UNIVERSITY OF TARTU

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science

Computer Science

Kristjan Krips

A Tool for the Formal Analysis of Symmetric Primitives

Master's thesis (30 ECTS)

Supervisor: Sven Laur, D.Sc. (Tech.)

Author: .... "........" May 2012

Supervisor: .... "........" May 2012

Admitted to thesis defense

Professor .... "........" May 2012

Tartu 2012

Contents

Chapter 1

Introduction

Cryptography is widely used to secure the electronic communication in the modern world. For example, mobile phone traffic, Internet shopping and Internet banking are secured with the use of cryptography. Cryptographic algorithms can be divided into fast symmetric and slower asymmetric algorithms. The symmetric algorithms use one secret key for both encryption and decryption, while the asymmetric algorithms use a public key for encryption a secret key for decryption. While using cryptography, we have to be sure that the cryptographic algorithms and cryposystems built from these algorithms are provably secure. Therefore, it is important to mathematically analyze and prove their security. This thesis describes one method for proving the security of cryptographic primitives.

Symmetric primitives are low-level symmetric-key algorithms that are specified by their behavior. Symmetric encryption schemes and hash functions are examples of symmetric primitives. A primitive can be modeled using a security game. A security game models the interaction between the primitive and a well defined environment that contains an adversary. When given a security game containing a symmetric primitive, the probability of a successful attack can be found by doing a finite number of game rewritings.

Game rewriting is a technique in game-based proofs that creates a new modified game. A game-based proof consists of a finite number of game rewritings that make the security analysis of the initial game easier. A game is modified by applying a game transformation on it, that changes the construction of the game. However, a modification of the game can change the probability of a successful attack against the modeled primitive. Therefore, we should know how the success probability of an adversary changes by applying a game transformation. Reductions are used to solve the problem with unknown probabilities. Reduction is a proof construction that shows that a specific game transformation can be applied. By applying a reduction based game transformation, we know how the success probability of an adversary changes. E.g. a reduction based game transformation can replace an if condition with one of its branches or simulate a random function.

As the game-based security proof of a primitive can contain several game transformations it is convenient to decompose the proof, to make the it readable. Therefore, we need to rewrite the security game after each game transformation. However, it is not convenient to do such proofs on paper and therefore a tool would help the researcher.

In this work, I am extending a tool called ProveIt with reduction based game transformations that make it possible to prove the security of symmetric primitives. ProveIt is a tool that is currently being developed for doing the game-based proofs. It lets the user to choose which reductions to do on a given game. After parsing a game into an abstract syntax tree (AST) ProveIt lets the user to transform the initial game one step at a time in order to find the probability of a successful attack against it. After a finite number of game transformations the proof is complete and the probability of a successful attack is found. However, currently ProveIt is able to do only a few reduction based game transformations and this limits the usability of the tool.

In this thesis we describe the necessary subset of game transformations that are required for doing game-based proofs. We show where different game transformations can be applied and describe how we implemented these transformations in ProveIt.

As a result of this work we created semantics to the language used in ProveIt, implemented several reduction based game transformations and gave the theoretical background for these transformations. For defining the semantics, I had to learn how to construct operational semantics and small-step semantics. Besides that, to implement the game transformations, I had to study the reductions that allow to use these transformations. To be more specific about the new transformations, I implemented the following game transformations: Dead Code Elimination, Statement Switching, Remove Condition, Replace Function Call, Wrap and Random Function Simulation. In particular, to implement the Dead Code Elimination, I had to create a control flow graph and implement liveness analysis on it.

1.1 ProveIt

ProveIt is a proof assistant for game-based proofs. ProveIt has its own high level language with a specific syntax and the games have to written in this language. After inserting the game ProveIt uses a parser to generate an abstract syntax tree of the game. When the game is parsed then ProveIt shows views for the game, the abstract syntax tree and the proof steps. The syntax tree is composed of statements, expressions and operations. The reductions are done on this abstract syntax tree using the game transformations enabled by ProveIt. The game transformations have to be done manually, i.e., the user has to interact with ProveIt. If a game transformation can be applied to different nodes in the AST, then the user can choose the node on which the reduction is made. However, before applying a game transformation all necessary conditions are checked and therefore the user is guaranteed that a completed game transformation is valid. The resulting security game for each successful reduction is saved in a list of proof steps that is displayed to the user. The game in a proof step can be modified and this creates a new branch to the proof. When the proof is finished all the proof steps are displayed to the user and the security guarantees are assembled.

Chapter 2

Syntax of ProveIt language

2.1 Introduction

We begin by describing what we mean by a *security game*. A cryptographic primitive can be formulated through a game (program) that models the environment where the primitive is used and allows the adversary to attack the primitive. We can measure the success of an adversary in a game by the probability of a correct output for a given problem. ProveIt uses a modified version of the imperative While language [NN92]. The modified language is called ProveIt language and it has to be used for formalizing the games.

2.2 Syntax of ProveIt language

In the following section, we introduce the syntax of the ProveIt language and describe the rules for combining the syntax into valid input. The syntax that we describe is based on the manual [Kam11]. The ProveIt language consists of statements that are composed of variables, operations and expressions. Expressions are composed of operations and variables, e.g. $a + b$ is an arithmetic expression as it contains an arithmetic operation and $x < y \,||\, x > y$ is a logical expression as it contains logical operations.

2.2.1 Constants and variables.

ProveIt language uses four main categories of objects: constants, variables, sets and functions. Constants are numbers, e.g. 2, 3, 4 are valid examples of constants in ProveIt language. The current agreement is to write the variable names with alphanumeric characters, but there are no strict rules fixed e.g. the following variable names a, $b3$, xZY are valid. By the current convention we write sets with uppercase letters, e.g. $\mathcal{N}$, $\mathcal{Z}$, $\mathcal{RR}$. Function names can be written with lowercase alphanumeric characters and therefore function names f, $f1$ are valid. We use a convention to write the names of local variables, that are defined in the function body, with the function name followed by a dot and the name of the local variable. For example, to use a local variable x in function f we write $f.x$.

2.2.2 Operations and expressions.

Expressions are used as a basis for building statements in ProveIt language. Expressions can be divided into arithmetic expressions and logical expressions based on the used operation.

Arithmetic operations are commonly used for modeling the arithmetics in the cryptographic primitives or the interaction with the adversary. Arithmetic operations are defined between two variables and are described in Table 2.1. This table describes each operation by giving the symbol of the corresponding operator, the name of the operation, an example written in ProveIt syntax and the output of the example after being rendered in ProveIt.

The precedence of the arithmetic operations is given by the ordering in Table 2.1, the operation with the highest precedence is in the top of the table and the operation with the lowest precedence is in the the bottom of the table.

To use a lower precedence operation as a member of a higher precedence operation, the lower precedence operation has to be enclosed in brackets. If we want to write two to the power of $a + b$, then we have to write $2\hat{}(a + b)$. As a concrete example of the precedence of the arithmetic operations consider the following rendered expression $(a + b) \cdot (a - b) \cdot 2^{2+a}$. In this expression the addition, the subtraction and the exponent have to be enclosed in brackets.

In addition to arithmetic operations, ProveIt language has logical operations for modeling conditions. As with the arithmetic operations, the logical operations are

Operator	**Operation name**	**Example**	**Output**
^	exponentiation	a^b	a^b
$*$	multiplication	$a * b$	$a \cdot b$
$\backslash$	division	$a\backslash b$	$\frac{a}{b}$
%	reminder	$a\%b$	$a \bmod b$
$+$	addition	$a + b$	$a + b$
$-$	subtraction	$a - b$	$a - b$

Table 2.1: List of all arithmetic operations that are allowed in ProveIt.

used for modeling cryptographic primitives and the interaction with the adversary. These operations are described in the Table 2.2. The table about logical operations is structured in the same way as the Table 2.1 about arithmetic operations. In particular, the precedence of the operations is given by the ordering of the table.

Operator	**Operation name**	**Example**	**Output**
!	negation	$!a$	$-a$
$<$	less than	$a < b$	$a < b$
$>$	greater than	$a > b$	$a > b$
$<=$	less than or equal to	$a <= b$	$a \leqslant b$
$>=$	greater than or equal to	$a >= b$	$a \geqslant b$
$! =$	inequality	$a! = b$	$a \neq b$
$=$	equality	$a = b$	$a = b$
&&	logical and	$a\&\&b$	$a \wedge b$
$\|\|$	logical or	$a\|\|b$	$a \vee b$

Table 2.2: Logical operations of the ProveIt language.

The precedence of the logical operations is lower than the precedence of the arithmetic operations. For instance, in the expression $4 + a > 5 + a - 3$ we first evaluate $4 + a$ and $5 + a - 3$ and then compare the results.

Additionally, ProveIt language has operations for sets. These operations allow to create a new set from elements, check the set membership and take the union, intersection and difference of sets, i.e., to create new sets. The syntax and the description of these operations is given in Table 2.3.

Operator	**Operation name**	**Example**	**Output**
\in	in	a\in M	$a \in \mathcal{M}$
\notin	not in	a\notin M	$a \notin \mathcal{M}$
\union	union	K\union M	$\mathcal{K} \cup \mathcal{M}$
\intersect	intersection	K\intersect M	$\mathcal{K} \cap \mathcal{M}$
\setminus	difference	K\setminus M	$\mathcal{K} \setminus \mathcal{M}$
{}	set tuple	$\{0, 1, 2\}$	$\{0, 1, 2\}$

Table 2.3: Operations for creating new sets and checking set membership.

Besides the previously described operations ProveIt language allows to define Cartesian product, describe function types and create a set of functions with the same type. Cartesian product and function type operations are defined between sets and are used to build the function signature. A set of functions is used for uniformly choosing a function from the set of functions with the same type. These operations are described in Table 2.4.

Operator	**Operation name**	**Example**	**Output**
\times	Cartesian product	K\times M	$\mathcal{K} \times \mathcal{M}$
->	function type	M -> C	$\mathcal{M} \rightarrow \mathcal{C}$
{}	set of functions	$\{f : M$ -> $C\}$	$\{f : \mathcal{M} \rightarrow \mathcal{C}\}$

Table 2.4: Operations for the function signature and the set of functions.

2.2.3 Statements

Simple statements. The simplest examples of statements 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.

As previously described, the syntax of ProveIt language forces the sets to be written in uppercase letters. Table 2.5 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 2.5: Syntax examples for assignment and uniform choice.

Control flow. 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. An if statement branches the execution of a program based on the result of a logical operation. The statement has to contain a logical expression and can contain up to two branches, the if branch and the optional else branch.

A for statement executes a block of code for a fixed number of times. The for statement is actually a C-style for cycle. The header of a for statement has to contain an assignment statement, a logical expression and another assignment statement. The statements and the logical expression in the header of the for statement are separated by semicolons.

A while statement executes a block of code until the condition fails. The condition has to be a logical expression. The syntax examples for the control flow statements are given in Table 2.6.

Functions and scoping. ProveIt language has statements for function signature, function definition and a function call. 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. The function body has to contain a return statement. The return statement can return a single value or a tuple of values.

A function call statement denotes running a function with the given arguments. The arguments of the function call are variables or constants. The function call can

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

Table 2.6: Examples of an if statement, for statement and a while statement.

be used to evaluate a variable or a tuple of variables that are enclosed in brackets. The example statements for defining and using the functions are given in Table 2.7.

As the function definition can create local variables we have to describe the scoping rules of ProveIt language. There are two scopes in ProveIt language, the local scope and the global scope. A program written in ProveIt language can contain global and local variables. Global variables are defined outside of the function body and are accessible at any program point. I.e., the variables of the security game are considered to be global variables. Local variables are defined in the function body. Besides that, we consider the arguments of the function as local variables. Therefore, the local variables are accessible only in the function body and can not be accessed by other functions, i.e., by the functions that have not defined these variables. These rules force the local variables to be named uniquely, there can not be a local variable with the same name as a global variable. Recall that we can use the naming convention $f.x$ for a local variable defined in a function f.

Statement	**Syntax example**	**Rendered example**
function signature	$f : \mathcal{K}$ -> $\mathcal{C}\backslash times\,\mathcal{C}$ $g : \mathcal{K}\backslash times\,\mathcal{M}$ -> $\mathcal{C}$ $h : \mathcal{K}$ -> $\mathcal{C}$	$f : \mathcal{K} \to \mathcal{C} \times \mathcal{C}$ $g : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$ $h : \mathcal{K} \to \mathcal{C}$
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 2.7: Examples of a function signature, function definition and a function call.

$y := \mathcal{A}(x)$	$y := \mathcal{A}_1(x)$	$\mathcal{A}_1 : \mathcal{X} \to \mathcal{Y}$
$z := \mathcal{A}(x)$	$z := \mathcal{A}_2(x)$	$\mathcal{A}_2 : \mathcal{X} \to \mathcal{Z}$
$z := \mathcal{A}^f(x)$	$z := \mathcal{A}_3^f(x)$	$\mathcal{A}_3^f : \mathcal{X} \to \mathcal{Z}$
$\dots$	$\dots$	$\dots$

Table 2.8: An example of difference between adversarial routines.

Adversaries and oracle calls. We can think of an adversary as a program that is built to play against a fixed game. However, the adversary is probabilistic and stateful, i.e., it has memory of previous interactions. Although we use a single symbol $\mathcal{A}$ for each adversary call, we actually have a collection of adversarial routines which share the memory. The routine selection is determined by the type of the input, type of the expected output and the list of functions $\mathcal{A}$ is allowed to access during the call. Therefore, we can define the adversary for an infinite number of types. This is illustrated by the example in Table 2.8, where the adversarial routines from the first column are specified by the corresponding routines in the second and third column. In Table 2.8 x is of type $\mathcal{X}$, y is of type $\mathcal{Y}$ and z is of type $\mathcal{Z}$. In this example, adversarial routine $\mathcal{A}_2$ differs from the routine $\mathcal{A}_3$, as the latter is allowed to access a function `F` during the call. To illustrate the difference between adversarial routines we will use the game $\mathcal{G}$, where the adversary is a collection of routines that share the memory. In this game, the adversary has access to the encryption oracle denoted by EO and to the decryption oracle denoted by DO and has to decide if c is an encryption of m_0 or m_1. The adversary outputs one if it decides that c is an encryption of m_0 and outputs zero if it decides that c is an encryption of m_1. The difference between the adversarial routines is emphasized by giving the types of the adversarial routines next to their usage in the game $\mathcal{G}$.

```
EO($x$)
    $y := Enc(x, sk)$
    return $(y)$

DO($y$)
    $x := Dec(y, sk)$
    return $(x)$

$\mathcal{G}$
    $sk \leftarrow$ Gen
    $(m_0, m_1) \leftarrow \mathcal{A}_1^{EO,DO}()$        $\mathcal{A}_1^{EO,DO} : \emptyset \to \mathcal{X} \times \mathcal{X}$
    $c \leftarrow Enc(m_0, sk)$
    return $\mathcal{A}_2^{EO}(c)$                   $\mathcal{A}_2^{EO} : \mathcal{Y} \to \{0,1\}$
```

Adversary can be specified in several ways in the ProveIt language, it can have zero or more arguments, zero or more oracles and an optional type of the adversary. The adversary is denoted by $\backslash adv()$ in the ProveIt language. The example syntax for different types of adversaries is given in Table 2.9.

