Proofs as Cryptography: a new interpretation of the Curry-Howard isomorphism for software certificates

Amrit Kumar, Pierre-Alain Fouque, Thomas Genet, Mehdi Tibouchi

Proofs as Cryptography: a new interpretation of the Curry-Howard isomorphism for software certificates

K. Amrit [1], P.-A. Fouque [2], T. Genet [3] and M. Tibouchi [4]

Abstract

The objective of the study is to provide a way to delegate a proof of a property to a possibly untrusted agent and have a small certificate guaranteeing that the proof has been done by this (untrusted) agent. The key principle is to see a property as an encryption key and its proof as the related decryption key. The protocol then only consists of sending a nonce ciphered by the property. If the untrusted agent can prove the property then he has the corresponding proof term (λ-term) and is thus able to decrypt the nonce in clear. By sending it back, he proves that the property has been proven without showing the proof. Expected benefits include small certificates to be exchanged and the zero-knowledge proof schema which allows the proof term to remain secret. External agents can only check whether a proof exists without having any information about it. It can be of interest if the proof contains some critical information about the code structure for instance.

1. École polytechnique, Paris
2. École normale supérieure, Paris
3. Université de Rennes 1, IRISA, Rennes
4. NTT Secure Platform Laboratories, Tokyo

Contents

1 Introduction

The objective of this study is to provide a mechanism by which a host system can determine with certainty that it is safe to execute a program supplied by a source. For this to be possible, the code producer might need to convince the consumer that the source code or the program satisfies certain required properties. These properties can very well be the requirement that the program preserves certain invariants. One way to achieve the goal is to make use of Proof-Carrying Code technique aka PCC [Nec97]: the code producer supplies with the code a *safety proof* that attests to the code's adherence to a previously defined *safety policy*. The code consumer can then easily and quickly validate the proof. This technique doesn't rely on any cryptographic system.

An alternative way to achieve the goal could be to provide a way to delegate the proof of a property to a possibly untrusted agent and have a small certificate guaranteeing that the proof has been done by the agent. This technique is useful in cases where no predefined *safety policy* is required and the code producer doesn't have to prove the property on his own and hence speeds up the process at the code producer's end. [5] Being in the same setting as the PCC, the code producer and the code consumer could agree on a predefined policy (property) and the code producer after writing the program proves the property but it doesn't send it along with the native source code. Another very important aspect of this mechanism is that it provides a new interpretation of the Curry-Howard isomorphism for software certificates. In both the mechanisms we rely on the analogy between proofs (safety proofs) and types. The analogy carries over to proof validation and type checking.

This technique relies on cryptographic mechanisms and implements a zero-knowledge proof of the fact that the proof in the hand of the prover is a correct proof of the property in question. The property to be proven is seen as an encryption key while the proof-term is the corresponding secret key. Compared to PCC technique which eventually is more of a code consumer friendly mechanism, the one that we have studied is more of a code producer friendly technique as he doesn't need to send the proof which could be of enormous size.

This internship report starts with an introduction to propositional logic and a discussion on Curry-Howard isomorphism and then it makes a detour from Proof Carrying Code technique to our technique as an improved alternative to PCC. As a proof is done in the framework of typed **SKI** combinatorial calculus which is different from the one considered in the discussion of PCC, a complete section is devoted on its presentation. Last but not least, a considerable amount of time was spent searching the appropriate cryptographic system and hence a treatment of Functional encryption (initially thought to be useful) is left in appendix.

2 Propositional logic

In mathematical logic, a propositional calculus or logic (also called sentential calculus or sentential logic) is a formal system in which formulas of a formal language may be interpreted as representing propositions. A system of inference rules and axioms allows certain formulas to be derived, called theorems; which may be interpreted as `true` propositions. The series of formulas which is constructed within such a system is called a derivation and the last formula of the series is a theorem, whose derivation may be interpreted as a proof of the truth of the proposition represented by the theorem. Truth-functional propositional logic is a propositional logic whose interpretation limits the truth values of its propositions to two, usually `true` and `false`. Truth-functional propositional logic and systems isomorphic to it are considered to be zeroth-order logic.

5. We suppose that the producer and the consumer have no prior communication and the consumer upon receiving a program (written by a certain producer) wants to verify the validity of a certain property on the program.

2.1 Background

Every logic comprises a (formal) language for making statements about objects and reasoning about properties of these objects. This view of logic is very general and actually we restrict our attention to mathematical objects, programs, and data structures in particular. Statements in logical language are constructed according to a predefined set of formation rules (depending on the language) called *syntax rules*.

A logical language can be used in different ways. For instance, a language can be used as a *deduction system* (or a proof system); that is, to construct proofs or refutations. This use of a logical language is called *proof theory*. In this case, a set of facts called axioms and a set of deduction rules (inference rules) are given, and the object is to determine which facts follow from the axioms and the rules of inference. While using logic as a proof system, one is not concerned with the meaning of the statements that are manipulated, but with the arrangement of these statements, and specially, whether proofs or refutations can be constructed. In this sense, statements in the language are viewed as cold facts and the manipulations involved are purely mechanical, to the point that they could be carried out by a computer. This does not mean that finding a proof for a statement does not require creativity, but that the interpretation of the statements is irrelevant.

However, the statements expressed in a logical language often have an intended meaning. The second use of a formal language is for expressing statements that receive a meaning when they are given what is called an interpretation. In this case, the language of logic is used to formalize properties of structures, and determine when a statement is `true` of a structure. This use of logical language is called *model theory*.

One of the interesting aspects of model theory is that it forces us to have a precise and rigorous definition of the concept of truth in a structure. Depending on the interpretation that one has in mind, truth may have quite a different meaning. For instance, whether a statement is `true` or `false` may depend on parameters. A statement `true` under all interpretations of the parameters is said to be valid. A useful (and quite reasonable) mathematical assumption is that the truth of a statement can be obtained from the truth (or falsity) of its parts (sub-statements). From a technical point of view, this means that the truth of a statement is defined by recursion on the syntactical structure of the statement.

The two aspects of logic described above are actually not independent, and it is the interaction between the model theory and proof theory that makes logic an interesting and effective tool. One might say that model theory and proof theory form a couple in which the individuals complement each other. To summarize, a logical language has a certain *syntax*, and the meaning or *semantics* of statements expressed in this language is given by an interpretation in a structure. Given a logical language and its semantics, one usually has one or more *proof systems* for this logical system.

A proof system is acceptable only if every provable formula is indeed valid. In this case, we say that the proof system is *sound*. Then, one tries to prove that the proof system is *complete*. A proof system is complete if every valid formula is provable. Depending on the complexity of the semantics of a given logic, it is not always possible to find a complete proof system for that logic. This is the case, for instance, for second-order logic. However, there are complete proof systems for propositional logic and first-order logic. In the first-order case, this only means that a procedure can be found such that, if the input formula is valid, the procedure will halt and produce a proof. But this does not provide a decision procedure for validity. Indeed, as a consequence of a theorem of Church, there is no procedure that will halt for every input formula and decide whether or not a formula is valid.

