Overview of EasyCrypt tool and its possible applications for ProveIt

Kristjan Krips

University of Tartu, Estonia

Abstract. We aim to show if and how it would be possible to use the functionalities of EasyCrypt in ProveIt. For that, we compare the EasyCrypt language and the ProveIt language, describe what kind of properties can be proven with EasyCrypt about games and specify the main transformation rules for the compiler form ProveIt language to EasyCrypt language.

1 Introduction

ProveIt is a tool that should help researchers to verify the correctness and security of protocols that use symmetric primitives. Symmetric primitives are low-level symmetric-key algorithms. The probability of breaking the protocol can be found with ProveIt by doing a finite number of reductions.

EasyCrypt is an automated tool that can verify the security proofs of cryptographic games [BGC+12]. To do that, it uses Probabilistic Relational Hoare Logic, Satisfiability Modulo Theories (SMT) solvers and theorem provers. EasyCrypt has formal proofs for their proving process.

Currently ProveIt does not have formal proofs for all of the proof steps that can be done with the tool. Contrary to EasyCrypt, ProveIt does not support transformations that manipulate arithmetic operations. Therefore, it would be useful to use EasyCrypt in the toolchain of ProveIt. This would give ProveIt additional proving functionalities and it could be possible to verify the proofs created in ProveIt.

2 ProveIt language and EasyCrypt language

2.1 ProveIt language

ProveIt has its own syntax and the input to ProveIt has to be written in ProveIt language. ProveIt language is a modified version of the imperative While language [NN92]. ProveIt language differs from While as it contains probabilistic statements, it is a probabilistic While language.
The ProveIt language consists of statements that are composed of variables, operations and expressions. Expressions are composed of operations and variables. ProveIt language uses four main categories of objects: constants, variables, sets and functions.

Statements The simplest examples of statements in ProveIt language are assignment and uniform choice. An assignment evaluates a variable by giving it the value from the right side of the assignment operator. The right side of an assignment operator can be a function call, an expression, a variable or a constant. On the left side of the assignment operator can be a single variable or a tuple of variables. To write a tuple in ProveIt language the variables have to be enclosed in brackets.

A uniform choice evaluates a variable by assigning it a random value from a given set, i.e., the value is chosen with uniform probability. The left side of the uniform choice operator can be a single variable or a tuple of variables enclosed in brackets. Table 1 gives examples for assignments and uniform choice.

Statement	**Syntax example**	**Rendered example**
assignment	$x := b$ $(x, y) := (a, b)$	$x := b$ $(x, y) := (a, b)$
uniform choice	x <- Z	$x \leftarrow Z$

Table 1: Syntax examples for assignment and uniform choice.

In addition, there are statements for functions, statements for the adversary and statements that modify the control flow, i.e., statements that affect the order of program execution. If statements, for statements and while statements modify the control flow of a program. The syntax examples for the control flow statements are given in Table 2.

Statement	**Syntax example**	**Rendered example**
if statement	$\mathsf{if}(x < 6)\{$ $x := x + 1\}$	$\mathsf{if}(x < 6)$ $\lfloor x := x + 1$
	$\mathsf{if}(x < 6)\{$ $x := x + 1\}$ $\mathsf{else}\{$ $x := x + 2\}$	$\mathsf{if}(x < 6)$ $\lfloor x := x + 1$ else $\lfloor x := x + 2$
for statement	$\mathsf{for}(x := 0; x < 6; x := c)\{$ $y := x + 4$ $c := c + 1\}$	$\mathsf{for}(x := 0; x < 6; x := c)$ $\lceil y := x + 4$ $\lfloor c := c + 1$
while statement	$\mathsf{while}(x < 6)\{$ $x := x + y$ $y := y + 1\}$	$\mathsf{while}(x < 6)$ $\lceil x := x + y$ $\lfloor y := y + 1$

Table 2: Examples of an if statement, for statement and a while statement.

ProveIt language has statements for function signature, function definition, a function call and function sampling. A function signature defines the name, domain and range of the function. A function definition specifies what the function does by including the body of the function. A function call statement denotes running a function with the given arguments. A function sampling statement uniformly chooses a function from the set of all functions between the given domain and range. The example statements for defining and using the functions are given in Table 3.
An adversary in ProveIt language can have zero or more arguments. The adversary signature specifies if the adversary has access to oracles.