Type of adversary	**Syntax example**	**Rendered example**
Adversary without arguments	$\backslash adv()$	$\mathcal{A}()$
Adversary with arguments	$\backslash adv(a, b)$	$\mathcal{A}(a, b)$
Adversary with oracles and arguments	$\backslash adv\hat{}\{f, g\}(a, b)$	$\mathcal{A}^{f,g}(a, b)$
Adversary with oracles, type and arguments	$\backslash adv\_c\hat{}\{f, g\}(a, b)$	$\mathcal{A}_c^{f,g}(a, b)$

Table 2.9: Syntax examples for different types of adversaries.

Chapter 3

Semantics of ProveIt language

The ProveIt language is a simple imperative language. Like any other language it has semantics. Its semantics is very close to the semantics of the While language with some exceptions that are explained below.

We define a specific semantics for ProveIt language. Namely, our goal is to define the semantics in a manner that allows it to be used in a concrete-security framework, where the adversary has tight time limits. To describe semantics of ProveIt language, we have to introduce the state, stack, list of variables `Var` and list of potential values `Val`. The list of variables `Var` contains the variables used in the program and list of potential values `Val` denotes the range of values that can be assigned to the variables described by `Var`. The names and values of the variables are pushed to the stack during the execution of the program, i.e., the stack is used to handle the global and local variables. The state describes the values of variables at a given point in the execution of a program. When the execution changes a value of a variable, then the state of the program is changed. The changes in the state can be used to analyze the program.

For deterministic languages, the state is defined to be a deterministic function $\sigma : \texttt{Var} \to \texttt{Val}$, that assigns values to the variables. In ProveIt language, the state is changed probabilistically as the language contains random choice, e.g., the uniform choice operator assigns a value uniformly from a given set. The modified version of the While language which formalizes randomized languages is called pWhile or probabilistic While language. In pWhile, the state can be formalized in two equivalent ways.

In the first formalization we define the state as a function $\sigma_1 : \texttt{Var} \times \Omega \to \texttt{Val}$, where Ω is the randomness. The reasoning behind this formalization is to derandomize the program. For that, an extra input $\omega \in \Omega$ is introduced that determines the exact values of the random choices. With this formalization we could define semantics that depends on the randomness. However, it is difficult to formalize the way how randomness is used.

For the second formalization, we need to define the distribution of variable values. *A distribution* is list of all possible variable values where each evaluation of variables has a probability that shows how likely it happens. The sum of the probabilities has to be equal to one. The distribution can be used to model variables that have randomized values. An example distribution of variable values is shown in Table 3.1.

Variable values	$a = 5$ $b = 2$	$a = 3$ $b = 4$	$a = 4$ $b = 7$
Probabilities	$\frac{1}{3}$	$\frac{1}{2}$	$\frac{1}{6}$

Table 3.1: An example that illustrates the distribution of variable values.

The second conceptual way to formalize probabilistic semantics is to consider a tree of all potential random choices. For that, we describe the execution of a program with a tree of states. This gives us small-step semantics as the tree describes all states of the program. A node in this tree fixes the values of the variables for the corresponding program point. Therefore, the node contains a table of variables with the corresponding evaluations. Besides, a node contains the program counter (PC) that tells which statement is executed next. There are directed edges between the nodes that point to the potential follow-up states, where the program can go after the execution of the next instruction. Each edge is labeled with a probability that shows how likely the execution of the program goes to the corresponding node. The root node of the tree denotes the beginning of the program. The program counter of the root node is set to zero and the none of the variables are evaluated. In each execution step, the program counter is increased and new nodes containing the evaluated variables are added to the tree. The probability of the execution reaching a certain state can be found by multiplying the probabilities of the edges that lead to the corresponding node. As the execution is randomized the tree must describe all possible program points where the execution can go. Thus, if a node is not a leaf node, then it has edges to the child nodes. It is important to note that the probabilities of child nodes have to sum up to one, i.e., all execution paths are described by the tree. The leaf nodes denote the termination of the program and therefore they determine the output distribution of the program. Now we see that the constructed tree describes all states and thus corresponds to the small-step semantics, i.e., it describes the behavior of the program during the execution. An illustration of the tree-based semantics is depicted on the Figure 3.1.

We can get another version of the tree based small-step semantics if we group the states with the same program counter value into a set that resembles a distribution. This set of variable values does not always create a distribution as the probabilities of the states with the same program counter may not sum to one. An example of a tree semantics that generates this kind of a set is given on the Figure 3.1. Compared to the tree-based semantics, this version of the small-step semantics is less precise as some information is lost.

By computing only the distribution of the leaf nodes we can define the big-step semantics. Big-step semantics takes the input and gives the output distribution. Note that the big-step semantics does not describe all states and therefore it contains less information than the small-step semantics.

A:

```
$b := 1$
$x := 0$
while$(b = 1)$
    $x := x + 1$
    $b \leftarrow \{0, 1\}$
return$(x)$
```

B:

PC=0
b:=? x:=?

. . .

PC=3	PC=3	PC=3	. . .
$b := 1$ $x := 1$	$b := 1$ $x := 2$	$b := 1$ $x := 3$	. . .
1	$\frac{1}{2}$	$\frac{1}{4}$	. . .

C:

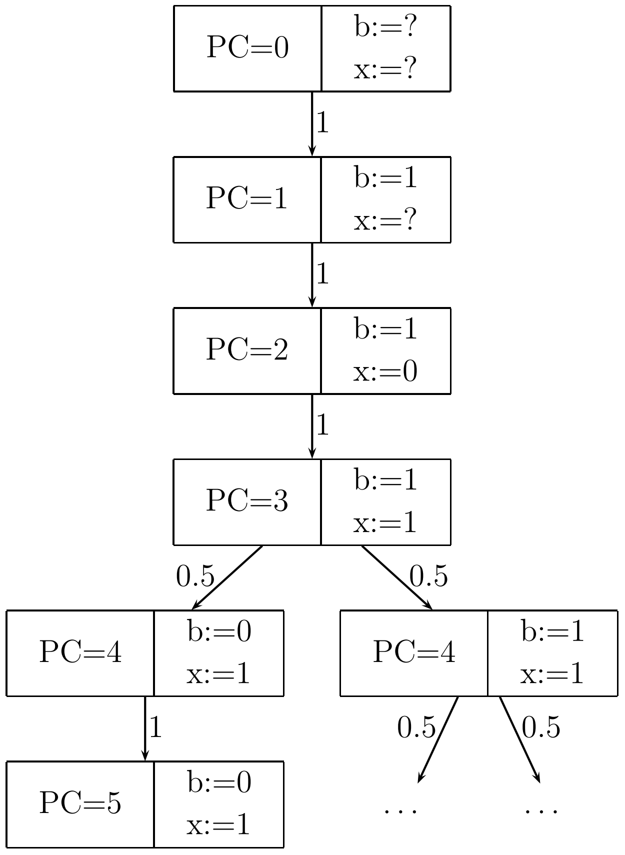


Figure 3.1: **A:** Code that can go to an infinite loop due to the probabilistic statement. The **B** and **C** part use this code. **B:** The sets of states can be used to define the second type of semantics. Note that the set of states with PC=3 does not create a distribution. **C:** The tree-based semantics of the code from **A**.

The following section will give the tree semantics for ProveIt language, i.e., we will describe how to build the state tree based on the code. For doing that, we will introduce the types that are used in defining the semantics. These types are described in Table 3.2. We will use small-step semantics and operational semantics

Type	**Meaning**
`Var`	Variables used in the program
`Val`	Range of variable values
`Time`	Used time
`Tvalue`	Truth value, either zero or one
`Exp`	Arithmetic expression
`Bexp`	Logical expression
`State`	State of the program
`Fname`	Function name
`Fargs`	Function arguments
`Astate`	State of the adversary

Table 3.2: The types that are used in defining the semantics for ProveIt.

to describe how the computations are performed on the computer. Operational semantics specifies how and it which order to execute a block of code. Therefore, the operational semantics specifies how the execution paths are constructed for the previously described tree. The execution paths are relevant as they define the behavior of the program. For more information about all types of semantics, see [NN92].

3.1 Interpretation of execution

ProveIt language models the interaction with the adversary in a defined environment that is commonly depicted as a game between the challenger and the adversary. In this game, the adversarial code is hidden and only the calls to the adversary are visible. As both the challenger and the adversary are programs, they both have a stack for variables, a program counter and a time limit. The global variables are held in the bottom of the stack and the topmost stack frame contains the names and values of the local variables that are currently being used. Each time a new function is called, we add a new local state to the stack. The last element of the stack is removed when one exits a function via the `return` call.

The semantics will describe how the code is executed, how the variables and time are handled and how the adversary is executed. The description of the challenger and adversary is depicted in the Figure 3.2.

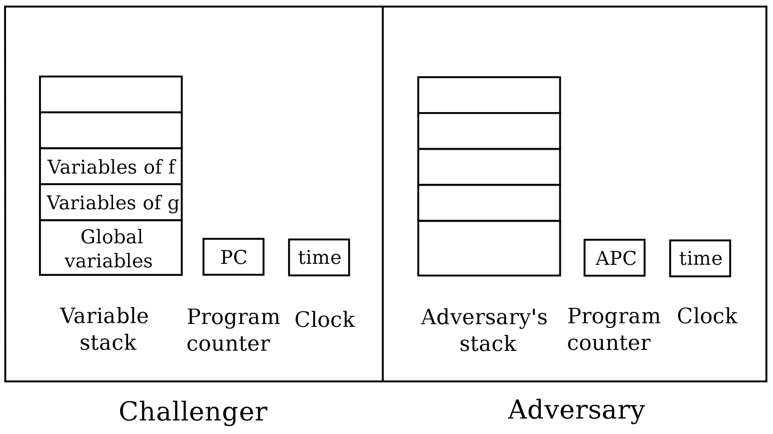


Figure 3.2: Interpretation of execution in the ProveIt language.

3.1.1 Store and its operations

Variables. To execute a program, we need to store the values of global and local variables and the internal state of the adversary. The global variables are stored in the stack frame formalized as a function σ_0 that maps global variable names to their values. To store the local variables, we keep a stack of local states $\sigma_1, \ldots, \sigma_k$ where each local state σ_i maps local variables of function f to their values provided that the i^{th} function call is to the function f. Each time a new function is called, a new local state σ_{k+1} is added to the stack. The last element of the stack is removed when one exits a function via the call `return`. We use $\sigma_g = (\sigma_0, \sigma_1, \ldots, \sigma_k)$ to denote both the global and local variables.

States. The state of the program is defined by the state σ_g and the state of the adversary. Adversarial state is a function σ_a that maps local variables defined by the adversary to the corresponding values. Since the code of the adversary is not visible to us, we treat σ_a as a vector where elements are added, removed or modified. In other terms, we do not know the names of the adversary's variables. The complete state of the program is given by store σ that is a tuple $\sigma = (\sigma_g, \sigma_a)$. To define the semantics with the help of store, we need to describe how we can query and modify the variables.

It is not possible to access adversary's variables and therefore we can not query them. For the challenger's variables we define querying the value of a variable x by

$$\sigma(x) = \begin{cases} \sigma_0(x), & \text{if } x \text{ is a global variable} \\ \sigma_k(x), & \text{if } x \text{ is a local variable} \\ \perp, & \text{if } x \text{ is not in the store.} \end{cases}$$

Due to the previously described scoping rules, the local variables of a called function cannot be accessed by other functions. The local variables of a function call are pushed to the topmost stack frame and therefore it is only possible to query the local variables of the last function call, i.e., from the top of the stack.

To update a value of variable in the store, we write $\sigma[x \mapsto a]$ which means that we replace the old value of x with a. For global variables, the variable name is looked up in the stack and the corresponding value is updated:

$$\sigma[x \mapsto a](x) = \sigma_g[x \to a](x) = \sigma_g(x) = a\,.$$

To update the local variables we behave in the same way as the scoping rules forbid the local variable names to match a global variable name:

$$\sigma[x \mapsto a](x) = \sigma_g[x \to a](x) = \sigma_g(x) = a\,.$$

When a function is called, a new stack frame has to be added to the store. Let f be the corresponding function with arguments $x_1, \ldots, x_n$. Then $\sigma[f\!\downarrow a_1, ..., a_n]$ denotes adding a new stack frame to the state of the program, i.e.,

$$\sigma_{k+1}[x_1 \mapsto a_1, \ldots, x_n \mapsto a_n],$$

where $a_1, \ldots, a_n$ are the argument values of the function f. Recall that the arguments of the function are local variables.

Let $\sigma[\mathcal{A}\downarrow a_1, ..., a_n]$ denote calling $\mathcal{A}$ with new arguments $a_1, ..., a_n$. By our convention, $\sigma[\mathcal{A}\!\downarrow a_1, ..., a_n]$ extends the state of the adversary σ_a by adding $a_1, ..., a_n$ to the end of σ_a.

Let $\sigma[\uparrow]$ denote the deletion of the topmost stack frame of the program. We do now know the contents of the vector σ_a and we are not able to query it. Therefore, we do not define this operation for the adversary as only the adversary itself can delete its information.

Timing. The time resource can be limited for both the challenger and the adversary. Let L_g denote the time limit of the challenger and L_a the time limit of the adversary. Due to this, we assign a specific variable t_a to count the time usage of the adversarial functions. Besides that, let t_g denote the time used for executing all commands that do not include adversary, i.e., the time usage of the challenger. In the beginning of the execution, these variables are evaluated to zero and each execution step and adversarial step increases the time count of the corresponding variables. In addition, let t denote $t = (t_g, t_a, b)$, where b is a bit that shows whose time is being counted. By using the bit b we can increase the time count with one rule. For that, we write

$$t \triangleleft \delta = \begin{cases} (t_g + \delta, t_a, b) & \text{if } b = 0, \\ (t_g, t_a + \delta, b) & \text{if } b = 1\,, \end{cases}$$

where δ denotes the used time. To start counting the time of the challenger we write $t[b \to 0]$ or $(t + \delta)[b \to 0]$ and to start counting the time of the adversary we write $t[b \to 1]$ or $(t + \delta)[b \to 1]$ if δ denotes the increase in the time count. Therefore, we count the time usage of one party at a time. The game is started by the challenger and therefore initially $b = 0$ and $t = (0, 0, 0)$.