There are many ways of proving the completeness of a proof system. Oddly, most proofs establishing completeness only show that if a formula A is valid, then there *exists* a proof of A. However, such arguments do not actually yield a method for *constructing* a proof of A (in the formal system). Only the existence of a proof is shown.

Propositional logic is the system of logic with the simplest semantics. Yet, many of the concepts and techniques used for studying propositional logic generalize to first-order logic.

2.2 Generic Description

In general terms, a calculus is a formal system that consists of a set of syntactic expressions (well-formed formulas), a distinguished subset of these expressions (axioms), plus a set of formal rules that define a specific

binary relation, intended to be interpreted as logical equivalence on the space of expressions. A propositional calculus is a formal system $\mathcal{L} = \mathcal{L}(\mathcal{A}, \Omega, \mathcal{Z}, \mathcal{I})$ where :

- The set of alphabet $\mathcal{A}$ is a finite set of elements called propositional symbols or propositional variables. Syntactically speaking, these are the most basic elements of the formal language $\mathcal{L}$ otherwise referred to as atomic formulas or terminal elements.
- Ω is the set of a finite number of logical connectives. The set Ω is partitioned into disjoint subsets as :

$$\Omega = \Omega_0 \cup \Omega_1 \ldots \cup \Omega_j \ldots \cup \Omega_n$$

  The set Ω_j is the set of all operators of arity j. In the more familiar propositional calculi Ω is typically partitioned as follows:

$$\Omega_0 = \{\bot, \top\}$$
$$\Omega_1 = \{\neg\}$$
$$\Omega_2 = \{\vee, \wedge, \rightarrow\}$$

- The set $\mathcal{Z}$ is a finite set of transformation rules that are called inference rules when they acquire logical applications.
- The set $\mathcal{I}$ is the set of axioms in this language.

2.2.1 A Simplified Axiom System

We consider a propositional calculus $\mathcal{L} = \mathcal{L}(\mathcal{A}, \Omega, \mathcal{Z}, \mathcal{I})$ where :

- We consider the set $\mathcal{A}$ to be large enough that would suffice the needs of our discussion. For example $\mathcal{A} = \{p, q, r, s, t, u, v\}$.
- We take : $\Omega_1 = \{\neg\}$ and $\Omega_2 = \{\rightarrow\}$.
- The set $\mathcal{Z}$ is taken to be singleton, the rule being : if p and $p \rightarrow q$ are `true` then we can infer that q is also `true`.
- The set $\mathcal{I}$ is the set of axioms in this language and consists precisely of the following ones:
  - $(p \rightarrow (q \rightarrow p))$
  - $((p \rightarrow (q \rightarrow r)) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow r)))$
  - $((\neg p \rightarrow \neg q) \rightarrow (q \rightarrow p))$

2.2.2 Natural Deduction System

We consider a propositional calculus $\mathcal{L} = \mathcal{L}(\mathcal{A}, \Omega, \mathcal{Z}, \mathcal{I})$ where :

- The set $\mathcal{A}$ is supposed to be large enough that would suffice the needs. For example $\mathcal{A} = \{p, q, r, s, t, u, v\}$.
- We consider : $\Omega_0 = \{\bot, \top\}$, $\Omega_1 = \{\neg\}$ and $\Omega_2 = \{\vee, \wedge, \rightarrow\}$.
- The set $\mathcal{Z}$ is a defined in Figure 1, the transformation rules are intended to be interpreted as the inference rules of so called *natural deduction system*.
- The system presented here has just one axiom that says $p \rightarrow p$.

Here we use the sequent notation $A_1, A_2, \ldots, A_n \vdash B$ to represent judgements in natural deduction. The standard semantics of a judgement in natural deduction is that it asserts that whenever $A_1, A_2 \ldots, A_n$ are all `true`, B will also be `true`.

2.3 Classical, Intuitionistic & Minimal Logic

The logics presented above are classical logic and in general they are characterized by a number of equivalent axioms :

Proof by contradiction

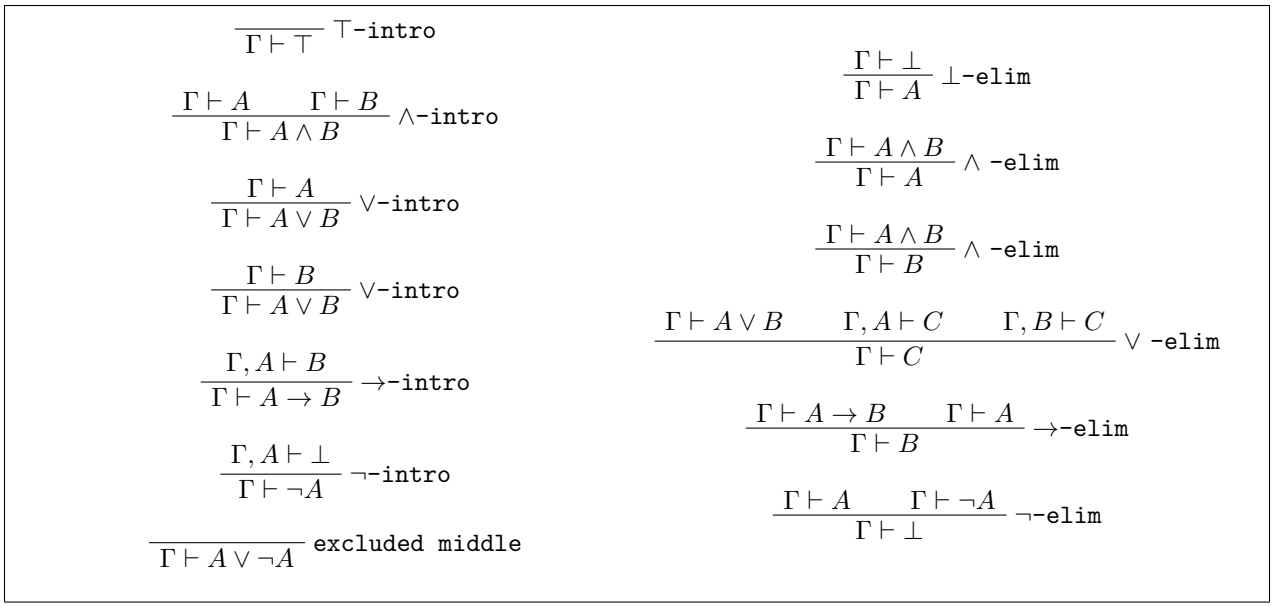


Figure 1: Inference rules of natural deduction

Law of the Excluded Middle

Double-Negation Elimination

Pierce's Law $((A \to B) \to A) \to A$

We can define two other logical frameworks depending on the presence or absence of certain rules. To be precise we are interested in Minimal and Intuitionistic logic. Defined in [Mer83] and developed by Ingebrigt Johansson, minimal logic is a sub-logic of intuitionistic logic which means that set of provable propositions in minimal logic is a proper subset of the corresponding set in intuitionistic logic.