Statement	**Syntax example**	**Rendered example**
function sampling	$f <- Func(f : \mathcal{K}\ times\ \mathcal{P}\ -> \mathcal{C})$ $g <- Func(g : \mathcal{X}\ -> \mathcal{Y})$	$f \leftarrow Func(f : \mathcal{K} \times \mathcal{P} \to \mathcal{C})$ $g \leftarrow Func(g : \mathcal{X} \to \mathcal{Y})$
function signature	$f : \mathcal{K}\ -> \mathcal{C}\ times\ \mathcal{C}$ $g : \mathcal{K}\ times\ \mathcal{M}\ -> \mathcal{C}$ $h : \mathcal{K}\ -> \mathcal{C}$	$f : \mathcal{K} \to \mathcal{C} \times \mathcal{C}$ $g : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$ $h : \mathcal{K} \to \mathcal{C}$
function definition	$\mathsf{fun} f(x)\{$ $x := x\%2$ $y := x + 1$ $return(x, y)\}$	$\mathsf{fun} f(x)$ $\left[\begin{array}{l} x := x\ mod\ 2 \\ y := x + 1 \\ return(x, y) \end{array}\right.$
function call	$(x, y) := f(a)$ $x := g(a, b)$ $x := h(a)$	$(x, y) := f(a)$ $x := g(a, b)$ $x := h(a)$

Table 3: Examples of a function sampling, function signature, function definition and a function call.

Statement	**Syntax example**	**Rendered example**
adversary signature	$Adv : \mathcal{X}\ -> \mathcal{Y}$ $Adv\_1 : \mathcal{X}\ times\ \mathcal{Z}\ -> \mathcal{Y}$ calls h $Adv\_2 : \mathcal{X}\ times\ \mathcal{Z}\ -> \mathcal{Y}$ calls h,f	$Adv : \mathcal{X} \to \mathcal{Y}$ $Adv_1 : \mathcal{X} \times \mathcal{Z} \to \mathcal{Y}$ calls h $Adv_2 : \mathcal{X} \times \mathcal{Z} \to \mathcal{C}$ calls h,f
adversary call	$x := Adv\_1(x, z)$	$x := Adv_1(x, z)$

Table 4: Examples of an adversary signature and an adversary call.

2.2 EasyCrypt language

EasyCrypt language is also a probabilistic While language. EasyCrypt language contains assignment statements, random sampling statements, conditional statements, while loop and function calls. It is a strongly typed language and the syntax is dependently typed, i.e., only correctly typed programs are allowed. The semantics of EasyCrypt language is formalised in the Coq proof assistant [dt04].

In EasyCrypt language the games are composed of global variables, defined functions and abstract adversary declarations. The following description of the syntax of EasyCrypt language is summarised from [BGC+12].

Basics

Types. EasyCrypt provides built in types and also allows the user to define new types. The built in types are: unit, bool, int, real, bitstring, list (polymorphic), map (polymorphic), product types (uses infix notation). It is possible to create abstract types, type synonyms and parametric types. When creating a new type it is important to check that the new type identifier does not conflict with an already existing type.
It is possible to define abstract types by writing the keyword `type` followed by the name of the abstract type, e.g.,

```
type private_key .
```

Type synonyms can be declared by writing `type type_identified = type_exp`, where type expression is composed of built in types, e.g.,

```
type private_key = int. .
```

Parametric types are declared with the following syntax: `type 'var type_ident`, where var is the parameter, the symbol ' denotes that the type is parametric and type_ident stands for the type identifier. E.g., the following declares a parametric array

```
type 'var array
```

and then we can declare an array that contains only the type int by writing

```
type intlist = int array .
```

Constants. Constants can be declared with the following syntax:
`cnst identified : type_expression [exp]`, where exp is an optional expression, e.g., `cnst pk : private_key`.

Operations. Operations are written with the following syntax:
`op op_indentifier : fun_type [as id]`, where the operation identifier can be either alpha-numeric or a binary operator. The operation identifier can contain the following symbols: $=, <, \tilde{}, +, \%, \hat{}$ in square brackets. Here is an example of the usage of the operation keyword

```
op add : real -> real .
```

To declare a polymorphic operation the operands have to be written out in the polymorphic identifier, such example is given in the EasyCrypt documentation:
`op hd : 'a list -> 'a .`
It is also possible to create operations from expressions, see [BGC+12]. EasyCrypt has an explicit separation between deterministic and randomised operations. There is a probabilistic operation that is used to create a probability distribution. The syntax for probabilistic operations is `pop ident : fun_type`, where ident stands for the indentifier, e.g.,

```
pop gen_key : int -> private_key  .
```

Logical formulas. Logical formulas are composed of Boolean expressions, connectivities, defined predicates and variable quantifiers. The universal quantifiers is denoted by `forall (x,y:int), p(x,y)`, where p is a first-order formula and x, y are variables. The logical formulas can be used in creating axioms and in judgements to claim the equality of games. E.g., we could define an axiom for commutative addition by

```
axiom com_add : forall(x, y:int), x + y = y + x.
```