Base semantics. We do not define the entire semantics of the ProveIt language. In particular, we do not define the semantics for arithmetic expressions as this could be easily derived. Therefore, we assume that the functions for evaluating expressions and counting the time usage of the expressions are already defined. Let the letter $\mathcal{S}$ denote the semantics function that takes the syntactic construction and state as input and outputs the result of executing the expression on the given state. Besides that, let the letter $\mathcal{T}$ denote the timing function that takes the syntactic construction and state as input and outputs the time that is used for executing the expression on the given state.

Besides that, we do not define the semantics of the adversary as the behavior of adversary is unknown. We can only observe the semantics of the adversary. I.e., the semantics is given to us by functions $\mathcal{S}_a$, $\mathcal{T}_a$ and $\mathcal{P}_a$. We do not know how these functions work but we can use them to describe the semantics. Therefore, we use the name external semantics do denote the semantics that is given by these functions. The first function $\mathcal{S}_a$ gets the state of the adversary and an index as input and outputs a follow-up state and return values or a follow-up state and a function call, where the follow-up state corresponds to the index. The second function $\mathcal{T}_a$ gets the state of the adversary and an index as input and outputs the time usage that correspond to the ith follow-up state given by the function $\mathcal{S}_a$ on the same input. The third function $\mathcal{P}_a$ gets the state of the adversary and index as input and outputs the probability that corresponds to the ith follow-up state given by the function $\mathcal{S}_a$ on the same input.

3.2 Semantics of expressions

An expression consists of operations on variables, constants or other expressions. Therefore, an expression does not contain a function call, instead a variable can be evaluated by a function call and the corresponding variable used in the expression. As an exception, we also consider a constant as an expression. We describe the type of the semantics of arithmetic expressions as

$$\mathcal{S} : \texttt{Exp} \rightarrow (\texttt{State} \rightarrow \texttt{Val})\,.$$

E.g., if we have an expression `e` and a state σ_g, then the arithmetic expression `e` on state σ_g is written as $\mathcal{S}[\![\mathtt{e}]\!](\sigma_g)$.

However, we need to keep track of the execution time and for that we use the previously described timing function

$$\mathcal{T} : \texttt{Exp} \rightarrow (\texttt{State} \rightarrow \texttt{Time})\,.$$

The time used for executing the arithmetic expression `e` in state σ_g is written as $\mathcal{T}[\![\mathtt{e}]\!](\sigma_g)$. This function gets the arithmetic expression as input and outputs the time that is used for executing it.

We describe the type of the semantics of logical expressions as

$$\mathcal{S} : \texttt{Bexp} \rightarrow (\texttt{State} \rightarrow \texttt{Tvalue})\,,$$

where `Tvalue` is a truth value. E.g., if we have a logical expression `e` and a state σ_g, then the logical expression `e` on state σ_g is written as $\mathcal{S}[\![\mathtt{e}]\!](\sigma_g)$.

As with the arithmetic expressions, we need to keep track of the execution time. Similarly, we use the timing function

$$\mathcal{T} : \texttt{BExp} \rightarrow (\texttt{State} \rightarrow \texttt{Time})\,,$$

that gets the logical expression as input and outputs the time that is used for executing it. The time used for executing the logical expression e in state σ_g is written as $\mathcal{T}[\![\mathsf{e}]\!](\sigma_g)$.

3.3 Notation of the semantics

To denote a rule in semantics, we write the initial configuration followed by the follow-up configuration. The initial configuration contains the code block of interest and the state and the follow-up configuration contains the results of the rule. We denote the configurations by enclosing them in angle brackets. In the following sections, we will use the letter P to denote the part of the program that has not been executed. An example rule that assigns a value could be denoted by

$$\langle x := a; P, \sigma, t\rangle \to \langle P, \sigma[x \mapsto a], t^*\rangle \,,$$

where $x := a$ is the command of interest, $t^* = t \triangleleft 1$ and P denotes the rest of the program. If there are no other commands to execute, then P is empty.

Also note that the previous example show that the commands are separated by a semicolon.

We can modify the notation to add more information to the rule. We divide the notation of the rule into two parts and separate them by a horizontal line. The lower part corresponds to the rule and the upper part corresponds to the conditions that must be fulfilled to apply the rule. This is illustrated by the following example

$$\frac{t^* = t \triangleleft 1}{\langle x := a; P, \sigma, t\rangle \to \langle P, \sigma[x \mapsto a], t^*\rangle}.$$

In the following, we put the conditions to the upper part of the rule. However, if there are too many conditions then we will write some of them after the rule. In addition, if the rule is probabilistic, i.e., the rule is applied with a probability less than one, then we mark the corresponding probability on the arrow of the rule. The rules that are applied with probability one do not contain the probability on the arrow for reasons of clarity.

3.4 Semantics of simple steps

Assignment. The assignment of an expression affects one global or local variable x by assigning the value a of the expression e to it. In the following, the letter P denotes the program that is not yet executed. The semantics of assigning a single value is described by the rule ASSIGN

$$\frac{a = \mathcal{S}[\![e]\!](\sigma_g), \quad t^* = t \lhd (\mathcal{T}[\![e]\!](\sigma_g) + 1)}{\langle x := e; P, \sigma, t\rangle \to \langle P, \sigma[x \mapsto a], t^*\rangle} \quad ,$$

where t^* is the new time count after executing the expression and assigning to x.

The assignment of multiple values affects global or local variables $x_1, \dots, x_n$ by assigning values $a_1, \dots, a_n$ of the corresponding expressions $e_1, \dots, e_n$ to them. The assignment of multiple values is described by the rule VASSIGN

$$\frac{\forall i \in \{1, \dots, n\} : a_i = \mathcal{S}[\![e_i]\!](\sigma_g), \quad t^* = t \lhd (\sum_{i=1}^{n} \mathcal{T}[\![e_i]\!](\sigma_g) + n)}{\langle (x_1, \dots, x_n) := (e_1, \dots, e_n); P, \sigma, t\rangle \to \langle P, \sigma[x_1 \mapsto a_1, \dots, x_n \mapsto a_n], t^*\rangle},$$

where t^* denotes the time usage after the assignment of multiple values.

Uniform choice. The uniform choice will uniformly assign a value from the given set $\mathcal{X}$ to the variable x. To measure the time usage of uniform choice we are given a function $\mathcal{T}[\![x \leftarrow X]\!] : \emptyset \to \texttt{Time}$, that outputs how much time is used by performing uniform choice. The semantics of uniform choice is described by the rule UNIFORMCHOICE

$$\frac{\forall a \in \mathcal{X}, \quad t^* = t \lhd \mathcal{T}[\![x \leftarrow \mathcal{X}]\!], \quad p = \frac{1}{|\mathcal{X}|}}{\langle x \leftarrow \mathcal{X}; P, \sigma, t\rangle \underset{p}{\to} \langle P, \sigma[x \mapsto a], t^*\rangle} \quad ,$$

which splits the state into $|\mathcal{X}|$ different follow-up states, where each follow-up state is accessed with the same probability. Therefore, the value a of x is a taken uniformly from the set $\mathcal{X}$. The changed t^* denotes the time usage after the uniform choice, including the time used for sampling the value from the set $\mathcal{X}$.

3.5 Semantics of control flow statements

If-then-else block. The semantics of the if statement is given by two rules. The first rule of the semantics expresses the case where the logical expression evaluates to true. This is described by the rule IfTrue

$$\frac{\mathcal{S}[\![b]\!](\sigma_g) = \text{true}, \quad t^* = t \lhd \mathcal{T}[\![b]\!](\sigma_g)}{\langle \mathsf{if}\ b\ \mathsf{then}\ \{S_1\}\ \mathsf{else}\ \{S_2\}\,; P, \sigma, t\rangle \to \langle S_1; P, \sigma, t^*\rangle}.$$

Here, S_1 and S_2 can be code blocks and therefore we enclose them in curly brackets.

The second rule of the semantics expresses the case where the logical expression evaluates to false. This is described by the rule IfFalse

$$\frac{\mathcal{S}[\![b]\!](\sigma_g) = \text{false}, \quad t^* = t \lhd \mathcal{T}[\![b]\!](\sigma_g)}{\langle \mathsf{if}\ b\ \mathsf{then}\ \{S_1\}\ \mathsf{else}\ \{S_2\}\,; P, \sigma, t\rangle \to \langle S_2; P, \sigma, t^*\rangle}.$$

While loop. The semantics of the while statement is given by two rules as the execution depends on the value of the logical expression. The first rule of the semantics, where the logical expression evaluates to true, is described by the rule WhileTrue

$$\frac{\mathcal{S}[\![b]\!](\sigma_g) = \text{true}, \quad t^* = t \lhd \mathcal{T}[\![b]\!](\sigma_g)}{\langle \mathsf{while}(b)\ \{S\}\,; P, \sigma, t\rangle \to \langle S; \mathsf{while}(b)\ \{S\}\,; P, \sigma, t^*\rangle}.$$

The second rule of the semantics, where the logical expression evaluates to false is described by the rule WhileFalse

$$\frac{\mathcal{S}[\![b]\!](\sigma_g) = \text{false}, \quad t^* = t \lhd \mathcal{T}[\![b]\!](\sigma_g)}{\langle \mathsf{while}(b)\ \{S\}\,; P, \sigma, t\rangle \to \langle P, \sigma, t^*\rangle}\ .$$

For loop. The for statement consists of an header and a body. The execution starts from the first assignment in the header and is followed by the execution of the logical expression. The value of the logical expression divides the execution into two. If it evaluates to false then the execution of the for loop is stopped and the execution moves to the next statement. However, if the logical expression evaluates to true then the body of the for statement is executed and then followed by the execution of the second assignment in the header. After that the execution goes back to the logical expression where the next executable statement is decided in the previously described manner. Therefore, we can define the semantics for the for loop by using the semantics previously defined for the while loop. This is described by the rule ForToWhile

$$\langle \mathsf{for}(S_1; b; S_2)\ \{C\}\,; P, \sigma, t\rangle \to \langle S_1; \mathsf{while}(b)\ \{C; S_2\}\,; P, \sigma, t\rangle,$$

where S_1 denotes the first assignment in the header of the for loop, S_2 denotes the second assignment in the header of the for loop and C denotes the body of the for loop.

3.6 Semantics of function calls

Function call. In the semantics of the function call, we allow arguments of a function call to be expressions. For handling the local variables, including the arguments, a new stack frame is created in the beginning of the function call and the stack frame is removed from the stack when the call is completed.

Let $x_1, \dots, x_n$ denote the argument names of the function f and let $e_1, \dots, e_n$ be the corresponding expressions in the function call. Besides that, let $a_1, \dots, a_n$ denote the corresponding values of the expressions. Finally, let $\mathsf{f.body}$ denote the function body of f. Then, the semantics of the function call is described by the rule Fcall

$$\frac{\forall i \in \{1, \dots, n\} : a_i = \mathcal{S}[\![e_i]\!](\sigma_g), \quad t^* = t \lhd (\sum_{i=1}^{n} \mathcal{T}[\![e_i]\!](\sigma_g) + \varepsilon)}{\langle \mathsf{f}(\mathsf{e_1}, \dots, \mathsf{e_n}); P, \sigma, t\rangle \to \langle \mathsf{f.body}; P, \sigma[f\!\downarrow a_1, ..., a_n], t^*\rangle} \quad ,$$

where ε denotes the time that is used to create a new stack frame.

Assignment by a function call. Now, we define semantics for the function call that returns values. For that, we define two rules. The first rule FcallAssign is described by

$$\frac{\forall i \in \{1, \dots, n\} : a_i = \mathcal{S}[\![e_i]\!](\sigma_g),\ t^* = t \lhd (\sum_{i=1}^{n} \mathcal{T}[\![e_i]\!](\sigma_g) + \varepsilon),\ \sigma^* = \sigma[f\!\downarrow a_1, ..., a_n]}{\langle (y_1, \dots, y_\ell) = \mathsf{f}(\mathsf{e_1}, \dots, \mathsf{e_n}); P, \sigma, t\rangle \to \langle \mathsf{f.body}; \mathsf{update}(y_1, \dots, y_\ell); P, \sigma^*, t^*\rangle} \quad ,$$

where ε denotes the time that is used for creating a new stack frame.

The second rule describes the situation when the function body has been evaluated and the return call has created returnable values $b_1, \dots, b_\ell$ and we need to store them as variables $y_1, \dots, y_\ell$. This rule is named Update and it is described by

$$\frac{\sigma^* = \sigma[y_1 \mapsto b_1, \dots, y_\ell \mapsto b_\ell], \quad t^* = t \lhd \ell}{\langle \mathsf{update}(y_1, \dots, y_\ell); P, \sigma, t, (b_1, \dots, b_\ell)\rangle \to \langle P, \sigma^*, t^*\rangle},$$

where t^* denotes the time that is used for evaluating the variables $y_1, \dots, y_\ell$.

Return call at the end. The return call is the last statement of the function and it will output the returnable values. We treat the returnable values $e_1, \ldots, e_\ell$ as expressions. The semantics rule of the return call is named ReturnCall and it is described by

$$\frac{\forall i \in \{1, \ldots, \ell\} : b_i = \mathcal{S}[\![e_i]\!](\sigma_g), \quad t^* = t \triangleleft (\sum_{i=1}^{\ell} \mathcal{T}[\![e_i]\!](\sigma_g) + \varepsilon) \quad \sigma^* = \sigma[\uparrow]}{\langle \mathsf{return}\ (e_1, \ldots, e_\ell); P, \sigma, t\rangle \to \langle P, \sigma^*, t^*, (b_1, ..., b_\ell)\rangle} \quad ,$$

where ε denotes the time that is used for removing the topmost stack frame and $(b_1, ..., b_\ell)$ are the returnable values of the return call.

3.7 Semantics of the adversary calls

External semantics of the adversary. As the exact description of the adversarial behavior is unknown to us, we assume that it is given to us. As we do not know the exact semantics of the adversary, then the following is a description of externally observable semantics of the adversary. There are three main cases to observe: the next case and the action of the adversary, the amount of time needed for this and the probability of this move. For deterministic adversary the first two cases are described by

$$\mathcal{S}_a : \mathtt{Astate} \to (\mathtt{Astate} \times \mathtt{Val}^*) + (\mathtt{Astate} \times \mathtt{Fname} \times \mathtt{Fargs}) \quad ,$$

$$\mathcal{T}_a : \mathtt{Astate} \to \mathtt{Time} \quad ,$$

where `Astate` denotes the state of the adversary and $\mathtt{Val}^*$ denotes zero or more return values, `Fname` denotes the name of the function call and `Fargs` denotes the arguments of the function call. However, if we consider a probabilistic adversary then the external semantics of the adversary must output several values. Therefore, we index next states with natural numbers and write

$$\mathcal{S}_a : \mathtt{Astate} \times \mathbb{N} \to (\mathtt{Astate} \times \mathtt{Val}^*) + (\mathtt{Astate} \times \mathtt{Fname} \times \mathtt{Fargs}) \quad ,$$

$$\mathcal{T}_a : \mathtt{Astate} \times \mathbb{N} \to \mathtt{Time} \quad ,$$

$$\mathcal{P}_a : \mathtt{Astate} \times \mathbb{N} \to [0, 1] \quad ,$$

where $\mathtt{Val}^*$ denotes zero or more values. In these functions, the index determines the follow-up state. The plus sign in the function $\mathcal{S}_a$ means that the state of the adversary goes either to the state `Astate` $\times$ `Val`* or to the state `Astate` $\times$ `Fname` $\times$ `Fargs`. The function $\mathcal{T}_a$ returns the time usage of the adversary for the specified follow-up state and the function $\mathcal{P}_a$ outputs the probability for the follow-up state specified by the index.