In all the three logics we have two rules corresponding to negation:

- **Elimination of negation** : If we can prove a proposition A and its negation $\neg A$, then we have a contradiction noted $\bot$.
- **Introduction of negation** : If a proposition A leads to a contradiction then $\neg A$ is valid. The rule can even be formulated as the definition of negation : $\neg A := A \to \bot$.

The three logics differ on the consequence drawn by a contradiction.

- Classical logic uses *reductio ad absurdum* and deduces from $\neg A \to \bot$ that A is valid. This is in fact the elimination rule for double negation because $\neg A \to \bot$ is the synonym of $\neg\neg A$.
- Intuitionistic logic deduces any proposition from a contradiction: $\bot \to B$ which is the rule *ex falso quodlibet* aka the principle of explosion.
- Minimal logic treats $\bot$ as any other proposition and hence has no particular significance.

2.4 Examples from Minimal Logic

Example 1: $(\neg A \wedge \neg B \leftrightarrow \neg(A \vee B))$

We suppose that we have $\neg A \wedge \neg B$ and prove that $\neg(A \vee B)$ i.e. the hypothesis $A \vee B$ leads to a contradiction. We have two cases : if A is valid then it is in contradiction with the hypothesis $\neg A$ similarly for B. So, in any case, we have a contradiction.

Conversely, we suppose that we have $\neg(A \vee B)$ and we prove $\neg A$ i.e. A leads to contradiction. But if A is valid then $A \vee B$ is valid which contradicts the hypothesis. Similarly for B. However, we only have

$(\neg A \vee \neg B \leftrightarrow \neg(A \wedge B))$. The converse is valid only in classical logic.

Example 2: $A \rightarrow \neg\neg A$ We suppose A then the supplementary hypothesis $\neg A$ results in a contradiction. Hence we have the result. The converse is not valid in minimal logic and neither in intuitionistic logic. However we have $\neg\neg\neg A \rightarrow \neg A$.

Example 3: We can show that in minimal logic $\neg\neg(A \rightarrow B) \rightarrow (\neg\neg A \rightarrow \neg\neg B)$. But the converse is valid both in intuitionistic and classical logic but not in minimal logic.

Example 4: For the contra-positive argument : we can show that in minimal logic we have $(A \rightarrow B) \rightarrow (\neg B \rightarrow \neg A)$, $(A \rightarrow \neg B) \rightarrow (B \rightarrow \neg A)$ and $(\neg A \rightarrow \neg B) \rightarrow (B \rightarrow \neg\neg A)$ but we don't have $(\neg A \rightarrow \neg B) \rightarrow (B \rightarrow A)$ which is a variant of *reductio ad absurdum*.

3 Curry-Howard Isomorphism

In programming language theory and proof theory, the Curry-Howard isomorphism is the direct relationship between computer programs and proofs. It is a generalization of a syntactic analogy between systems of formal logic and computational calculi. The Curry-Howard isomorphism is the observation that two families of formalisms—namely, the proof systems on one hand, and the models of computations on the other—are in fact structurally the same kind of objects. In other words, *a proof is a program, the formula it proves is a type for the program.*

In its more general formulation, the Curry-Howard isomorphism is a correspondence between formal proof calculi and type systems for models of computations. In particular, it splits into two correspondences. One at the level of formulas and types that is independent of which particular proof system or model of computation is considered, and one at the level of proofs and programs which, this time, is specific to the particular choice of proof system and model of computation considered.

At the level of formulas and types, the correspondence says that implication behaves as a `function type`, conjunction as a `product type` (this may be called a tuple, a struct, a list, or some other term depending on the language), disjunction as a `sum type` (this may be called a union), a `false` formula as the `empty type` and a `true` formula as the `singleton type` (whose sole member is the null object). Quantifiers correspond to dependent function space or products (as appropriate). The following table summarizes the above discussion:

Logic side	Programming side
universal quantification	generalized function space($\prod$) `type`
existential quantification	generalized cartesian product ($\sum$) `type`
implication	`function type`
conjunction	`product type`
disjunction	`sum type`
`true` formula	`unit type`
`false` formula	`empty type`

3.1 Correspondence between Hilbert-style Deduction Systems and Combinatory Logic

According to Curry: the simplest types for the basic combinators **K** and **S** of combinatory logic surprisingly correspond to the respective axiom schemes $\alpha \rightarrow (\beta \rightarrow \alpha)$ and $(\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma))$ noted ϕ in Figure 2 used in Hilbert-style deduction systems. A complete section is devoted on this calculus as it is used to express formulas or properties of a program in our technique. We obtain a similar correspondence where both the columns are in one-to-one correspondence.

Hilbert-style implication logic	**Simply typed combinatory logic**
$\dfrac{\alpha \in \Gamma}{\Gamma \vdash \alpha}$ `assumption`	$\dfrac{x : \alpha \in \Gamma}{\Gamma \vdash x : \alpha}$
$\dfrac{}{\Gamma \vdash \alpha \to (\beta \to \alpha)}$ $\texttt{axiom}_K$	$\dfrac{}{\Gamma \vdash K : \alpha \to (\beta \to \alpha)}$
$\dfrac{}{\Gamma \vdash \phi}$ $\texttt{axiom}_S$	$\dfrac{}{\Gamma \vdash S : \phi}$
$\dfrac{\Gamma \vdash \alpha \to \beta \qquad \Gamma \vdash \alpha}{\Gamma \vdash \beta}$ `modus ponens`	$\dfrac{\Gamma \vdash E_1 : \alpha \to \beta \qquad \Gamma \vdash E_2 : \alpha}{\Gamma \vdash E_1 E_2 : \beta}$

Figure 2: Typed combinatory logic

Seen at a more abstract level, the correspondence can be restated as shown in the following table.

Logic side	Programming side
`assumption`	variable
`axioms`	combinators
`modus ponens`	application

3.2 Proof as program : applications

As seen above proofs act as programs and that a proof π of a property ϕ is equivalent to saying that the type of the λ-term associated with the proof is ϕ. This equivalence is useful in proof validation where one has to validate a proof π of a property (or a formula) ϕ.

This is useful in applications where users need to be convinced that a free software developed by an untrusted agent satisfies a certain property. This trust problem is specific for free software because : they are not always developed by well-known companies (that users may trust), such software developments rely on a large community of authors for development/ proofs or these developers cannot afford code signing by a certifying authority.