Predicates. The syntax for writing predicates is `pred ident(param) = p`, where ident is an identifier, param is a list of arguments and p is a first-order non-relational formula. E.g., predicates can be used in the axioms and lemmas. The following is an example from the EasyCrypt manual, [BGC+12]

```
pred injective(T:('a, 'b)  map) =
forall (x,y:'a), in_dom(x,T) => in_dom(y,T) => T[x] = T[y] => x = y.
```

Axioms and Lemmas. Axioms can describe abstract operators and types. In addition, axioms can be used for writing hypotheses over declared constants. Thus, axioms give additional informations to the provers used by EasyCrypt, therefore they are an important part of the proofs. E.g., we could define an axiom for commutative addition by

```
axiom com_add : forall(x, y:int), x + y = y + x.
```

Lemmas can be written to verify goals in the proofs. Lemmas can be declared with the following syntax: `lemma ident : p`, where ident is an identifier and p is a first-order non-relational formula.

Game declarations

Statements. Assignments and function calls must end with a semicolon, however there must not be a semicolon after a conditional block or a while loop. The syntax of the if statement and while is the same as in the ProveIt language. Similarly, the curly brackets are not needed if the body of the if or while contains only one instruction.

Probabilistic statements. EasyCrypt language allows probabilistic assignments and there are several way to create such assignment. It is possible to assign value from a probability expression, integer interval, arbitrary length bitstrings or probabilistic operation.

1. $x = \{0, 1\}$
2. $x = [0..q - 1]$
3. $x = \{0, 1\}^k$
4. $x = gen\_secret\_key(0)$

Function definitions. Functions can be defined by writing out the function body or by creating a synonym to a function that is already defined. The following syntax can be used to create a function

`fun identifier (typed_arguments) : return_type = { function_body } .`

To create a function from an already existing function, the following syntax has to be used

`fun identifier = game_identifier.fun_identifier .`

Adversary. Adversary signatures must be defined outside of a game declaration. The syntax of an adversary signature is:
`adversary sign_ident(typed_args) : result_type {sign_1, ..., sign_k}`, where result_type is a type expression for the return type and sign_1, ..., sign_k is a list of oracle signatures which only contain the input and output types of the oracles. The adversary signature may have no oracle signatures, in this case the list of oracle signatures is empty.
Adversary can be defined in an abstract way or as a synonym of an already defined adversary. The following syntax has to be used to define an adversary in an abstract way

`abs adv_identifier = adv_sign_identifier` $\{ident_1, \ldots, ident_k\}$,

where $ident_1, \ldots, ident_k$ are already defined functions that represent the oracles. To create a synonym of an already existing adversary the following syntax has to be used

`fun adv_identifier = game_identifier.adv_identifier .`

Game. A game can be defined with the syntax `game_ident = {game body}`. The game body can contain global variable declarations, function definitions and abstract adversary declarations. In the game definition the variable declarations must not be separated by semicolons.
The other option is to redefine a game. In this case it is possible to remove or add variables or to redefine functions from an already defined game. This can be done with the following syntax:
`game ident = existing_ident var_modifiers where ident_1 ={fun_body} and ...and ident_k = {fun_body}`, where var_modifiers is a list of optional statements of the form `remove ident_1, ..., ident_k`. Var_modifiers can also contain a list of new variable declarations..

3 Proving process in EasyCrypt

EasyCrypt uses Hoare logic and weakest precondition calculus to prove the correctness of the programs. EasyCrypt uses Why3 Software verification platform [FP13]. It works in top of SMT solvers, automated provers and interactive provers. The SMT solvers used in EasyCrypt are Alt-Ergo [alt], CVC3 [BT07], Z3 [dMB08] and Yices [DdM06]. The automated provers used in EasyCrypt are Vampire [RV02], E-Prover [e-p], SPASS [WBH$^+$02]. EasyCrypt also uses an interactive prover Coq.

3.1 Proving steps

As a first step the user has to declare the types, constants, operations and predicates that will be used in the following proof. In the second step the user writes the axioms and lemmas that will be used in the games in the corresponding proof. The third step consists of creating the games. This means creating the functions, adversaries, statements.The fourth step is to create the pRHL judgements and to prove them, this is done by applying EasyCrypt tactics, see [BGC$^+$12]. The fifth and final step is to create the probability claims.