With this information we can model the behavior of the adversary as a finite-state machine (FSM). This FSM is depicted on the Figure 3.3, where the adversary is denoted by Adv and the label $\mathcal{S}_a$ denotes the transition function that changes the state of the adversary. The depicted FSM shows that the adversary may return values and call an oracle one or more times.

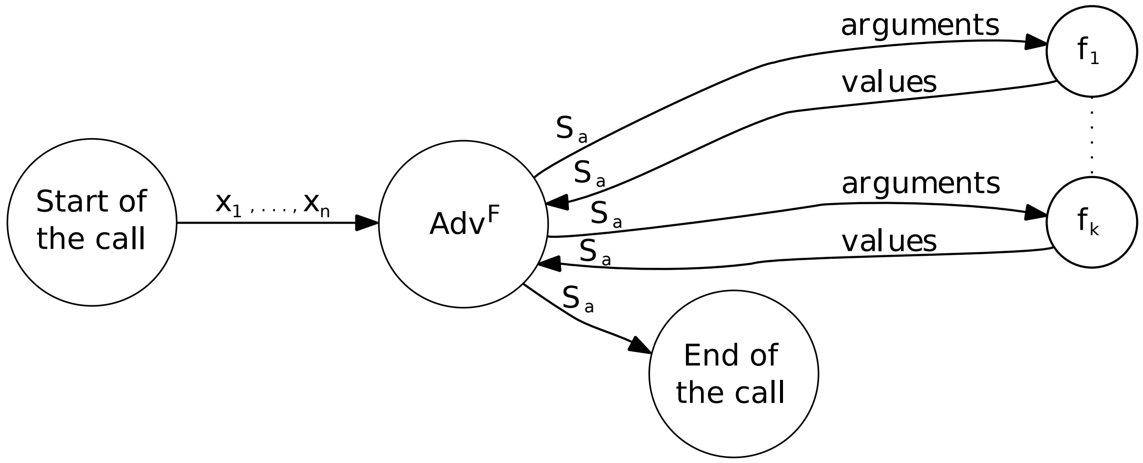


Figure 3.3: A FSM of an adversary with arguments $x_1, \ldots, x_n$ and oracle access to functions $f_1, \ldots, f_k$.

Semantics of the adversary calls. In the following, we denote the oracle functions with F and allow $\mathsf{F} = \emptyset$ or $\mathsf{F} = \{f_1, \ldots, f_n\}$. Therefore, we can denote both types of adversaries $\mathsf{Adv}^{f_1,\ldots,f_n}(e_1, \ldots, e_n)$ and $\mathsf{Adv}(e_1, \ldots, e_n)$ with $\mathsf{Adv}^F(e_1, \ldots, e_n)$.

First we view a call to the adversary that gives the adversary new information. We treat the arguments of the adversary as expressions. As an adversary call uses time of the adversary, the time counting is switched to adversary. The semantics of the adversary call is given by the rule AdvCall and it is described by

$$\frac{i \in \{1, \ldots, n\} : a_i = \mathcal{S}[\![e_i]\!](\sigma_g), \quad t^* = (t \triangleleft (\sum_{i=1}^{n} \mathcal{T}[\![e_i]\!](\sigma_g) + \varepsilon))[b \to 1]}{\langle \mathsf{Adv}^\mathsf{F}_*(e_1, \ldots, e_n); P, \sigma, t\rangle \to \langle \mathsf{Adv}^\mathsf{F}_*; \mathsf{update}(\emptyset); P, \sigma[A\downarrow a_1, \ldots, a_n], t^*\rangle} \ ,$$

where ε denotes the time used for adding values $a_1, \ldots a_n$ to the end of σ_a.

We also need to cover the case where the adversarial routine returns some values. As an adversary call uses time of the adversary, the time counting is switched to adversary. The semantics of assignment by an adversary is given by a rule AdvAssign and it is described by

$$\frac{i \in \{1,\dots,n\} : a_i = \mathcal{S}[\![e_i]\!](\sigma_g), \sigma^* = \sigma[A\!\downarrow a_1,\dots,a_n], t^* = (t \lhd \sum_{i=1}^{n} \mathcal{T}[\![e_i]\!](\sigma))[b \to 1]}{\langle (x_1,\dots,x_n) = \mathsf{Adv}^{\mathsf{F}}(e_1,\dots,e_n); P, \sigma, t\rangle \to \langle \mathsf{Adv}^{\mathsf{F}}; \mathsf{update}(\mathsf{x}_1,\dots,\mathsf{x}_\mathsf{n}); P, \sigma^*, t^*\rangle},$$

where ε denotes the time used for adding $a_1,\dots a_n$ to σ_a and $\sigma^* = (\sigma_g, \sigma_a^*)$.

The third rule describes changing the state of the adversary and returning values. As in this rule the adversary returns values and stops, then the time counting is switched to challenger. To find the next state and the returnable values we use functions $\mathcal{S}_a$ and $\mathcal{P}_a$. To find the used time of the adversary, we use the function $\mathcal{T}_a$. This rule is named EndAdvCall and it is described by

$$\frac{\forall i \in \mathbb{N} : (\sigma_a^*, b_1,\dots,b_n) = \mathcal{S}_a(\sigma_a, i), p_i = \mathcal{P}_a(\sigma_a, i), p_i > 0), t^* = (t \lhd \mathcal{T}_a(\sigma_a, i))[b \to 0]}{\langle \mathsf{Adv}^{\mathsf{F}}; P, \sigma, t\rangle \underset{p_i}{\to} \langle P, \sigma^*, t^*, (b_1,\dots,b_n)\rangle},$$

where $\sigma^* = (\sigma_g, \sigma_a^*)$.

In addition, we have to give a rule that describes choosing the oracle call. This rule is named OracleCall and it is described by

$$\frac{\forall i \in \mathbb{N} : (\sigma_a^*, f, a_1,\dots,a_n) = \mathcal{S}_a(\sigma_a, i), p_i = \mathcal{P}_a(\sigma_a, i), t^* = t \lhd \mathcal{T}_a(\sigma_a, i),\ f \in \mathsf{F}, p_i > 0}{\langle \mathsf{Adv}^{\mathsf{F}}; P, \sigma, t\rangle \underset{p_i}{\to} \langle f(a_1,\dots,a_n); \mathsf{Adv}^{\mathsf{F}}; P, \sigma^*, t^*\rangle},$$

where $\sigma^* = (\sigma_g, \sigma_a^*)$.

If the adversary calls a function that it is not allowed to call then the program is stopped. This rule is named WrongCall and it is described by

$$\frac{\forall i \in \mathbb{N} : (\sigma_a^*, f, a_1,\dots,a_n) = \mathcal{S}_a(\sigma_a, i), \quad t^* = t \lhd \mathcal{T}_a(\sigma_a, i), \quad f \notin \mathsf{F}, \quad p_i > 0}{\langle \mathsf{Adv}^{\mathsf{F}}; P, \sigma, t\rangle \underset{p_i}{\to} \perp},$$

where the symbol $\perp$ denotes that the program is terminated without returning the output value.

We need one final rule to describe the adversary gaining the information from the oracle call. This rule is named InformationUpdate and it is described by

$$\frac{t^* = t \lhd \varepsilon}{\langle \mathsf{Adv}^{F}; P, \sigma, t, (b_1,\dots,b_n)\rangle \to \langle \mathsf{Adv}^{F}; P, \sigma[A\!\downarrow b_1,\dots,b_n], t^*\rangle},$$

where ε denotes the time used for adding $a_1,\dots a_n$ to σ_a.

3.8 Bounded runningtime model

In the previous description of the semantics we did not check the time limits. However, we can still remove the executions that violate the time limit from the semantics. We notice that with the previously described rules we created tree based small-step semantics.

We know that the leaf nodes contain the return statement values of the game. These values are created by the rule ReturnCall and the program does not have any statements left to execute. Therefore, the leaf nodes contain the state, time count and the return value, i.e., $\langle \emptyset, \sigma, t, \texttt{value} \rangle$, where $\emptyset$ denotes an empty program.

Using the querying rule, we can query all the leaf nodes from the semantics tree. Therefore, we create two rules for modifying the leaf node values based on the used time. The first rule is for the case where the challenger and the adversary have not exceeded their time limits and the second rule is for the case when at least one of them has exceeded the time limit. Recall that we use L_a to denote the time limit of the adversary and L_g to denote the time limit of the challenger. The first rule is named OutputValue and it is described by

$$\frac{t_a \leqslant L_a, \quad t_q \leqslant L_q}{\langle \emptyset, \sigma, t, \texttt{value} \rangle \to \texttt{value}.}$$

The second rule is named output bottom and it is described by

$$\frac{t_a > L_a \vee t_q > L_q}{\langle \emptyset, \sigma, t, \texttt{value} \rangle \to \perp.}$$

Thus, by adding these rules to the semantics we can make sure that the result of the game is not revealed when the challenger or the adversary has violated the time limit.

Let $\mathcal{G}^{\mathcal{A}}$ denote a game. There are three possible outcomes for this game: 0, 1 or $\perp$. In addition, $\Pr[\mathcal{G}^{\mathcal{A}} = 1]$ counts the total probability of the leaves returning a value 1.

Chapter 4

Techniques for static program analysis

Control flow of a program describes how and in which order a program is executed. A control flow graph allows to visualize all possible execution paths of a program, i.e., the sequence of commands that are carried out after the initialization of the program. Based on the control flow graph we can build liveness analysis, that checks which commands can be removed from the program without changing the end result of the program. Therefore, the control flow graph can be used for analyzing the program and making the execution more efficient.

4.1 Control flow

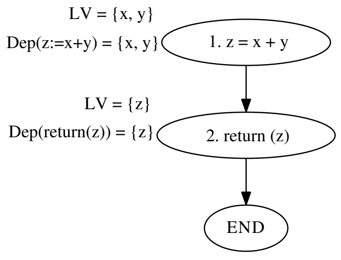
A *control flow graph* (CFG) consists of statements of a program and the description of how the control of the execution can flow between the statements. The vertices of the graph are the statements. An edge from a vertex s_1 to vertex s_2 means that the execution of statement in s_2 can directly follow the execution of statement in s_1. Control flow graphs are mostly used for deriving facts about execution traces.

We use the term execution trace for describing the execution path of a program. An execution trace corresponds to the path from the root node to the leaf node in the semantics tree. Therefore, all execution traces are given by the semantics tree. An example game and the corresponding execution trace is depicted on the Figure 3.1 in Chapter 3.

From the definition, we see that the all execution traces can be built from the control flow graph. Namely, we can do it by executing the statements and evaluating random choices with all possibilities. Using this method, we can build and link the program states. This is done rigorously by the formal semantics, i.e., the CFG shows how the rules of semantics are applied. If we add probabilities to the links that connect the table nodes then we have built all execution traces of a program, i.e., the tree-based description of the semantics.

Our goal is to create the CFG for games written in the ProveIt language as then we can do the liveness analysis. A game in the ProveIt language is like the main function in programming language C, except that we can also specify function signatures inside the game. As the function signature, such as $f : M \to C$, is not a command, it is not included in the CFG. However, we can create a CFG for function definition body, as the function body contains a sequence of commands followed by a return statement. We create a separate CFG for each function and do not include it in the main CFG. We use both types of CFG in the liveness analysis. As an illustration of the concept, we depict a game $\mathcal{G}_2$ and the corresponding control flow graph on the Figure 4.1. On this figure, the control flow graph nodes are labeled with LV and Dep. These labels denote the set of alive variables and the set of dependent variables. These terms are explained in section 4.3. The labels on Figure 4.1 illustrate the liveness analysis on these control flow graphs.

```
$f(x, y)$
    $z := x + y$
    return $(z)$
```



```
$\mathcal{G}_2$
    $f : K \times M \to C$
    $sk \leftarrow K$
    $m \leftarrow M$
    $c := f(sk, m)$
    $d := \mathcal{A}(c)$
    if$(c = d)$
        $sk \leftarrow K$
        return $(0)$
    else
        return $(1)$
```

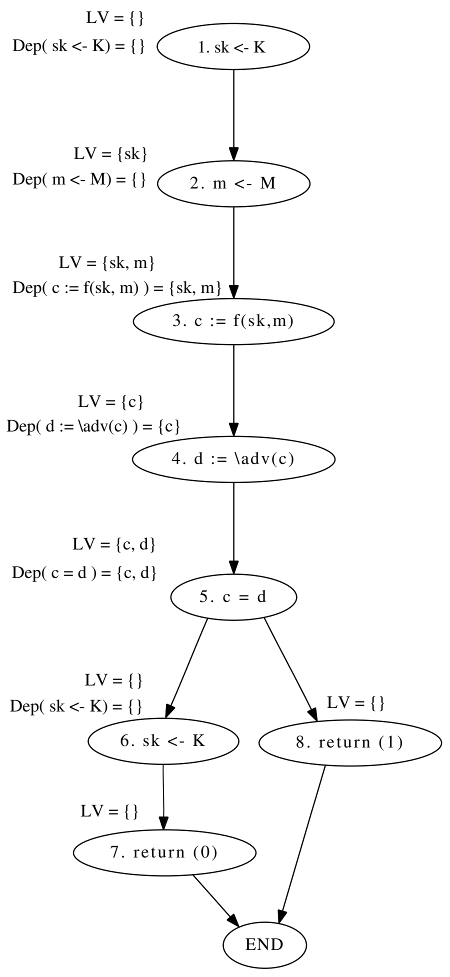


Figure 4.1: An example of how the control flows in function f and in the game $\mathcal{G}_2$.

4.2 From abstract syntax tree to control flow graph

In this section, we describe how the control flow graph is created from the abstract syntax tree (AST) of ProveIt. We start by traversing the AST using depth first search in order to create the CFG nodes and label them in the depth first order. In each recursion step, we check the type of the AST node and based on that create a node for the CFG. After that the CFG node is added to a list of CFG nodes.