Now the question that stands is how exactly a user can be convinced that a software (or a program) is safe to use without relying on reputation or any certifying authority. There are several possibilities :

- using the Proof-Carrying Code framework [Nec97]
- probabilistic checking of the proof [AS92]
- checking that the poof has been constructed using zero-knowledge protocols and Curry-Howard isomorphism.

4 SKI Combinatorial Calculus

The logical framework in which we are working is not the same as the one used in the PCC [Nec97]. Our framework is that of a propositional calculus with a simplified axiomatic system. In the literature this logic is called **SKI(SK)** Combinatorial Calculus. The following sections present the features of this logical framework.

4.1 Presentation

We use the definition presented in subsection 2.2 to define **SKI** calculus as an example of propositional logic and in fact as the implicational fragment of any intuitionistic logic.

4.1.1 Definition

SKI calculus is obtained for the following instance of the sets:
- $\mathcal{A} = \{p, q, r, s, t, u, v\}$ or any such finite set. Another way of defining the alphabet would be to take $\mathcal{A} = \{S, K, I, (,)\}$
- $\Omega_2 = \{\rightarrow\}$.
- The rule of *modus ponens* : if p and $p \rightarrow q$ are `true` then we can infer that q is also `true`.
- The set $\mathcal{I}$ being the set containing:

  S: $(p \rightarrow (q \rightarrow p))$

  K: $((p \rightarrow (q \rightarrow r)) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow r)))$

  I: $(p \rightarrow p)$

4.1.2 Terms and Derivations

The set $\mathcal{T}$ of terms is defined recursively as follows:

1. **S**,**K**,**I** are terms.
2. If τ_1 and τ_2 are terms, then $(\tau_1\tau_2)$ is also a term.
3. Nothing is a term unless required to be by the rules 1 and 2.

A derivation is a finite sequence of terms satisfying the following rules:

1. If Δ is a derivation ending in the term $\alpha((K\beta)\gamma)\zeta$, then Δ followed by $\alpha\beta\zeta$ is also a derivation.
2. If Δ is a derivation ending in the term $\alpha(I\beta)\zeta$, then Δ followed by $\alpha\beta\zeta$ is also a derivation.
3. If Δ is a derivation ending in the term $\alpha(((S\beta)\gamma)\delta)\zeta$, then Δ followed by $\alpha((\beta\delta)(\gamma\delta))$ is also a derivation.

4.2 Properties

The ceremony above captures much of the conventional style in which logicians present the combinator calculus. But the conventional ceremony describing the combinator calculus does not match the natural binary structure of the formal system. It is much more natural to understand the calculus as operating on binary tree-structured terms with the symbols **S**,**K**, **I** at the leaves. The parentheses are in some sense only there to indicate the tree structure, and shouldn't be regarded as part of the abstract alphabet.

Informally, and using programming language jargon, a tree (xy) can be thought of as a "function" x applied to an "argument" y. When "evaluated", the tree "returns a value" , i.e. transforms into another tree. Of course, all three of the "function", the "argument" and the "value" are either combinators, or binary trees, and if they are binary trees they too may be thought of as functions whenever the need arises. The evaluation operation is defined as follows :

I returns its argument :

$$Ix = x$$

K when applied to any argument x, yields a one-argument constant function Kx, which, when applied to any argument, returns x.

$$Kxy = x$$

S is a substitution operator. It takes three arguments and then returns the first argument applied to the third, which is then applied to the result of the second argument applied to the third. More clearly :

$$Sxyz = (xz)(yz)$$

4.2.1 Computations

Example 1: SKSK evaluates to **KK(SK)** by the **S**-rule. Then if we evaluate **KK(SK)**, we get **K** by the **K-rule**. As no further rule can be applied, the computation halts there.

Example 2: For all trees α and β, $\mathbf{SK}\alpha\beta$ will always evaluate to β in two steps, $\mathbf{K}\beta(\alpha\beta) = \beta$, so the ultimate result of evaluating $\mathbf{SK}\alpha\beta$ will always equal the result of evaluating β. We say that $\mathbf{SK}\alpha$ and **I** are "functionally" equivalent because they always yield the same result when applied to any β.

Example 3: Self-application SII is an expression that takes an argument and applies that argument to itself:

$$SII\alpha = I\alpha(I\alpha) = \alpha\alpha$$

Example 4: Recursion We can write a function that applies something to the self application of something else:

$$(S(K\alpha)(SII))\beta = K\alpha\beta(SII\beta) = \alpha(SII\beta) = \alpha(\beta\beta)$$

This function can be used to achieve recursion. If β is the function that applies to the self application of something else, then self-applying β performs α recursively on $\beta\beta$. More clearly, if $\beta = S(K\alpha)(SII)$ then :

$$SII\beta = \beta\beta = \alpha(\beta\beta) = \alpha(\alpha(\beta\beta)) = \dots$$

Example 5: Reverser S(K(SI))K reverses the terms in $\alpha\beta$:

$$S(K(SI))K\alpha\beta \to K(SI)\alpha(K\alpha)\beta \to SI(K\alpha)\beta \to I\beta(K\alpha\beta) \to I\beta\alpha \to \beta\alpha$$

Using example 2 it can be shown that **SKI** calculus is not the minimum system that can fully perform the computations of lambda calculus, as all occurrences of **I** in any expression can be replaced by **SKK** or **SKS** or **SK** and the resulting expression will yield the same result. So, **I** is merely syntactic sugar.

4.2.2 Universal Qualities of SKI

The combinator calculus was designed precisely to be *universal* in the sense that it can accomplish every conceivable rearrangement of sub-terms just by means of applying terms to them. That is, given a rule for rearranging n symbols into the shape of a term (allowing copying and deleting of individual symbols), there is a term that can be applied to each choice of n symbols so that several derivation steps will accomplish that rearrangement. The examples of **SII** as a repeater and **S(K(SI))K** as a reverser suggest how this works. This particular quality of a formal system is called *combinatory completeness*. Every formal system that contains something acting like **S** and something like **K** is combinatorialy complete.

Rearrangement arise in formal systems whenever we substitute symbols with variables. The combinator calculus was designed specifically to show that substitution for variables can be reduced to more primitive looking operations.

By accident, the combinator calculus turns out to be universal in a much more powerful sense than combinatory completeness. The combinator calculus is a *universal programming system*—its derivation can accomplish everything than can be accomplished by computation. That is, terms can be understood as programs, and every program that we can write in every programming language can be written also as a term in the combinator calculus. Since formal systems are the same thing as computing systems, every formal system can be described as an interpretation of the terms in the combinator calculus. In short, the combinator calculus is *Turing complete*. Last but not least, **SKI** calculus generates the implication fragment of intuitionistic logic.