4 Translation from ProveIt language to EasyCrypt language

4.1 Translating the syntax

The rules for translating the types and variable declarations from ProveIt language to EasyCrypt language can be found straightforwardly.

EasyCrypt language supports the variable evaluations that are available in ProveIt language, it is possible to evaluate a single variable or a tuple of variables. The

assignable can be a variable, constant, expression or a function call. In the following table we give examples of the translation rules.

Statement	ProveIt	EasyCrypt
assignment	a:=b	a=b;
	a := f(x)	a = f(x);
	(a,b) := (c,d)	(a,b) =(c,d);

EasyCrypt does not allow to uniformly choose a value from an abstract set. Therefore, this kind of uniform choice statement has to be transformed into a sequence of statements in EasyCrypt that have the same semantic meaning.

Statement	ProveIt	EasyCrypt
uniform choice	$a <-\{0,1,2\}$	a={0,1,2};
	$a <-K$	type sample_K. *pop gen_element* : ()− > *sample_K* $a = gen\_element()$;

There is a function definition in EasyCrypt language that corresponds to both the function signature and function definition in the ProveIt language. Therefore, the signature and function definition from ProveIt language should be translated into a function definition in EasyCrypt language.

Statement	ProveIt	EasyCrypt
function signature	$f : \mathcal{K}$ -> $\mathcal{C}$ *times* $\mathcal{C}$	type K. type C.

function definition	fun$f(x)\{$
	$x := x - 2$
	$y := x + 1$
	$return(x, y)$
	}
		fun f(x : K) : (C, C) = {
		$x := x - 2$;
		var $y := x + 1$;
		$return(x, y)$;
		}

The syntax of the if statement is identical in both languages. Also the syntax of the while statement is identical in both languages. The for statement and do-while statement of the ProveIt language have to be transformed into a while statement in the EasyCrytpt language.

Statement	ProveIt	EasyCrypt

for	for$(x := 0; x < 6; x := x + 1)\{$
	$y = y + x$
	}
		var x = 0;
		while$(x < 6)\{$
		$y = y + x$
		}
		$x = x + 1$

do-while	do{ $y = y + x$ }while$(x < 5)$	$y = y + x$ while$(x < 5)\{$ $y = y + x$ }

The adversary signature in EasyCrypt language specifies the return type of the adversary. Other than that the transformation is straightforward. In the following example we assume that the function f has one parameter of type int and that it outputs one return value of type int.

Statement	ProveIt	EasyCrypt
adversary signature	Adv_1 : Int -> Int	adversary $Adv\_1(a : int) : int \{\}$.
adversary signature	Adv_1 : Int -> Int calls f	adversary $Adv\_1(a : int) : int \{int \text{ -> } Int\}$.
adversary definition		abs $Adv = Adv\_1\{\text{f}\}$.

The main function is similar in both languages, there are small differences in the header of the function. In the following table we show how to transform the header of the main function written in ProveIt language into a corresponding header of main function in the EasyCrypt syntax.

Statement	ProveIt	EasyCrypt
main header	main {	fun Main(): int = {

4.2 Translating the game transformations

The proof steps are done in EasyCrypt by applying tactics provided by EasyCrypt. Our goal is to link the game transformations used in ProveIt with the tactics provided by EasyCrypt. To do that we translate the game transformations used in ProveIt into EasyCrypt tactics. We give the overview of only the tactics that are used for translating transformations available in ProveIt, for the complete description of EasyCrypt tactics, see [BGC+12].

Tuple decomposition. Tuple decomposition transformation in ProveIt takes an evaluation of a tuple and decomposes it into a sequence of assignment statements. This is a syntactic transformation so to translate the transformation we could create a new game in the EasyCrypt proof where the tuple is removed and the new assignment statements are added. This can be done with the game redefining statement which was briefly described earlier. A claim should be entered after creating the new game to claim that the old game and new game are equivalent.

Function inlining. Function inlining transformation in ProveIt takes a function call and replaces it with the contents from the corresponding function definition body. There is a tactic in EasyCrypt for function inlining. It allows to specify the function names or the positions of function calls that will be inlined. It uses the following syntax:

$$\text{inline } [side]\ [P_1, \ldots, P_j |\ position], \tag{1}$$

where position is a number or last. If several function names are given then the inlining is done in the given order. In the case where there are no arguments

then all function calls are inlined. This tactic can be directly used to translate function inlining transformation in ProveIt into EasyCrypt.