A CFG node has to contain a statement or a logical expression from the corresponding AST node. Instead of copying the information from AST, we link the CFG nodes with the corresponding AST nodes. We also add empty fields *LV* and *required* to the CFG node, these field are used by the liveness analysis. In addition, the CFG node has to contain a list of parent nodes, a list of child nodes, an unique label and the block depth. The block depth is a number that states how deep the corresponding statement is nested, e.g., the statements that are not nested have the block depth one. The node corresponding to the first statement of the AST has always the block depth equal to one. The block depth value is used in the liveness analysis for handling the nested statements. An illustration for the block depth is depicted on the Figure 4.2.

```
$\mathcal{G}$
$m := 1$                  $BD = 1$
$b \leftarrow \{0,1\}$    $BD = 1$
if$(b\ =\ 1)$             $BD = 1$
    $m := m + 1$          $BD = 2$
    $b \leftarrow \{0,1\}$  $BD = 2$
    if$(b\ =\ 1)$         $BD = 2$
        $m := m + 1$      $BD = 3$
return $(m)$              $BD = 1$
```

Figure 4.2: The statements have been tagged with the block depth (BD).

The unique label is a number that corresponds to the found AST node number, e.g., if the statement is a seventh found AST statement, then CFG node will be labeled with 7. In the following, each time a tree node is created the corresponding label is added to it. The unique labels are used to link the CFG nodes together. Thus, the list of parent nodes and the list of child nodes actually contain the unique labels of these nodes. Due to the labeling method, some of the child and parent labels are added to a CFG node after the depth first search has found and labeled them. By the time the depth first search is completed, all CFG nodes have been added to a list of CFG nodes. This list is used to link the CFG nodes to create the CFG. A single CFG node is depicted on the Figure 4.3.

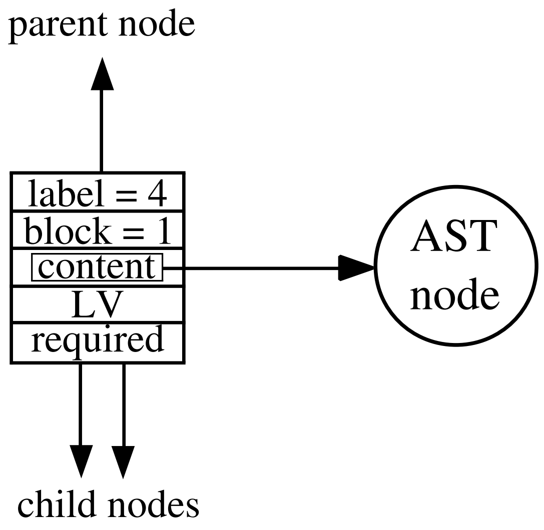


Figure 4.3: A single CFG node, where LV denotes the set of alive variables.

Simple statements. If the AST node is a function signature, then we ignore it and move to the next node. Otherwise, if the AST node is an assignment, uniform choice, function call or a call to the adversary then we add to the CFG node a link to the corresponding statement, i.e., a link to the AST node. In addition, we add the label of the direct ancestor node to the list of parent nodes. The child node labels are added in the next recursion step, i.e., in each recursion step we add the current tree node label to the parent tree node. Handling of the CFG nodes with more than one parent or child node is described in the next paragraph.

Finally, we set the block depth for the CFG node. If the AST statement is not nested inside an if statement, for statement or while statement, then we set block depth to be one. Otherwise, we set block depth of the CFG node to the block depth of the innermost conditional statement.

Block statements. Besides simple statements, we still have to describe what we do with the if statements, while statements and for statements. We refer to these statements as block statements as they contain other statements within them. Therefore, we should not add the block statements to the CFG and instead add the statements and operations that are contained within these statements. Besides that, if this type of a statement is nested inside another if statement, while statement or a for statement, then the condition node of the nested statement has the block depth set to the block depth of the condition node of the innermost statement.

For the if statement, we create a CFG node for the condition and add the label of the parent node to the list of parent node labels. The if statement contains one or two statement blocks based on structure of the if statement and therefore the condition node has at least two child nodes. To add all the child node labels to a condition node, we have to wait until both statement list are recursively traversed by the depth first search. We can add the next child label when the statement block has been traversed. Besides that, we have to remember the labels of the nodes that correspond to the last statement in the statement lists and use these labels to link the node that follows the if statement. To illustrate the structure of the CFG that corresponds to an if statement, a simplified CFG is depicted on the Figure 4.4.

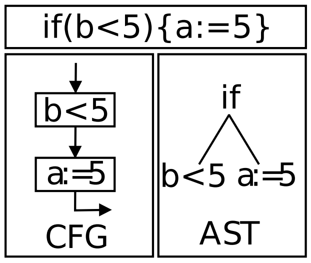


Figure 4.4: A simplified AST node and a simplified CFG of the if statement.

The for statement is handled in a similar manner. The statement itself is ignored and the CFG node corresponding to the first assignment in the header is added to the list of CFG nodes. After that the CFG node corresponding to the condition is added to the list of CFG nodes. Then the statement block of the for statement is recursively traversed and the node corresponding to the last statement of the block is linked with the node that corresponds to the second statement in the header of the for statement. Then the node corresponding to the second statement of the header is linked with the condition node. As a final step the condition node is linked with the node that is created after the traversal of the while statement. To illustrate the structure of the CFG that corresponds to a for statement, a simplified CFG is depicted on the Figure 4.5.

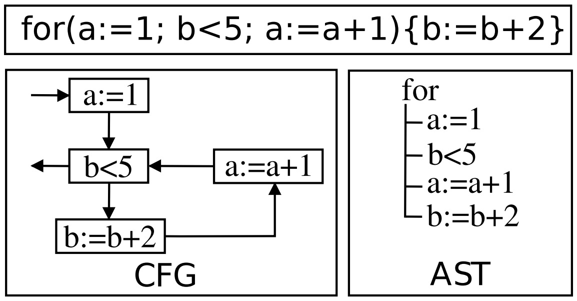


Figure 4.5: A simplified AST node and a simplified CFG of the for statement.

The while statement is handled in a similar manner to the for statement. First the condition node is added to the list of CFG nodes and then the statement block is recursively traversed. The node corresponding to the last statement in the statement block is linked to the condition node and the condition node is linked with the node that is created after the traversal of the while statement. To illustrate the structure of the CFG that corresponds to a while statement, a simplified CFG is depicted on the Figure 4.6.

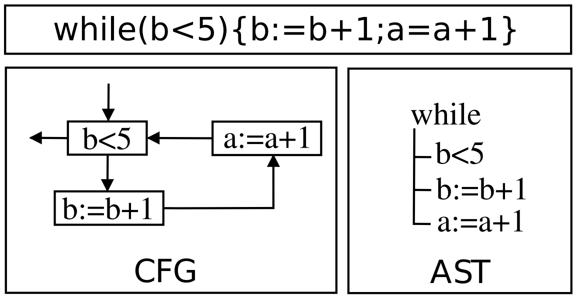


Figure 4.6: A simplified AST node and a simplified CFG of the while statement.

Finalization. With this information we can create the control flow graph. For that, we connect the CFG nodes by using the saved information about the parent and child nodes. Due to this construction, we know for each node where the control can go and from where it could have come from.

Implementation. The control flow graph generation is implemented by a function named `createControlFlowGraph`, which takes an AST node, block depth and parent label as input. The algorithm is started on the root node of the AST with arguments block depth equal to one and parent label equal to minus one. As a result, the algorithm fills a vector with uniquely labeled CFG nodes where the CFG nodes are connected to their child and parent nodes.

4.3 Liveness analysis

4.3.1 Preliminaries

We use the liveness analysis to create a game transformation for ProveIt that removes the redundant statements from the game. To describe the liveness analysis, we need to define the liveness of a variable and the meaning of dependent variables.

A variable is *alive* in a vertex of the control flow graph if there is a path from the vertex to a return vertex such that returnable value depends on the value of the variable. If an alive variable is evaluated then all the variables that are used for the evaluation can affect the return result and are therefore set alive. Removing the alive variable could change the probability distribution of the output. A variable is *dead* in the state `s` of the semantics tree if the variable is not read by any successor state of `s`, where a successor state is a node in the semantics tree that is reachable from the state `s`. If a variable is dead in the CFG then it is also dead in the semantics tree but the converse does not have to hold. If an alive variable is evaluated, then it is set to be dead as the previous statements in the AST contain this variable with a different value and this value might not be used to find the return value.

We use the CFG to find the alive variables in the liveness analysis. In particular, we find the dependent variables of statements that evaluate an alive variable, as then the returnable value depends on these variables. We know that the CFG describes the semantics tree and therefore we may do the analysis using the CFG. However, before describing the algorithm of the liveness analysis, we give a definition for the dependent variables and describe the rules for finding the dependent variables of different statements.

A *dependent variable* of a statement is a variable that is used in the execution of that statement. We denote the dependent variables of a statement `s` by `Dep(s)`. There are three types of variables a statement can depend on: variables in expressions, function arguments and global variables read by a function. Therefore, we define functions `Var`, `Args` and `GRead` for finding the dependent variables. Function `Var` takes an expression as input and outputs the variables used in the expression. Recall that we do not allow function calls to be present in expressions. Additionally, in assignment of a variable we consider the variable to be an expression and therefore the function `Var` also handles variables. Function `Args` takes a function call or an adversary call as input and outputs the arguments of the function call or the arguments of the adversary call. Function `GRead` takes a function name as input and outputs the global variables that are read by statements of the corresponding function body.

Assignment. It is easy to see that for assignment $x := e$ the set of dependent variables is described by the rule

$$\texttt{Dep}(x := e) \ = \ \{\texttt{Var}(e)\}\,,$$

where $\texttt{Var}(e)$ denotes the variables used in the expression e. The dependent variables of binary operations are described by the rule

$$\texttt{Var}(e_1 \ \texttt{op} \ e_2) \ = \ \texttt{Var}(e_1) \cup \texttt{Var}(e_2),$$

where `op` denotes a binary operation between expressions e_1 and e_2. The dependent variables of unary operations are described by the rule

$$\texttt{Var}(\texttt{op} \ e) \ = \ \texttt{Var}(e),$$

where `op` denotes a unary operation on an expression e.

Uniform choice. The statement uniform choice does not have dependent variables, i.e.,

$$\texttt{Dep}(a \leftarrow Z) \ = \ \{\}\,.$$

Function call. The dependent variables of a function call and the dependent variables of an assignment by a function call are described by the following rule

$$\texttt{Dep}(f(k_1, \ldots, k_n)) = \texttt{Dep}(y := f(k_1, \ldots, k_m)) \ = \ \texttt{Args}(f(x)) \cup \texttt{GRead}(f),$$

where $\texttt{Args}(f(x))$ returns the arguments of the function f and $\texttt{GRead}(f)$ returns the global variables read by the function f. This rule is correct if the arguments of the function are not expressions. The rule is limited in this way due to the current implementation of ProveIt.

Assignment by an adversary. Finally, the dependent variables of an adversary are described by the rule

$$\mathtt{Dep}(a := \mathcal{A}_c^{f_1,\ldots,f_n}(x)) \;=\; \mathtt{Args}(\mathcal{A}_c^{f_1,\ldots,f_n}(x)) \cup \mathtt{GRead}(f_1) \cup \ldots \cup \mathtt{GRead}(f_n).$$

In addition to this, if a statement containing an adversary is set to be alive, then automatically all statements that contain the adversary are alive. This is caused by the fact that the adversary can save the variables that it has previously read.

An illustration of the alive and dependent variables is given on the Figure 4.1. Now we have the required preliminaries for the liveness analysis and we can start to describe the analysis.

Implementation. Finding the dependent variables of assignments, operations and function calls is implemented by a function `getDependent` that gets a CFG node as an input and outputs a vector of strings that contains the dependent variables.

4.3.2 Description of liveness analysis

To begin with the liveness analysis, we have to build a control flow graph. The idea of the liveness analysis is to traverse the control flow graph and to mark all nodes that contain alive variables. After several traversals all nodes that contain alive variables are found, i.e., all statements that can modify the output of the program are marked. The unmarked nodes correspond to the statements that do not affect the alive variables, i.e., the output of the program. These statements can be removed from the program.

Correctness of liveness analysis. We can view the paths from the beginning of a program to the end of the program as subsets of the execution trace with relation to the states, i.e., the set of states from the root node to the the leaf node. We know that a variable is dead if it is not used in the successive states in the execution trace and we use this property to find the dead variables in the CFG. We have to show that in the liveness analysis we mark only these nodes in the CFG as required which correspond to the states in the execution trace that contain alive variables.

Theorem 1 The liveness analysis described by Algorithm 1, Algorithm 2 and Algorithm 3 correctly identifiers the dead variables.

We will analyze the liveness algorithm for three different types of code blocks. These are the sequential code block, the cycle code block and the conditional code block. The sequential code block contains commands that are executed sequentially. The cycle code block is created by the while statement or by the for statement and the conditional code block is created by the if statement. To analyze these code blocks we denote a set of true alive variables by $\mathtt{LV}^{\circ}$ and the set of found alive variables in CFG node `v` by `v.LV`. The set $\mathtt{LV}^{\circ}$ denotes all alive variables at a given point in the CFG and `v.LV` denotes the set of alive variables found by the Algorithm 1. In addition, we denote that the CFG node `v` is in the set of parent nodes of CFG node `n` by $\{\mathtt{n}\} = \mathtt{ParentNodes(v)}$.

Sequential code block. A node in the sequential code block has exactly one parent node.

Lemma 1 Let there be a CFG node `v` and $\mathtt{LV}^{\circ} = \mathtt{v.LV}$. In addition, let there be a CFG node `n` so that $\{\mathtt{n}\} = \mathtt{ParentNodes(v)}$, i.e., `v` has a single parent node. Then after applying the Algorithm 2 on CFG node `v` the alive variables of CFG node `n` are correctly updated, i.e., $\mathtt{LV}^{\circ} = \mathtt{n.LV}$.

Proof. If a statement corresponding to a CFG node `n` does not evaluate an alive variable, then the algorithm does not find alive variables and therefore $\mathtt{LV}^{\circ} = \mathtt{n.LV}$.

If a statement corresponding to a CFG node `n` evaluates an alive variable, then the variable is set to be dead and the dependent variables of the statement are alive, i.e., $\mathtt{n.LV} := \mathtt{v.LV} \backslash (\mathtt{Write}(n) \cap \mathtt{v.LV}) \cup \mathtt{Dep(n)}$.

In this case Algorithm 2 removes the alive variable from the set of alive variables and adds the dependent variables of the statement corresponding to `n` to `n.LV`. This behavior is described by the steps 19, 14 - 16 in the Algorithm 2. □

Lemma 2 Let there be a CFG node `v` and $\mathtt{LV}^{\circ} = \mathtt{v.LV}$. In addition, let there be only one path to the node `n` from node `v`, i.e. each node in this path has one parent node.