As seen above the combinators **S**, **K**, **I** are functions and hence as the lambda calculus terms they are equivalent to :

– **S**:= $\lambda x.\lambda y.\lambda z.(xz)(yz)$
– **K**:= $\lambda x.\lambda y.x$
– **I** := $\lambda x.x$

Using the typing rules defined in Figure 2 we can talk about typed **SKI** calculus. We observe that not all the terms in the calculus are typable. One of the example is the term in the example 3 of 4.2.1. From now on, **SKI** calculus refers to the typed **SKI** calculus.

5 Proofs as Cryptography

This section presents our alternative to the PCC technique where we use cryptographic techniques to implement a zero-knowledge proof of **the validity of a proof of a given property**. We should note that we have not been able to make the technique completely functional. Part of this section presents the goal and the steps that we have taken to tackle the different problems that we have faced and highlights some of these which remain unresolved.

There are several ways to present the communication model between the user and the developer, hitherto called consumer and producer respectively. This change in nomenclature is in consonance with the different possible models. One of the possible model implementing this technique could be: instead of proving a property of a program on the developer's side, we delegate the proof to an untrusted agent and we receive a small certificate guaranteeing that the proof has been done by this agent. We can even generalize this mechanism where the communication is done on a broadcast communication network, where terminals on the other end are provers and the user is the verifier. Upon receiving a program, the user might need to verify if a property holds for the program and so he gets in touch with provers (in broadcast network) and verifies if any of the provers has a valid proof.

We can even have a model where the user and the developer have no prior communication on the property and the user upon receiving a program (written by a certain developer) wants to verify the validity of a certain property on the program. He then looks for a zero-knowledge proof of the validity of the proof.

Another possible model could follow the same route as the PCC technique where the user and the developer agree on a property and then after writing the program the developer proves the property on the program and then sends the program (not the proof) to the user. The user then verifies the validity of the proof and ultimately forwards it to the waiting process. The following subsection discusses the latter one which remains in tune with the PCC except the fact that the proof term (λ-term) corresponding to the proof is not sent along with the program.

5.1 Communication Protocol

In fact, the protocol can be conceived in the following way and a schematic representation is given by Figure 3. The user and the developer agree on a property ϕ that a program must satisfy. The developer writes a program P, proves the property ϕ on P and sends it to the user. The user then uses cryptographic means to verify the validity of the proof. The property gives the encryption key while a proof term π (λ-term corresponding to the proof) gives a secret key. He then encrypts a nonce using the property and sends it to the developer, if the developer has a valid proof, he has a decryption algorithm (eventually depending on the λ-term corresponding to the proof) which can decrypt the nonce and the developer then sends the decrypted nonce to the user. The user verifies the exactitude of the nonce sent and the nonce received. This zero-knowledge proof conception makes our principle far more simpler and efficient compared to PCC.

5.2 Encryption

The predefined property ϕ is expressed in the (typed) **SKI** combinatorial calculus discussed in section 4. We give a recursive definition to the way a nonce $\{n\}$ is encrypted under a property (a formula) ϕ. For that we define or initialize keys for each atomic terms like p, q, r used to define the alphabet $\mathcal{A}$, and we denote

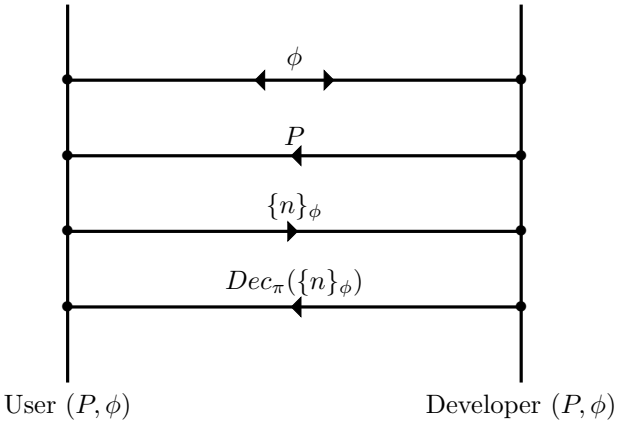


Figure 3: Communication Protocol

them by K_p, K_q, K_r. We then define the key $K_{p\to q}$ as $\{K_q\}_{K_p}$ to be read as the key K_q encrypted under the key K_p. Finally, encryption of a nonce $\{n\}$ under any formula $A \to B$ is the set $\{K_A, \{n\}_{K_B}\}$. The following examples encrypt a nonce $\{n\}$ under the axioms of the **SKI** calculus.

Example I: We encrypt a nonce $\{n\}$ under **I** as $\{n\}_{p\to p}$ which is the set $K_p, \{n\}_{K_p}$.

Example S: $\{n\}$ under **S** is $\{n\}_{(p\to(q\to p))}$ which is the set $\{K_p, \{n\}_{q\to p}\}$ and which eventually is the set $\{K_p, K_q, \{n\}_{K_p}\}$

Example K: $\{n\}$ under **K** is $\{n\}_{((p\to(q\to r))\to((p\to q)\to(p\to r)))}$ which is:

$$\begin{aligned}\{K_{(p\to(q\to r))}, \{n\}_{(p\to q)\to(p\to r)}\} &= \{\{K_{q\to r}\}_{K_p}, K_{p\to q}, \{n\}_{p\to r}\} \\ &= \{\{\{K_r\}_{K_q}\}_{K_p}, \{K_q\}_{K_p}, K_p, \{n\}_{K_r}\}\end{aligned}$$

5.3 Provable-Decryptable Equivalence

Loosely speaking, decryption of a nonce $\{n\}$ encrypted under a formula F is actually an algorithm that (at each instant) makes a choice from the available keys and a key encrypted under this key or eventually the nonce under the chosen key. If the property is provable in the **SKI** calculus, the process terminates with the decrypted nonce. Before delving in the problem of proving the fact that having a proof is equivalent to having a decryption algorithm, we can see that the examples taken in the previous subsection proves the fact that the encryption process has the soundness property. The *Provable-Decryptable* equivalence can even be expressed by saying that one cannot decrypt a nonce encrypted under a formula that is not provable in the **SKI** calculus.

5.3.1 Decryptable implies Provable

The only operator in our logic is implication $\to$, two rules of natural deduction that correspond to introduction and elimination of $\to$ are :

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ (\to) \text{ introduction} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash A \to B}{\Gamma \vdash B}\ (\to) \text{ elimination}$$

The introduction rule corresponds to the axiom **I** while the elimination rule corresponds to the inference rule of *modus ponens* in our logic. [6] We observe that the encryption algorithm implements a recursive $\rightarrow$ elimination which is to ascend upwards in the $(\rightarrow)$ introduction rule and that at any stage of the derivation tree (corresponding to the proof) and for any sequent $A_1, A_2, \ldots, A_n \vdash B$ the encryption at this stage gives $\{K_{A_1}, K_{A_2}, \ldots, K_{A_n}, \{n\}_B\}$. Thus the antecedent correspond to the keys while the succedent gives the encryption of the nonce under the key corresponding to it. The process terminates when the succedent is an atomic term (a formula devoid of the operator $\rightarrow$).