Statement switching. The statement switching transformation in ProveIt takes a statement and moves it past another statement if the move is allowed, i.e., it does not change the values of other statements and the moved statements. EasyCrypt has a swap tactic that can be directly used to translate the ProveIt transformation into a proof step in EasyCrypt. It uses the following syntax:

$$\mathsf{swap}\ [side]\ [pos - pos]\ num, \tag{2}$$

where pos-pos denotes the range of statement indexes that are moved and num denotes how many statements upward or downward the block is moved. If the last argument is negative then the block is moved up otherwise the block is moved down.

Basic tactics for proving the equality of games If we have a judgement about the equality of two games, then we have to prove it with the use of EasyCrypt tactics. Let the old game be identified by Old and the new game by New. Let the connection between the two games be described with following judgement:

$$equiv\ Fact1\ : New \tilde{} Old : \{true\} ==>= \{res\}. \tag{3}$$

$$\tag{4}$$

The judgement can be proven by sequentially applying tactics. We have already described the function inlining tactic and swap tactic, now we describe a few other the commonly used tactics.

Let. The let tactic allows to add a new assignment to the game, however the new variable has to be independent of the other variables used in the game.

Rnd. The end tactic allows to show the equivalence of two random assignments from the same distribution.

Wp. The wp tactic finds the weakest precondition of the games in the judgment. It works on deterministic games that contain no loops and no function calls. However, this tactic returns failure if there is nothing left to prove.

Trivial. The trivial tactic is combination of wp tactic and rnd tactic. In addition to wp, it tries to match random assignments.

Simpl. The simple tactic finds the weakest precondition of the games in the judgment starting from the bottom of the games. After that it removes the trivial cases. It works on games that are deterministic and don't contain loops.

Auto. The auto tactic finds the weakest precondition of the games in the judgment starting from the bottom of the game. It works on deterministic games which do not contain loops. When it finds a function call then it checks if there is an appropriate judgement. It stops if it reaches a random assignment statement.

Unroll. The unroll tactic takes the body of the while loop, places it inside an if condition and inserts it before the while statement. The if condition is the same as the while loop condition. The unroll tactic does not change the while statement.

Unroll for The unroll for transformation unrolls the for loop in the case of a simple looping condition. As EasyCrypt language does not contain a for loop we have to transform the for loop into a while loop. We can use the unroll tactic in EasyCrypt to prove that an unrolled while loop is equal to the while loop.

As a first step a new game would have to be created in EasyCrypt where the while statement is unrolled according to the unroll tactic. The completely unrolled game contains as many if conditions containing the while body as the number of times the while loop would be traversed.
As a second step another game is created based on the unrolled game but in this case the while statement is removed. A judgment is used to claim that the two games are equal and then the judgement is proved by applying different tactics. First the if conditions are evaluated to the true value using the condt tactic and then the while loop is evaluated using the condf tactic which removes the while loop as the condition of the loop evaluates to false. The following contains an example of the applied tactics.

1. condt{1} at 2,3,4,5;trivial.
2. condf{1}.
3. wp.
4. simpl.

Dead code elimination The dead code elimination transformation in ProveIt removes the redundant code from the game. This is a syntactic transformation so to translate the transformation we could create a new game in the EasyCrypt proof where the redundant code is removed and claim that the old game and the new game are equal. To use the weakest precondition tactic the game would have to be deterministic, loop free and without function calls. So at least the following tactics should have to be used to prove the equality of these games.

1. inline.
2. rnd.
3. wp.

In addition the loops should be unrolled as the weakest precondition tactic does not work on loops.

5 Summary

As a result of this paper we have the knowledge how to transform some ProveIt game sequences into EasyCrypt language. We found out how to use the EasyCrypt tactics to prove the results of some syntactic transformations in ProveIt. We also briefly described how the proving process works in EasyCrypt.

References

[alt]

[BGC+12] Gilles Barthe, Benjamin Gregoire, Juan Manuel Crespo, Cesar Kunz, and Santiago Zanella Beguelin. *The EasyCrypt tool - Documentation and User's Manual*, September 2012.

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[DdM06] B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, August 2006.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceed*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[dt04] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[e-p]

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 – Where Programs Meet Provers. In *ESOP'13 22nd European Symposium on Programming*, volume 7792 of *LNCS*, Rome, Italie, March 2013. Springer.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: a formal introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1992.

[RV02] Alexandre Riazanov and Andrei Voronkov. The design and implementation of vampire. *AI Commun.*, 15(2,3):91–110, August 2002.

[WBH+02] Christoph Weidenbach, Uwe Brahm, Thomas Hillenbrand, Enno Keen, Christian Theobalt, and Dalibor Topi? SPASS version 2.0. In Andrei Voronkov, editor, *Automated deduction, CADE-18 : 18th International Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 275–279, Kopenhagen, Denmark, 2002. Springer.