Then after applying the Algorithm 2 on CFG node `v` the alive variables of CFG node `n` are correctly updated, i.e., $\mathtt{LV}^{\circ} = \mathtt{n.LV}$.

Proof. As each CFG node in this path has only one parent then the path corresponds to a sequential code block. Therefore, by sequentially applying **Lemma 1** we get $\mathtt{LV}^{\circ} = \mathtt{LV}$ after the sequential code block.

Algorithm 1 starts the execution of sequential block from the return statement, described by the steps 2-4. Algorithm 2 handles the other sequential blocks, this is described by the steps 14 - 16, 19 and by the steps 2 - 3 in the case of a function call. □

Conditional code block. In the condition block the algorithm enters the block from the end and moves upwards towards the condition. There are two paths from the end of the condition block to the condition node, the if path with the else path or the if path and a direct path from the condition to the next statement.

Lemma 3 Let after the conditional code block $\texttt{LV}^\circ$ = `v.LV`, i.e., the parent of `v` is the end node of the condition block. Let `n` be a CFG node before the if condition. In addition, let there be two paths form `v` to `n`. Then by applying Algorithm 2 on the node `v`, the set of alive variables is correctly updated and before the condition node $\texttt{LV}^\circ$ =`n.LV`.

Proof. The proof derives from **Lemma 2** as such condition block only increases the length of the sequential code block. □

Lemma 4 Let `v` be a CFG node after the if condition block, $|\texttt{ParentNodes(v)}| = 2$ and $\texttt{v.LV} = \texttt{LV}^\circ$. Let `n` be a CFG node that contains the if condition. In addition, let there be two paths form `v` to `n`. Then by applying Algorithm 2 on the node `v` we get $\texttt{n.LV} = \texttt{LV}^\circ$.

Proof. We see from **Lemma 2** that such condition block can only increase the length of the sequential code block and therefore by applying Algorithm 2 on the sequential code block the set of alive variables is correctly updated. Thus, up to the condition node the set of alive variables is correctly updated.

If in the sequential block an alive variable was found or a statement was set to be required, then the dependent variables of the condition are added to the set of alive variables. This is done by the Algorithm 2 using steps 17, 5 and 8. The sets of alive variables are joined in the condition node. This is done by the Algorithm 2 using the steps starting from 8. Therefore, the alive variables from the two different paths are correctly joined, i.e., the condition node contains the union of the found alive variables.

Finally, we have to show that the nodes before the condition have the correct alive variable sets. This results comes directly from the **Lemma 3** and therefore, we get $\texttt{n.LV} = \texttt{LV}^{\circ}$. □

Cycle code block. In the cycle code block we can have a for cycle or a while cycle. However, we can convert the for cycle into a while cycle and analyze only that. We can do this by moving the first assignment from the header of the for cycle just before the for cycle. The second assignment from the header of the for cycle can be moved to the end of the for cycle body. These are syntactical changes that do not change the semantics.

Now we can start to analyze the behavior of the Algorithm 2. In the case of while cycle we enter the code block from the condition node of the cycle, move to the end of the statement block and then continue by moving up. This is described by the steps 5 and 9 in the Algorithm 2. While moving upwards we check if any statement evaluates the alive variables. If such statement exists then we mark the alive variable as dead, the corresponding CFG node as required and the dependent variables of that node as alive. Besides that, we force the condition of the while cycle to be marked as required. This is described by the steps 14 - 16 in the Algorithm 2. The condition is required as the evaluation of the required statement depends on the condition node. The condition node is set required by the step 17 in the Algorithm 2. In addition, the dependent variables of the condition node have to be marked as alive variables. This is described by the step 8 in the Algorithm 2. The correct block depth marking of nested while conditions is achieved by the steps 12 and 17 in the Algorithm 2.

If alive variables are found in the cycle body then the dependent variables of the condition are added to the set of alive variables in previously described manner. Besides that, the new alive variables can depend on other variables of the cycle body. Therefore, the cycle body may have to be traversed several times until no new alive variables are found. This is described by steps 5 - 12 and in particular by the step 9 in the Algorithm 2.

Lemma 5 The number of required cycle traversals is finite.

Proof. There is a finite number of different variables in the cycle body. Therefore, we can upper bound the number of required cycle traversals by the number of different variables in the cycle body. □

Lemma 6 Algorithm 2 correctly analyses the while cycle.

Proof. By **Lemma 5** we know that a finite number of cycle traversals is required for the analysis. Traversing the cycle is described by steps 5 - 12 and step 21 in the Algorithm 2. The tree has a finite size and therefore the algorithm stops after a finite number of cycle traversals. We proved with **Lemma 4** that the Algorithm 2 correctly analyses conditional code blocks and therefore also correctly analyzes the sequential code blocks. We have to show that we find all alive variables by moving several times through the cycle. In addition, the next paragraph describes how the function calls are analyzed. Algorithm 2 stops entering the while cycle when the step 7 fails. Eventually all traversals fail in the step 7. By that time all alive variables have been written to the condition nodes. Therefore, by the time when all recursions have finished the initial condition node, from where the recursion started, contains all alive variables. Therefore, Algorithm 2 finds all alive variables from the while cycle. □

The function call. To finish the analysis, we have to show that the function call is analyzed correctly and the set of alive variables is updated correctly. Let `n` denote the CFG node with the function call. When a function call is found a new CFG is generated for the corresponding function body. After that another liveness analysis is done to the body of the function but in this case the Algorithm 1 gets the set of found alive variables `n.LV` as input. When the liveness analysis is completed it is checked if and which arguments of the function are in the set of alive variables. If some arguments of the function are alive, then we replace them with the corresponding arguments of the function call. After that the liveness analysis of the function is completed and a set of alive variables $\mathtt{LV}_f$ can be fetched from the first CFG node of the function definition, i.e., the last update is done to the first node of the function body and therefore this node contains the updated set of alive variables. The previous steps are initialized by the step 2 - 3 in Algorithm 2. The CFG generation and liveness analysis is done by the Algorithm 3.

By combining the Algorithm 1, Algorithm 2 and Algorithm 3, all alive variables are found and all required statements of the CFG marked to be required. This is achieved because these algorithms traverse all possible paths from the terminating nodes to the start node.

Algorithm 1: Liveness($\mathsf{G}, \mathsf{V_R}, \mathsf{pLV}$)

Input: G is the control flow graph where each vertex has attributes required, visited, label and block. Initially attributes required and visited are set to zero. $\mathsf{V_R}$ is the set of return nodes, i.e., the vertices in the control flow where the program terminates. pLV is a set of previously found live variables and $\mathsf{pLV} \neq \emptyset$ only when the function body is analyzed.

Other: LV is the list of variables that are considered alive at the current analysis step. $\mathsf{MoveUp(v, RequiredBlock, LV)}$ is a function that checks the liveness of the variables in the parent nodes of the node v.

```
1 for $\mathsf{n} \in \mathsf{V_R}$ do
2     $\mathsf{n.required} := 1$
3     $\mathsf{n.LV} := \mathsf{Dep(n)} \cup \mathsf{pLV}$
4     $\mathsf{moveUp(n, n.block, n.LV)}$
5 end
```

Algorithm 2: MoveUp(v, RequiredBlock, LV)

Input: v is a node from the CFG. It has attributes content, statement, required and block. RequiredBlock is the block depth that is set to be required. LV is the list of variables that are considered alive at the current analysis step, for v.

Other: Write(s) is a function that returns the variables that are rewritten by the statement s.

```
1  for n ∈ ParentNodes(v) do                      /* use all parent nodes of v */
2      if n.statement = FunCall then                    /* check function call */
3          n.LV := checkFunction(n.content, n.LV)   /* n.content is AST node */
4      end
5      else if v.block > n.block && v.block = RequiredBlock then
                                                     /* check the condition node */
6          v.required := 1
7          if $v.LV \neq (n.LV \cup Dep(v) \cup v.LV)$ then newFound := true
                                              /* if true, then v.LV has changed */
8          $v.LV := n.LV \cup Dep(v) \cup v.LV$          /* update condition node */
9          if newFound = true && $v.statement \neq IfCondition$ then
                                                           /* move to the cycle */
               moveUp(v.getFurthestParent, RequiredBlock, v.LV)
10             newFound := false
11         end
12         RequiredBlock := RequiredBlock − 1       /* leave condition node */
13     end
14     else if $LV \cap Write(n) \neq \emptyset$ then         /* sequential block update */
15         n.required := 1
16         $n.LV := v.LV \setminus (Write(n) \cap v.LV) \cup Dep(n) \cup n.LV$            /* update */
17         if n.block > 1 then RequiredBlock := n.block
             /* if true, then condition of current cycle is required */
18     end
19     else $n.LV := n.LV \cup v.LV$; /* update n.LV in sequential block */
20 end
21 moveUp(n, RequiredBlock, n.LV)          /* check the parent nodes of n */
```

Algorithm 3: checkFunction(ASTNode, LV))

Input: ASTNode is an abstract syntax tree node. LV is a set of alive variables.
Other: FindFunctionDefinition(FunName) is a function that gets the function name and searches the function definition node from the AST. FindReturnNode(ASTNode) is a function that outputs a list of return nodes for the given AST node. createControlFlowGraph(ASTNode, block, parentLabel) gets the AST node, block depth and parent label as input and creates the control flow graph. Liveness(G, returnNodes, LV) does the liveness analysis. checkArgs(LV, args1, args2) checks if the arguments of the function call are in the set of alive variables LV and returns the set of alive variables where the alive function arguments are replaced with the corresponding function call arguments.

```
1 fNode := FindFunctionDefinition(ASTNode.getFunction)
2 $V_R$ := FindReturnNode(fNode)
3 G := createControlFlowGraph(fNode, 1, $-1$)
4 Liveness(G, $V_R$, LV)
5 $LV^f$ := G[0].LV
6 $LV^f$ := checkArgs($LV^f$, fNode.args, ASTNode.args)
7 return $LV^f$;
```

Chapter 5

Game based transformations

There are different types of game transformations in ProveIt and they are all implemented by transforming the abstract syntax tree (AST). These reductions can be divided into three categories.

The first category contains game transformations that are based on cryptographic properties. For example, if X_0 and X_1 are indistinguishable distributions, then a uniformly chosen value $x \leftarrow X_0$ can be replaced with $x \leftarrow X_1$ in most cases. The cryptographic transformations can change the probability of a successful attack.

The second category contains game transformations that do not change the end result of the game. These reductions apply syntactical changes to the game. For example, dead code elimination removes the statements from the game that are never used for generating the output. Another example is the case when the position of a statement in the game can be changed without affecting other statements. Syntactic reductions do not change the probability of a successful attack.

The third category contains reductions that decompose the game into several games. For example, a game containing an if condition for equality can be decomposed into two games based on the result of the condition. This creates two branches to the game-based proof sequence. The reductions from this category can change the probability of a successful attack.

The following sections describe the game transformation that we added to ProveIt. We begin each game transformation section by giving an overview of the transformation and explain why the transformation is required. After that, we give the preconditions that are required in order to apply the transformation. Next, we describe the syntactical change that is caused by applying the transformation. If the transformation does not change the output distribution of a game then we say that the initial game and the game after the transformation are equivalent. We use the symbol $\equiv$ to denote this kind of equality between games. After describing the syntactical change we give a security guarantee with a corresponding proof for each transformation. We end each transformation section with the description of the implementation.

5.1 Dead code elimination

A game may contain redundant statements that are not needed to produce the output. We can remove these statements without changing the output distribution of the game. This has two benefits, it simplifies the games written in ProveIt language and makes some of the cryptographic transformations available that were not available before. To remove the redundant statements, we have to do a liveness analysis. For doing that we have to build a control flow graph and run the liveness analysis algorithm on it. The statements that are not marked as required after the liveness analysis can be removed from the game as these statements are not used in producing the output. The game transformation that does the liveness analysis and then removes the redundant nodes is named dead code elimination (DCE). For more information about dead code elimination and liveness analysis, see [NNH99].

Preconditions. There are no preconditions for performing dead code elimination.

Syntactical change. The DCE removes the dead code from the game, i.e., the assignments to dead variables and statements that modify only dead variables.

Security guarantee. Let $\mathcal{G}_0$ be the original game and let $\mathcal{A}$ denote the adversary, then after dead code elimination we get $\mathcal{G}_1 = DCE(\mathcal{G}_0)$ and $\mathcal{G}_0^{\mathcal{A}} \equiv \mathcal{G}_1^{\mathcal{A}}$.

We will prove that the DCE does not change the output distribution of the game. We divide the proof into two lemmas, where the first one says that we can omit the statement that does not modify alive variables and the second one says that the DCE algorithm works correctly. For the first lemma we define $\mathsf{Write}(\mathsf{s})$ as a set of variables that are evaluated by the statement s.

Lemma 7 If $\mathsf{Write}(\mathsf{s})$ consists only of dead variables then omitting the statement s from the AST does not change the program output.

Proof. This follows directly, as no following statements in the trace of the program execution read the dead variables. In other words, there are no alive child statements in the CFG that reads the dead variables. □

Lemma 8 The dead code elimination algorithm that is based on Algorithm 1, Algorithm 2 and Algorithm 3 correctly removes the redundant statements from the game.

Proof. This follows directly from the proof of Theorem 1. After doing the liveness analysis we have marked all required statements and the rest of the statements can be removed. This solution is given by Algorithm 4. □

Implementation description. First the control flow graph is built. As a next step we run the liveness analysis algorithm on the control flow graph. The liveness analysis is described by Algorithm 1 and Algorithm 2 in the section 4.3. The liveness analysis marks for each CFG node if it is required, i.e. if the game output depends from the corresponding statement execution. After the liveness analysis the redundant nodes are removed from the AST. For that, the control flow graph is traversed and for each node that is not required the corresponding statement is removed from the AST. This is done by the Algorithm 4.

Algorithm 4: DCE($\mathsf{AST}, \mathsf{G}, \mathsf{V_R}$)

Input: G is the control flow graph where each vertex has an attribute required. AST is the abstract syntax tree that contains the statements. The veritces of G link to the content of the AST nodes. $\mathsf{V_R}$ is the set of return nodes from the AST.

Other: $\mathsf{RemoveNode}(\mathsf{AST}, \mathsf{v})$ removes the statement that corresponds to the node v from the AST. $\mathsf{Liveness}(\mathsf{G}, \mathsf{V_R}, \mathsf{LV})$ does the liveness analysis on the CFG G.

```
1 $\mathsf{LV} := \mathsf{Liveness}(\mathsf{G}, \mathsf{V_R}, \emptyset)$
2 for $\mathsf{v}$ in $\mathsf{G}$ do
3     if $\mathsf{v.required} \neq 1$ then
4         $\mathsf{RemoveNode}(\mathsf{AST}, \mathsf{v})$
5     end
6 end
```

An example of dead code elimination. An example of a control flow graph and the dead code elimination is shown on Figure 5.1. On this figure the nodes of the control flow are tagged the information about alive variables and dependent variables. These control flow graphs are based on the games depicted on Figure 5.1.