The decryption process starts by searching the key corresponding to the atomic term under which the nonce is encrypted or any available encryption of the key. The decryption of this key might require decryption of other keys. Each decryption (of either a key or eventually the nonce) mirrors into the application of $(\rightarrow)$ elimination rule. For instance, $\{K_p, \{K_q\}_{K_p}, \{n\}_{K_q}\}$ corresponds to the encryption using the sequent $p, (p \rightarrow q) \vdash q$, which is precisely the $(\rightarrow)$ elimination. In general if K_Γ decrypts $\{n\}_{K_A}$ then $\Gamma \vdash A$. Decryptability implies the existence of a function π such that $\pi(K_\Gamma) = K_A$. The function π in fact implements a recursive application of *modus ponens* and hence implements $(\rightarrow)$ elimination rule. So, if encryption is ascending upwards (away from the root) [7] in the derivation tree, decryption is descending downwards (towards the root) in the derivation tree and hence is equivalent to building the proof tree. This proves that *decryptable* implies *provable*.

5.3.2 Provable implies Decryptable

This part of the implication is non-trivial and hence requires careful observation via examples.

Example 1 $p \rightarrow ((p \rightarrow q) \rightarrow q)$ Encryption of a nonce n under the formula gives :

$$\begin{aligned}\{n\}_{p\rightarrow((p\rightarrow q)\rightarrow q)} &= \{K_p, \{n\}_{((p\rightarrow q)\rightarrow q)}\} \\ &= \{K_p, K_{p\rightarrow q}, \{n\}_{K_q}\} \\ &= \{K_p, \{K_q\}_{K_p}, \{n\}_{K_q}\}\end{aligned}$$

Encryption termination step

$$\dfrac{\dfrac{\dfrac{\overline{p, (p \rightarrow q) \vdash p} \quad \overline{p, (p \rightarrow q) \vdash (p \rightarrow q)}}{p, (p \rightarrow q) \vdash q}}{p \vdash (p \rightarrow q) \rightarrow q}}{\vdash p \rightarrow ((p \rightarrow q) \rightarrow q)}$$

Proof 1: Proof tree

We use the available key K_p to retrieve K_q and ultimately n.
To be more formal we can use the proof-term corresponding to Proof 1 to decrypt the nonce. The proof-term corresponding to the above formula is : $\lambda x.\lambda f.fx$ and captures the idea of *typing rules*: if x is of `type` p and f is of `type` $p \rightarrow q$ then fx is of `type` q.

The corresponding decryption algorithm would be $\lambda x.\lambda f.\lambda c.c(fx)$ and works in the following way :

$$\begin{aligned}\lambda x.\lambda f.\lambda c.c(fx)\{n\}_{p\rightarrow((p\rightarrow q)\rightarrow q)} &= \lambda x.\lambda f.\lambda c.c(fx)K_p\{K_q\}_{K_p}\{n\}_{K_q} \\ &= \{n\}_{K_q}(\{K_q\}_{K_p}K_p) \\ &= \{n\}_{K_q}K_q \\ &= n\end{aligned}$$

6. Using the soundness of $\rightarrow$ wrt $\vdash$: if $\Gamma \rightarrow A$, then $\Gamma \vdash A$, and completeness of $\rightarrow$ wrt $\vdash$: if $\Gamma \vdash A$, then $\Gamma \rightarrow A$.
7. root being the formula to be proven

Example 2 $((p \to q) \to q) \to p$

$$\begin{aligned}\{n\}_{((p\to q)\to q)\to p} &= \{K_{(p\to q)\to q}, \{n\}_{K_p}\} \\ &= \{\{K_q\}_{\{\{K_q\}_{K_p}\}}, \{n\}_{K_p}\}\end{aligned}$$

As no free key is available it seems we cannot decrypt n and this is exactly what we **expect** (if we hope that provability implies decryptability) because the above formula is not provable in our logic. We can check different formulations of *contra-positive argument* :

Example 3.1 $(p \to q) \to ((q \to r) \to (p \to r))$

$$\{n\}_{(p\to q)\to((q\to r)\to(p\to r))} = \{\{K_q\}_{K_p}, \{K_r\}_{K_q}, K_p, \{n\}_{K_r}\}$$

Example 3.2 $(p \to (q \to r)) \to (q \to (p \to r))$

$$\{n\}_{(p\to(q\to r))\to(q\to(p\to r))} = \{\{\{K_r\}_{K_q}\}_{K_p}, K_q, K_p, \{n\}_{K_r}\}$$

Example 3.3 $((p \to r) \to (q \to r)) \to (q \to ((p \to r) \to r))$

$$\{n\}_{((p\to r)\to(q\to r))\to(q\to((p\to r)\to r))} = \{\{\{K_r\}_{K_q}\}_{\{\{K_r\}_{K_p}\}}, K_q, \{K_r\}_{K_p}, \{n\}_{K_r}\}$$

Example 3.4 $((p \to r) \to (q \to r)) \to (q \to p)$

$$\{n\}_{((p\to r)\to(q\to r))\to(q\to p)} = \left\{\{\{K_r\}_{K_q}\}_{\{\{K_r\}_{K_p}\}}, K_q, \{n\}_{K_p}\right\}$$

The first three decryptions are obvious and follow the same pattern as in the first example. But the fourth encryption cannot be reversed and again the reason **should** be the fact that the last formula is not provable in our logic (subsection 2.4).

Example 4 $\vdash ((p \to q) \to r) \to (q \to r)$
Encryption of a nonce n gives

$$\{n\}_{((p\to q)\to r)\to(q\to r)} = \{\{K_r\}_{\{\{K_q\}_{K_p}\}}, K_q, \{n\}_{K_r}\}$$

At the first sight the above nonce does not seem to be decryptable but here we note that the formula is provable and a derivation tree is given in Proof 2.

$$\textbf{Encryption termination step}\ \dfrac{\dfrac{\dfrac{\dfrac{}{p, q, (p \to q) \to r \vdash q}}{q, (p \to q) \to r \vdash (p \to q)} \quad \dfrac{}{q, (p \to q) \to r \vdash (p \to q) \to r}}{q, (p \to q) \to r \vdash r}}{(p \to q) \to r \vdash (q \to r)}}{\vdash ((p \to q) \to r) \to (q \to r)}$$

Proof 2: Proof tree

A careful observation would reveal that to decrypt the nonce the developer either needs the key K_p or $\{K_q\}_{K_p}$ while knowing K_q. The second case is realizable because the developer can ask the user to encrypt K_q under the key K_p (that he doesn't have). A number of questions on security might arise but it seems to be a legitimate demand on the part of developer. And if we permit such queries the decryption is automatic.

