \vdash \tau_2 \qquad S, \Gamma \vdash e_1 : \exists X.\tau_1 \qquad S \cup \{X\}, \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{S, \Gamma \vdash \text{let } \langle^{*}X, x\rangle := e_1 \text{ in } e_2 : \tau_2}$$

Ret
$$\frac{S, \Gamma \vdash e : \tau}{S, \Gamma \vdash \text{ret}^{A}(e) : A \textit{ Says } \tau}$$

Bind
$$\frac{S, \Gamma \vdash e_1 : A \textit{ Says } \tau_1 \qquad S, \Gamma[x \mapsto \tau_1] \vdash e_2 : A \textit{ Says } \tau_2}{S, \Gamma \vdash x \leftarrow e_1 \,;\, e_2 : A \textit{ Says } \tau_2}$$

Contains
$$\frac{q \equiv p + q}{S, \Gamma \vdash p \preccurlyeq q : p \preccurlyeq q}$$

Figure 11: System F$_{Says}$ Type Derivation Rules.