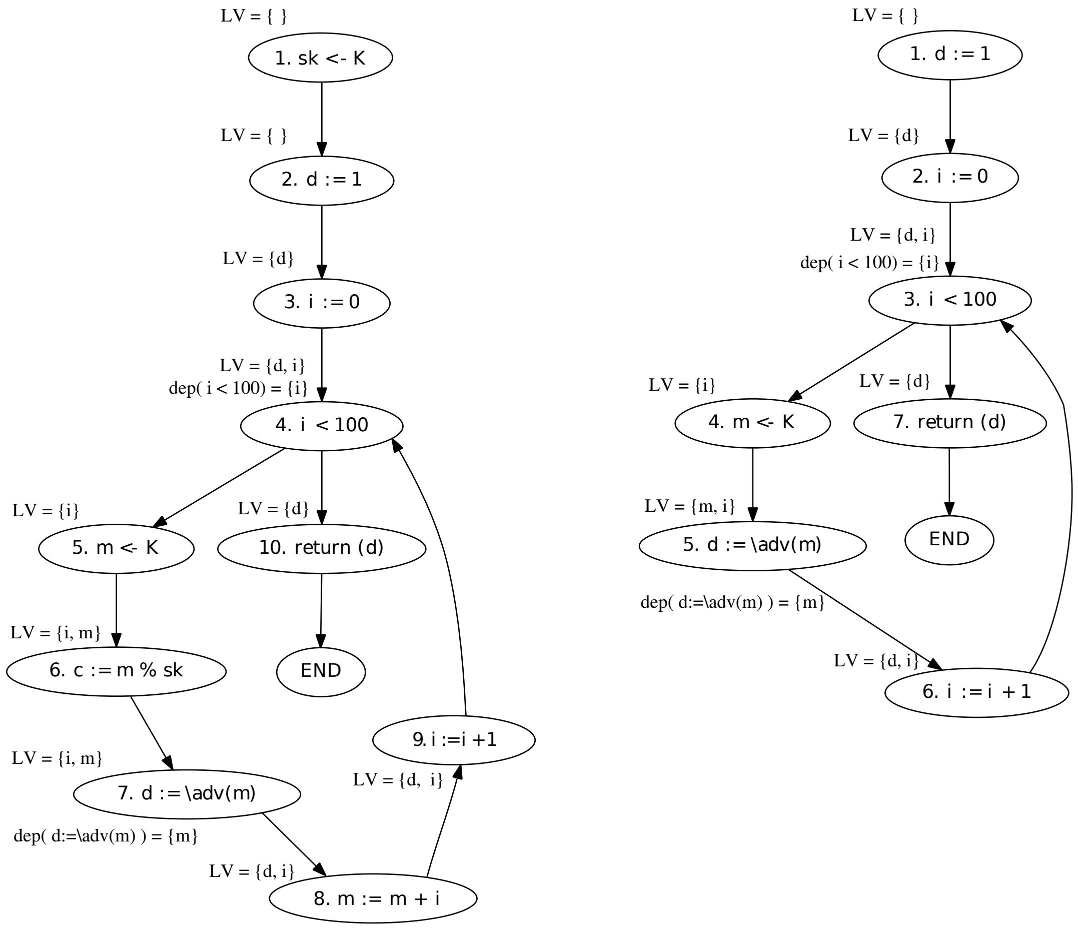


Figure 5.1: CFG of game $\mathcal{G}_0$ from Figure 5.1 before and after the dead code elimination. The CFG of the game $\mathcal{G}_0$ after the dead code elimination corresponds to the game $\mathcal{G}_1$ from Figure 5.1. On these figures `LV` denotes the set of alive variables and `dep(s)` denotes the dependent variables of `s`.

```
$\mathcal{G}_0$
  $sk \leftarrow K$
  $d := 1$
  $i := 0$
  while($i < 100$)
    $m \leftarrow K$
    $c := m\%sk$
    $d := \mathcal{A}(m)$
    $m := m + i$
    $i := i + 1$
  return ($d$)
```

```
$\mathcal{G}_1$
  $d := 1$
  $i := 0$
  while($i < 100$)
    $m \leftarrow K$
    $d := \mathcal{A}(m)$
    $i := i + 1$
  return ($d$)
```

Figure 5.2: The user views of game $\mathcal{G}_0$ before and after the dead code elimination.

5.2 Statement switching

A game might contain a statement that should be moved past another statement in order to apply a game transformation. We do that by using a transformation named Statement Switching (SS), that allows to move a statement after another statement. To start the transformation, the user selects a statement which he wants to move and applies Statement Switching transformation on it. After that the abstract syntax tree (AST) is traversed to check which statements fulfill the conditions required for moving the chosen statement. The user is presented with the list of allowed statements after which the selected statement can be placed. After a successful transformation a copy of the selected statement is inserted after the user chosen statement and the initial statement is removed from the AST.

Preconditions. Statement Switching can only be applied to statements. To move a statement, we have to make sure that the output distribution is not changed. Let the initial statement be denoted by s, and the k^{th} succeeding statement by $\mathsf{s_k}$. We can move the statement s after statement $\mathsf{s_k}$ if the following conditions are fulfilled:

$$\begin{aligned}
\mathsf{Write}(\mathsf{s_i}) \cap \mathsf{Dep}(\mathsf{s}) &= \emptyset \quad \forall i \in \{1, \ldots, k\} \\
\mathsf{Write}(\mathsf{s}) \cap \mathsf{Dep}(\mathsf{s_i}) &= \emptyset \quad \forall i \in \{1, \ldots, k\} \\
\mathsf{Write}(\mathsf{s}) \cap \mathsf{Write}(\mathsf{s_i}) &= \emptyset \quad \forall i \in \{1, \ldots, k\}
\end{aligned}$$

These conditions state that the dependent variables of the initial state s are not rewritten by statements $\mathsf{s_1}, \ldots, \mathsf{s_k}$, the variables evaluated by s are not read by statements $\mathsf{s_1}, \ldots, \mathsf{s_k}$ and that the variables evaluated by s are not evaluated by statements $\mathsf{s_1}, \ldots, \mathsf{s_k}$.

We can think of moving a statement past several statements by doing several swaps between two consequent statements. Therefore, we have to check this rule only for two consequent statements.

Syntactical change. Statement Switching transformation changes the positions of two statements in the game. In the following example, Statement Switching transformation is applied to the function signature in game $\mathcal{G}_1$ in order to place it after the second statement. Game $\mathcal{G}_2$ contains the result of this transformation.

```
𝒢₁                         𝒢₂
  $f : K \times M \to C$       $sk \leftarrow K$
  $sk \leftarrow K$            $f : K \times M \to C$
  $m \leftarrow M$             $m \leftarrow M$
  $c := f(sk, m)$              $c := f(sk, m)$
  return $(c)$                 return $(c)$
```

Security guarantee. Let $\mathcal{G}_0$ be the original game and $\mathcal{G}_1$ the game after the Statement Switching transformation and let $\mathcal{A}$ denote the adversary. Then the output distributions of games $\mathcal{G}_0$ and $\mathcal{G}_1$ are the same, i.e., $\mathcal{G}_0^{\mathcal{A}} \equiv \mathcal{G}_1^{\mathcal{A}}$.

Theorem 2 Statement switching does not change the output distribution of the game.

Proof. It is enough to prove that when the preconditions are fulfilled then moving a statement past the next statement does not change the output distribution of the game. If this is proved then we can do a bigger move by moving the statement one step at a time. This is proved not to change the output distribution of the game if all steps between the two consequent statements follow the preconditions. We denote the two consequent statements by s_1 and s_2 where s is the first statement. We have to show that when all three preconditions hold then we can move s_1 after s_2 without changing the output distribution of the game.

First, we have to check that $\mathsf{Write}(\mathsf{s}_2) \cap \mathsf{Dep}(\mathsf{s}_1) = \emptyset$. If this holds, then after moving s_1 past s_2 the dependent variables of s_1 have the same values as they had before the move. Therefore, if s_1 evaluates variables, then these variables are evaluated in the same way after the move.

Second, we have to check that $\mathsf{Write}(\mathsf{s}_1) \cap \mathsf{Dep}(\mathsf{s}_2) = \emptyset$. If this holds, then after moving s_1 past s_2 the variables evaluated by s_2 have the same values as they had before the move. Therefore, if s_2 evaluates variables, then these variables are evaluated in the same way after the move.

Third, we have to check that $\mathsf{Write}(\mathsf{s}_1) \cap \mathsf{Write}(\mathsf{s}_2) = \emptyset$. If this and the previous preconditions hold, then after moving s_1 past s_2 the variables evaluated by s_2 have the same values as they had before the move. Therefore, if s_2 evaluates variables, then these variables are not rewritten by s_1 after the move. It is evident that when the previous preconditions hold, then the variables evaluated by statement s_1 have the same values after the move as they had before the move.

We have shown that if all three preconditions hold, then moving s_1 after s_2 does not change the output of s_1 and s_2. Therefore, the output values of other statements are not changed by moving s_1 after s_2. Thus, this kind of move does not change the output distribution of the game.

Now that we have proved the base case we can sequentially move a statement when the preconditions of each move are fulfilled. □

Implementation details. We implemented the algorithm by using the idea of performing sequential transformations for two consequent statements. The preconditions are checked in the previously described manner and if the conditions are fulfilled then the statement is moved in the AST. More precisely, before doing the move all preconditions of the base steps are checked. This is done on a copy of the AST.

For doing the precondition checks the dependent variables of the two consequent statements are found and in addition the variables evaluated by these statement are found. Based on this information it is trivial to check the preconditions.

When the statement s is a function call then the corresponding function body is traversed to find the sets $\mathsf{Dep}(\mathsf{s})$ and $\mathsf{Write}(\mathsf{s})$. The same holds for the statement s that contains an adversary that has access to functions, all of these functions have to be checked to find the sets $\mathsf{Dep}(\mathsf{s})$ and $\mathsf{Write}(\mathsf{s})$.

5.3 Remove condition

Remove condition (RC) transformation will split a game into two games based on the condition of the if clause. The transformation begins with the user choosing which branch of the if condition to use. Based on the user's choice two new games are created, one for the user's choice and one complementary game with the opposite logical expression value. The idea of the transformation is to use only a single branch of the if statement. Before applying the transformation we know the probability for the condition to succeed. Therefore, after removing the condition and selecting an execution path, the two different executions paths differ by the probability of the condition to succeed. For example, if a game $\mathcal{G}$ contains an if condition that succeeds with probability 0.5 then both execution paths are equally probable.

Preconditions. This transformation can be applied only to an if statement. This if statement must not be nested inside other statements.

Syntactical change. Remove condition transformation removes the if condition on which the transformation is applied and replaces it with one of the child branches of the if condition.

In the following we give two examples for the Remove Condition transformation, in the first example the else branch is removed and in the second example the if branch is removed. For both examples two games are created to illustrate the difference between the execution probabilities. We use the game $\mathcal{G}$ for both examples. The first example is given by the games $\mathcal{G}_0$, $\mathcal{G}_1$ and the second example by games $\mathcal{G}_2$, $\mathcal{G}_3$.

```
$\mathcal{G}$
  $g \leftarrow N$
  $m \leftarrow N$
  if$(g = m)$
    $x := m + 1$
  else
    $x := g + 1$
  return $(x)$
```

```
$\mathcal{G}_0$
  $g \leftarrow N$
  $m \leftarrow N$
  $x := m + 1$
  return $(x)$
```

```
$\mathcal{G}_2$
  $g \leftarrow N$
  $m \leftarrow N$
  $x := g + 1$
  return $(x)$
```

```
$\mathcal{G}_1$
  $g \leftarrow N$
  $m \leftarrow N$
  return $(\neg(g = m))$
```

```
$\mathcal{G}_3$
  $g \leftarrow N$
  $m \leftarrow N$
  return $(g = m)$
```

Security guarantee. We can separate the game $\mathcal{G}$ containing the if condition into two games by selecting the branch which we want to remove. In the first case the if branch of the condition is used and the else branch is removed and in the other case the else branch of the condition is used and the if branch is removed. Let these be two games, game $\mathcal{G}_0$ in which the if part of the condition is used and let the game $\mathcal{G}_1$ be the complementary game. This is depicted on the Figure 5.3.

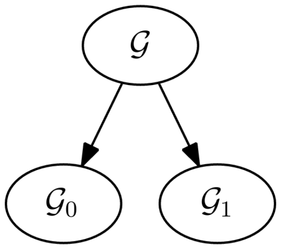

$$\begin{tikzcd} & \mathcal{G} \arrow[dl] \arrow[dr] & \\ \mathcal{G}_0 & & \mathcal{G}_1 \end{tikzcd}$$

Figure 5.3: Branching the game $\mathcal{G}$ based on the result of the condition.

Games $\mathcal{G}_0$ and $\mathcal{G}_1$ differ by the result of the evaluated condition. We can view an overall success probability of an adversary $\mathcal{A}$ in playing against game $\mathcal{G}$. Let the logical expression of the if condition be denoted by `b`.

Theorem 3 By applying Remove Condition transformation we can estimate the success of an adversary $\mathcal{A}$ against the game $\mathcal{G}$ by

$$\Pr[\mathcal{G}^{\mathcal{A}} = 1] \leqslant \Pr[\mathcal{G}_0^{\mathcal{A}} = 1] + \Pr[\mathcal{G}_1^{\mathcal{A}} = 1] \; .$$

Proof. This derives directly from the definition of the transformation and the semantics. □

Implementation details. The Remove Condition transformation splits a game by creating two new games. The old game is used as a basis for building the new games. If the precondition for the transformation holds and we know the user's choice, then a new game is created based on the contents of the remaining if branch. For that, the old if condition is replaced by the corresponding branch. The second game has the same contents up to the point of the condition and after that it returns the corresponding logical expression from the if condition. If the user chose to keep the if branch, then the second game returns the negation of the logical expression. If the user chose to keep the else branch, then the second game returns the logical expression.

5.4 Replace function call

Replace Function Call transformation replaces a function call with the definition of the corresponding function. The transformation takes the contents of the function body and copies them to the place where the function call was made. The transformation is started by selecting the function call that will be replaced. We denote Replace Function Call transformation with RFC.

Preconditions. The transformation has to be done on an assignment statement which contains a function call. The game has to contain the corresponding function definition. In addition, the names of the local variables inside the function body have to differ from the global variable names. If these preconditions are fulfilled, then the Replace Function Call transformation can be applied.