From the perspective of the proof term :

$$\lambda H.\lambda H_0.H\,(\underbrace{\lambda \underbrace{x}_{K_p}.\underbrace{H_0}_{K_q}}_{K_{p\to q}})$$

if x is of `type` p, H_0 of `type` q and H of `type` $(p \to q) \to r$, then the `type` of the initial formula is given by the λ-term given above. In terms of encryption keys, the decryption algorithm substitutes x by K_p, H_0 by K_q and so $\lambda x.H_0$ by $K_{p\to q}$. So, a decryption algorithm would indeed need the key $K_{p\to q} = \{K_q\}_{K_p}$ in order to decrypt the nonce. We observe that in the proof term H is applied to a λ-term which is not the case in the previous examples.

Example 5.1 $(((p \to r) \to r) \to ((q \to r) \to r)) \to (((p \to q) \to r) \to r)$
A nonce n encrypts to :

$$\begin{aligned}\{n\}_{(((p\to r)\to r)\to((q\to r)\to r))\to(((p\to q)\to r)\to r)} &= K_{(((p\to r)\to r)\to((q\to r)\to r))}, \{n\}_{((p\to q)\to r)\to r}\\ &= \left\{\{K_{(q\to r)\to r}\}_{K_{(p\to r)\to r}}, \{K_r\}_{K_{p\to q}}, \{n\}_{K_r}\right\}\\ &= \left\{\{\{K_r\}_{\{\{K_r\}_{K_q}\}}\}_{\{\{K_r\}_{\{\{K_r\}_{K_p}\}}\}}, \{K_r\}_{\{\{K_q\}_{K_p}\}}, \{n\}_{K_r}\right\}\end{aligned}$$

Once again formula not being provable implies that the nonce cannot be retrieved.

Example 5.2 $(((p \to q) \to r) \to r) \to (((p \to r) \to r) \to ((q \to r) \to r))$
A nonce n encrypted under the formula is given as :

$$\begin{aligned}\{n\}_{(((p\to q)\to r)\to r)\to(((p\to r)\to r)\to((q\to r)\to r))} &= \{K_{((p\to q)\to r)\to r}, \{n\}_{(((p\to r)\to r)\to((q\to r)\to r))}\}\\ &= \{K_{((p\to q)\to r)\to r}, K_{(p\to r)\to r}, \{n\}_{(q\to r)\to r}\}\\ &= \{\{K_r\}_{\{\{K_r\}_{\{\{K_q\}_{K_p}\}}\}}, \{K_r\}_{\{\{K_r\}_{K_p}\}}, \{n\}_{(q\to r)\to r}\}\\ &= \{\{K_r\}_{\{\{K_r\}_{\{\{K_q\}_{K_p}\}}\}}, \{K_r\}_{\{\{K_r\}_{K_p}\}}, \{K_r\}_{K_q}, \{n\}_{K_r}\}\end{aligned}$$

Before analyzing an encrypted nonce under this formula which is the converse of the implication in the previous example, we observe that a proof does exist and is given by the derivation tree in Proof 3.

If we have a look at the proof term

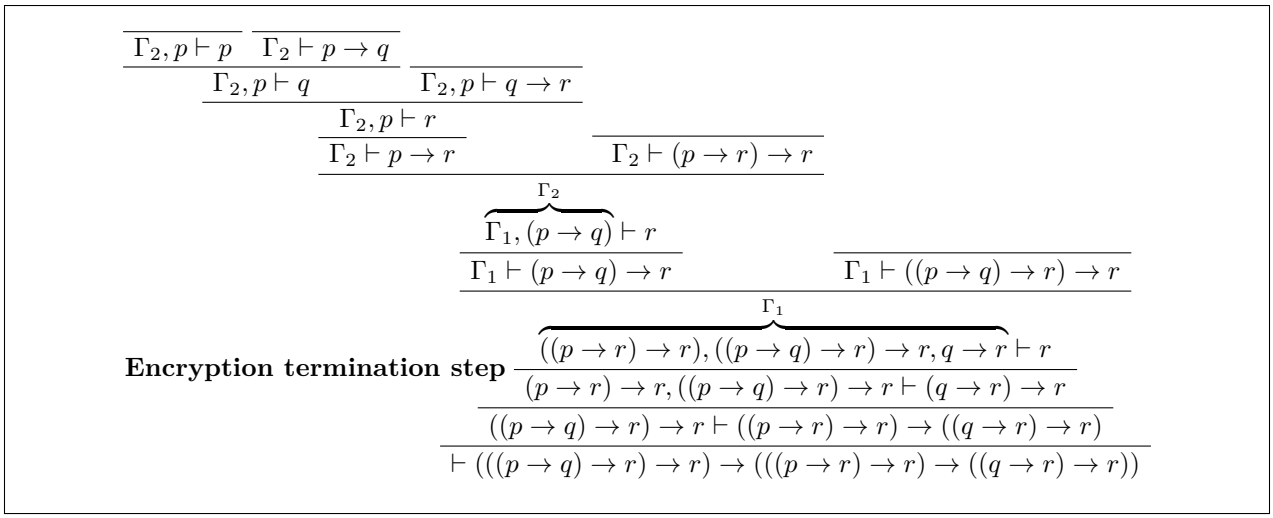
$$\lambda H.\lambda H_0.\lambda H_1.H(\lambda H_2.H_0(\lambda H_3.H_1(\underbrace{H_2}_{K_{p\to q}}\ \underbrace{H_3}_{K_p})))$$

it says that if : H is of `type` $((p \to q) \to r) \to r$, H_0 is of `type` $(p \to r) \to r$ and H_1 of `type` $q \to r$ then : the `type` of the initial formula is given by the λ-term. Here the types of H_3 and H_2 are p and $p \to q$ respectively. In the decryption algorithm, they should be substituted by keys K_p and $K_{p\to q}$ which are not available to the developer. Here again we observe that H applies on a λ-term as in Example 4.

The above example seems to be a threat to the proposition *Provability* implies *Decryptability* and remains unresolved.

Encryption termination step $\dfrac{\dfrac{\overline{p, q \vdash p}}{q \vdash (p \to p)}}{\vdash q \to (p \to p)}\ \{n\}_{q\to(p\to p)} = \{K_q, K_p, \{n\}_{K_p}\}$

Proof 4: Proof tree



Proof 3: Proof tree

A quick observation of the encryption process leads us to remark that the encryption algorithm is precisely moving upwards in the proof-tree and the algorithm terminates when the succedent at the indicated step becomes an atomic term which is $\bot$ in Proof 3, r in Proof 2 and q in Proof 1. So, encryption is not exactly building the proof itself though in some cases, they might precisely be the same; for example this happens for the property $q \to (p \to p)$, Proof 4. In the worst case, the size of the encrypted nonce can be linear in the size of the proof.

5.4 Remarks

We consider the two deduction rules (and the only ones in fact) taken from sequent calculus that act on implications.

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

This rule says that if one can decrypt $\{n\}_{A\to B}$ using the key K_Γ then one is able to decrypt $\{n\}_{K_B}$ if one has the keys K_A and K_Γ and vice-versa. This is true because by definition $\{n\}_{A\to B} = \{\{n\}_{K_B}, K_A\}$ and the nonce can be decrypted iff K_Γ and K_A yield K_B and this suffices to decrypt $\{n\}_{K_B}$ as well.

$$\frac{\Gamma \vdash A, C \qquad \Gamma, B \vdash C}{\Gamma, A \to B \vdash C}$$

In the same tune, the above rule says if keys K_Γ and $K_{A\to B} = \{K_B\}_{K_A}$ can decrypt $\{n\}_{K_C}$ then K_Γ must be able to decrypt $\{n\}_{K_A}$ and K_Γ, K_B must be able to decrypt $\{n\}_{K_C}$ and vice-versa. To be precise K_Γ must yield K_A (neglecting the trivial case where it yields K_C). For this we might suppose to have a function f that serve this purpose.

If we have some way to define such a function then the problem would be solved and then encryption under a formula is equivalent to applying the first rule and terminate at the point where the succedent is an atomic term. The decryption would be to apply the second rule or the first one to build the complete tree if possible.

6 Applications

As suggested in the paper [Nec97], PCC technique can be used to implement "safe network packet filters". It remains in the experimental stage. Many modern operating systems provide a facility for allowing application programs to receive packets directly from the network device. Typically, an application is not interested in receiving every packet from the network, but only a small fraction that exhibit a specific property (e.g. an application might want only TCP packets destined for a `Telnet port`. In such cases, it is highly profitable to allow application program to specify a boolean function on network packets, and then have this filter run within kernel's address space. The kernel can then avoid delivering uninteresting packets to the application, thereby saving the cost of many unnecessary context switches. Our technique can prove to be an alternative to PCC for such applications (if our technique could provide a sufficient condition i.e. provable implies decryptable).

Another possible application could be in the implementation of a program checker. A user upon receiving a program wants to verify whether the terminal executing the program has properly evaluated the program on a particular input. Another possible variant could be that : we suppose that an independent software vendor publishes a program with a signature and a user wants to execute the program on an input (possibly via an agent) and wants some sort of a proof or a signature derived from the original one that attests to the fact that it was exactly the same program that was evaluated on the input.

7 Conclusion and Future work

Until now, proofs have been seen as programs, the following table explains this with an example :

Property	**Proof**
`Type` ϕ	λ-term $\pi : \phi$
$\phi = (A \to (A \to B)) \to B$	$\pi = \lambda x.\lambda f.fx$

We observe that from any proof π_A of A, we can construct a proof $\pi_{A\to B}$ of $A \to B$ and using π we can construct a proof π_B of B.

$$\pi\pi_A\pi_{A\to B} = (\lambda x.\lambda f.\lambda fx)\pi_A\pi_{A\to B} = \pi_{A\to B}\pi_A = \pi_B$$

With the help of the technique presented in this report we try to see proofs as cryptography and try to extend the Curry-Howard isomorphism to cryptography though we are successful in only one direction (i.e. a decryption algorithm π for a nonce encrypted under a property ϕ gives a proof of the property).

Property	**Proof**
`Type` ϕ	λ-term $\pi : \phi$
Encryption key ϕ	Decryption algorithm π

Use of cryptography to implement zero-knowledge proof makes the implementation more efficient compared to that of PCC and hence the user is no longer forced to send the proof with the code which might contain informations that the user should not have access to. In the PCC framework, the VC predicate is computed twice : once to prove it and the other time to verify it. This might be a waste of time. The other point to note is that the verification process might be long and difficult; in our case it is just a string comparison of the nonce sent and the nonce received. In all these contexts, our technique proves to provide an upper hand to both the user and the developer.

The Proof-Carrying Code is considered to be intrinsically safe: most attempts to tamper with either code or the proof results in a validation error. In the few cases, when the code and the proof are modified such that validation still succeeds, the new code is also safe. But, in this case, the modified program might not be the one that the consumer is waiting for. Our technique rejects the result of any possible tamper either with the property or with the program and this is made possible by the intrinsic cryptographic system.

A further observation reveals that the encryption of a nonce under a property is building a part of the proof tree (in the worst case building it completely) and decryption is to finish the rest of the tree. The future work would be to prove that *Provability* implies *Decryptability* which might require a complete shift from the encryption algorithm. And then we could try to extend the logical framework and search for an efficient encryption scheme that can be implemented in first-order constructive logic (such that it deals with the quantifiers) and reduces the size of the encrypted nonce (which is sending a nonce under an atomic formula with several keys). Another line of work would be to make empirical tests on the application of this technique while reducing the size of the information passed to the developer.

References

[AFV11] S. Agarwal, D. M. Freeman, and V. Vaikunthanathan. Functional encryption for inner product predicates from learning with errors. In *Asiacrypt*, 2011.

[AS92] S. Arora and S. Safra. Probabilistic checking of proofs : A new characterization of NP. In *Proceedings of 33rd IEEE Symposium on Foundations of Computer Science*, 1992.

[Bel06] M. Bellare. *Introduction to Pairing-Based Cryptography.* University of California at San Diego, 2006.

[BRS05] V. Bernat, H. Rueß, and N. Shankar. First order cyberlogic. Computer Science Laboratory, SRI International, 2005.

[BSW11] D. Boneh, A. Sahai, and B. Waters. Functional encryption: Definitions and chanllenges. In *TCC*, 2011.

[BW07] D. Boneh and B. Waters. Conjunctive, subset and range querries on encrypted data. In *TCC*, 2007.

[Dow11] G. Dowek. *Proofs and Algorithms : An Introduction to Logic and Computability.* Springer-Verlag London, 2011.

[KSW08] J. Katz, A. Sahai, and B. Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *Eurocrypt*, 2008.

[Mer83] D. Meredith. Seperating minimal, intuitionist and classical logic. *Notre Dame Journal of Formal Logic*, 24(4):485–490, October 1983.

[Nec97] G. C. Necula. Proof carrying code. In *Popl*, 1997.

[O'D] M. J. O'Donnell. The SKI combinator calculus : a universal formal system. http://people.cs.uchicago.edu/~odonnell/Teacher/Lectures/Formal_Organization_of_Knowledge/Examples/combinator_calculus/.

[Sho06] V. Shoup. *Sequence of Games : A tool for Taming Complexity in Security Proofs.* New York University, 2006.

[SW05] A. Sahai and B. Waters. Fuzzy identity-based encryption. In *Eurocrypt*, 2005.

[Tho99] S. Thompson. *Type theory and Functional Programming.* University of Kent, 1999.

[Wat08] B. Waters. Functional encryption: beyond public key cryptography. http://userweb.cs.utexas.edu/~bwaters/presentations/files/functional.ppt, 2008.