Syntactical change. The statement containing the function call is replaced by the modified body of the function definition. In the copied body of the function definition the arguments of the function definition are replaced with the arguments of the function call. Besides that, every return node in the copied function definition body is replaced by an assignment node that contains the variable from the transformed statement and assigns to it the corresponding arguments of the return node. In addition, if there are variable conflicts, i.e., some of the local variables have the same name as the global variables, then the user has to manually solve the conflict by replacing the corresponding local variable names.

In the following example, Replace Function Call transformation is applied on the statement $y = f(c)$ in the game $\mathcal{G}_0$. The resulting game is described by game $\mathcal{G}_1$.

$\mathcal{G}_0$

```
fun $f(x)$
    $d \leftarrow G$
    $b := d + x$
    return$(b)$
$y := f(c)$
return $(y)$
```

$\mathcal{G}_1$

```
fun $f(x)$
    $d \leftarrow G$
    $b := d + x$
    return$(b)$
$d \leftarrow G$
$b := d + c$
$y := b$
return $(y)$
```

Security guarantee. If we are given an adversary $\mathcal{A}$ and an arbitrary game $\mathcal{G}_0$ that contains a proper function call statement, then after applying RFC on game $\mathcal{G}_0$ we get game $\mathcal{G}_1$ and $\mathcal{G}_0^{\mathcal{A}} \equiv \mathcal{G}_1^{\mathcal{A}}$.

Theorem 4 RFC transformation does not change the output distribution of the game if the game is written in ProveIt language.

Proof. The proof follows directly from the semantics of the function call. The semantics of the ProveIt language describes that a function call that evaluates a variable executes the corresponding function with the given arguments. As a result of the function call, the return value is assigned to the corresponding variable. □

Implementation details. We need to insert new nodes into the AST and for that we create a copy the function definition body. In the following steps we refer to the copied node and modify it so that the function call statement could be replaced with the modified function definition body.

First, the arguments of the function call and function definition are copied into temporary variables. Then the arguments used in the body of the function definition are replaced with the corresponding arguments of the function call. As a last step the return nodes have to be replaced. This is required as the return nodes assign a value to a variable, in our case the variable(variables) used in the function call statement. Therefore, we replace the return nodes with the assignment nodes that use the variable(variables) from the function call and values from the return nodes.

The implementation does not contain a method for solving the variable conflicts. Therefore, the variable conflicts have to be solved manually before applying the transformation.

After these steps the copy of the function definition body is of the correct form and we can use it to replace the function call statement without changing the execution trace.

5.5 Wrap

Wrap transformation reduces code by wrapping some statements into the adversary. After a Wrap transformation the adversary contains additional information and evaluates additional variables. The statement that is wrapped into the adversary is removed from the game.

To apply the transformation a statement that contains an adversarial routine has to be selected. After applying the Wrap transformation on the adversarial routine the user is allowed to choose which statement to wrap into the adversary.

In the following, game $\mathcal{G}_1$ shows the result of applying the wrap transformation on the statement $c := \mathcal{A}(x)$ in game $\mathcal{G}_0$ and wrapping the statement $m := n + 1$.

```
$\mathcal{G}_0$
    $x \leftarrow X$
    $n \leftarrow \mathcal{A}$
    $m := n + 1$
    $c := \mathcal{A}(x)$
    return $(c)$
```

```
$\mathcal{G}_1$
    $x \leftarrow X$
    $n \leftarrow \mathcal{A}$
    $(c, m) := \mathcal{A}(n, x)$
    return $(c)$
```

Preconditions. In order to start the transformation a statement has to be selected by the user. The transformation can be applied in the following cases:

- the selected statement contains an adversary,
- the wrappable statement is an assignment or a uniform choice,
- the wrappable statement can be moved next to the adversary by using Statement Switching transformation.

We need to check only the first two conditions and we can use Statement Switching transformation to check the third one. If the third precondition holds then we can move the wrappable statement before the selected statement. Therefore, we need to analyze only the cases where there are no other statements between the wrappable statement and the selected statement that contains the adversary.

Syntactical change. The wrap transformation is intended to remove the statements that create temporary variables used by the adversary. This transformation removes the wrappable statement from the game and makes the selected adversary evaluate the variable that is initially evaluated by the wrappable statement. In addition, the dependent variables of the wrappable statement are added to the argument list of the selected adversary.

Security guarantee. We can describe two types of wrap transformations, a basic transformation and an advanced transformation. The basic transformation can be applied only for the two consequent statements, while the advanced transformation does not set restrictions to the position of the wrappable statement. The advanced transformation uses the Statement Switching transformation on the wrappable statement to move it next to the selected adversarial statement. Thus, the advanced transformation is based on Statement Switching transformation and the basic wrap transformation.

From the previous description we see that our version of the wrap transformation corresponds to the advanced transformation. Therefore, the security guarantee of the Wrap transformation depends on the security guarantee of the Statement Switching transformation and the security guarantee of the basic wrap transformation.

Theorem 5 Basic wrap transformation does not limit the view of the adversary.

Proof. The preconditions of the basic wrap transformation make sure that the wrappable statement can not be read by other statements before the adversarial routine is called. The adversary evaluates the same variable(s) as the wrappable statement and therefore other statements that read the variable(s) stay functional.

However, wrapping a statement may cause the adversary to change its output as the adversary might have gotten extra information. The basic wrap transformation can give extra inputs to adversary and based on that the adversary could recompute its output values. Therefore, the adversary does not loose information by applying a basic wrap transformation and may instead gain some information that could increase its success probability. $\square$

Therefore, if we are given an adversary $\mathcal{A}$ and an arbitrary game $\mathcal{G}_0$ with statements that follow the preconditions, then after successfully applying the Wrap transformation we get $\mathcal{G}_1 = WRAP(\mathcal{G}_0)$ and $Pr[\mathcal{G}_0^{\mathcal{A}} = 1] \leqslant Pr[\mathcal{G}_1^{\mathcal{A}} = 1]$.

Implementation details. First, we check that the selected statement contains an adversary and that the wrappable statement is an assignment or a uniform choice.

After that we use the Statement Switching transformation to test if the wrappable statement can be moved next to the selected statement. The test is done on a copy of the AST. Thereafter, the dependent variables of the wrappable statement are saved into a temporary variable. In additon, the name of the variable that the wrappable statement evaluates is saved into another temporary variable. After that the wrappable statement is removed from the AST. To finish the transformation the adversary has to get the information from the wrappable statement and evaluate the same variable as the wrappable statement did. For that, we check if the adversary already evaluates this variable and if it does not then we add this variable to the output of the adversary. In addition, we check which of the dependent variables of the wrappable statement are not among the adversary's arguments and include these variables to the adversary's arguments.

Chapter 6

Random function simulation

We are able to simulate a random function of the form $f \leftarrow \{f : M \rightarrow N\}$ by replacing a call to it by a value taken uniformly from the range N. By querying a random functions with uniformly chosen arguments the returnable values are uniformly distributed. Therefore, the return value can be replaced with an uniformly chosen value from the same range. This can be done without changing the distribution of queried values if the queried values are unique. If they are not unique then the two different queries with the same argument will probably give different results. Let the game transformation that simulates the random function be denoted by RFS.

6.1 General transformation

The idea behind the RFS transformation is to build a simulator that models the behavior of a random function. For that, the function definition for random function simulation is built and used to replace the function signature of a random function. After that, the function calls can be replaced with the contents of the new function, i.e. with the contents of the simulator.

The function definition for the random function simulator has one argument and contains an if statement to check if the argument is in a collision set. This has to be checked to know if the adversary has already queried the random function with the given argument. If it has, then the adversary would notice the change while querying the same argument and thus the adversary could win and the simulation would not be valid. Therefore, we need to record the event when the adversary learns too much and for that we use a boolean variable *SimFailure*. Thus, if the condition in the function definition does not fail, i.e., the adversary queries the same value twice, then we set *SimFailure* := 1 and return one.

The other branch of the if statement simulates the case when the adversary makes an unique query to the function. In this case a value is uniformly chosen from the range set of the random function, added to the collision set and returned.

Now we give an example of a RFS transformation applied on the statement $f \leftarrow \{f : M \rightarrow N\}$ in the game $\mathcal{G}_0$. Game $\mathcal{G}_1$ shows the result of applying the transformation. Note that the end result of the game does depend on the value of *SimFailure*.

```
$\mathcal{G}_0$
    $SimFailure := 0$
    $f \leftarrow \{f : M \rightarrow N\}$
    $c \leftarrow \mathcal{A}$
    $d := f(c)$
    return $(d < 5 \vee SimFailure = 1)$
```

```
$\mathcal{G}_1$
    $SimFailure := 0$
    fun$f(x)$
        if$(x \in CS)$
            $SimFailure := 1$
            return$(1)$
        else
            $y \leftarrow N$
            $CS := CS \cup \{y\}$
            return $(y)$
    $c \leftarrow \mathcal{A}$
    $d := f(c)$
    return $(d < 5 \vee SimFailure = 1)$
```

Preconditions. To start the transformation the user has to select the function definition of the proper form. The transformation can be applied in the following case:

- the function in the selected statement has to be of the form $f \leftarrow \{f : M \rightarrow N\}$,
- there has to be a function call for the random function containing one argument.

Syntactical change. The signature of the random function is replaced with a function definition that simulates the random function.

The function definition has one argument and the function body contains an if statement. The if condition checks whether the argument is in the collision set and if it is then it sets *SimFailure:=1* and returns one. In the else branch, a value is uniformly taken from the range of the previously defined random function. The value is added to the collision set and then returned.

If the user chooses to replace a function call then the corresponding statement is replaced by the body of the newly constructed function definition by using the RFC transformation. A replacement of at least one function call creates an empty collision set to the game.

Security guarantee. We want to find the security guarantee for the RFS. Let $\mathcal{G}_0$ be an arbitrary game that contains statements that fulfill the precondition of the random function simulation. Let $\mathcal{G}_1$ be the result of successfully applying random function simulation to the game $\mathcal{G}_0$, i.e. $\mathcal{G}_1 = RFS(\mathcal{G}_0)$. Then the advantage of an adversary $\mathcal{B}$ in distinguishing games $\mathcal{G}_0$ and $\mathcal{G}_1$ depends on the value of the variable *SimFailure* denoted in the following by SF.

Theorem 6 The difference of between probabilities $\Pr[\mathcal{G}_0^{\mathcal{A}} = 1]$ and $\Pr[\mathcal{G}_1^{\mathcal{A}} = 1]$ is bounded by $\Pr[SF = 1]$, i.e., $\left|\Pr\left[\mathcal{G}_0^{\mathcal{A}} = 1\right] - \Pr\left[\mathcal{G}_1^{\mathcal{A}} = 1\right]\right| \leqslant \Pr[SF = 1]$.

Proof. The probability of an adversary $\mathcal{A}$ returning one in the game $\mathcal{G}_1$, is denoted by $\Pr[\mathcal{G}_1^{\mathcal{A}} = 1]$. Game $\mathcal{G}_1$ contains the boolean variable *SimFailure* and therefore we can split the total probability $\Pr[\mathcal{G}_1^{\mathcal{A}} = 1]$ into two parts based on the value of *SimFailure* that is denoted by SF,

$$\Pr[\mathcal{G}_1^{\mathcal{A}} = 1] = \Pr[\mathcal{G}_1^{\mathcal{A}} = 1 \wedge SF = 0] + \Pr[\mathcal{G}_1^{\mathcal{A}} = 1 \wedge SF = 1]\,.$$

The probability of an adversary $\mathcal{A}$ returning one in the game $\mathcal{G}_0$ is denoted by $\Pr[\mathcal{G}_0^{\mathcal{A}} = 1]$. In this game the adversary may have queried the same value twice. Let $QT = 1$ denote that a value is queried twice and let $QT = 0$ denote that all values are queried once. We can split the total probability $\Pr[\mathcal{G}_0^{\mathcal{A}} = 1]$ into two parts by using the QT value,

$$\Pr[\mathcal{G}_0^{\mathcal{A}} = 1] = \Pr[\mathcal{G}_0^{\mathcal{A}} = 1 \wedge QT = 0] + \Pr[\mathcal{G}_0^{\mathcal{A}} = 1 \wedge QT = 1]\,.$$

Now we see that $\left|\Pr\left[\mathcal{G}_0^{\mathcal{A}} = 1\right] - \Pr\left[\mathcal{G}_1^{\mathcal{A}} = 1\right]\right|$ is described by

$$\begin{aligned}\left|\Pr\left[\mathcal{G}_0^{\mathcal{A}} = 1\right] - \Pr\left[\mathcal{G}_1^{\mathcal{A}} = 1\right]\right| = |\Pr[\mathcal{G}_1^{\mathcal{A}} = 1 \wedge SF = 0] + \Pr[\mathcal{G}_1^{\mathcal{A}} = 1 \wedge SF = 1] - \\ - \Pr[\mathcal{G}_0^{\mathcal{A}} = 1 \wedge QT = 0] - \Pr[\mathcal{G}_0^{\mathcal{A}} = 1 \wedge QT = 1]|.\end{aligned}$$

We notice that $\Pr[\mathcal{G}_0^{\mathcal{A}} = 1 \wedge QT = 0] = \Pr[\mathcal{G}_1^{\mathcal{A}} = 1 \wedge SF = 0]$ as no values are queried twice in game $\mathcal{G}_1$, i.e., the values of the queries come from the same set and are uniformly distributed. Therefore, we are left with

$$\left|\Pr\left[\mathcal{G}_0^{\mathcal{A}} = 1\right] - \Pr\left[\mathcal{G}_1^{\mathcal{A}} = 1\right]\right| = |\Pr[\mathcal{G}_1^{\mathcal{A}} = 1 \wedge SF = 1] - \Pr[\mathcal{G}_0^{\mathcal{A}} = 1 \wedge QT = 1]|.$$

We notice that we can upper bound

$$\left|\Pr[\mathcal{G}_1^{\mathcal{A}} = 1 \wedge SF = 1] - \Pr[\mathcal{G}_0^{\mathcal{A}} = 1 \wedge QT = 1]\right| \leqslant \Pr[SF = 1]\ .$$

and therefore $\left|\Pr\left[\mathcal{G}_0^{\mathcal{A}} = 1\right] - \Pr\left[\mathcal{G}_1^{\mathcal{A}} = 1\right]\right| \leqslant \Pr[SF = 1]$.

□

Implementation details. To create the function definition we have to build the corresponding statements. When the function definition is built then we replace the function signature with the created function definition in the AST. In addition, we have to add the assignment statement *SimFailure:=0* to the beginning of the game and therefore we create a corresponding node and add it to the AST. We allow the user to replace function calls, this is implemented by calling the RFC transformation. If all function calls are replaced then the function definition is removed from the AST.

Chapter 7

Conclusion

It is difficult to do the game-based proofs as in each proof step a new game has to be written. Manual rewriting can cause several problems, namely no security guarantee is given, all reduction schema on which the proof steps are based have to be proved. In addition, manual game rewriting allows mistakes to be made. Still, game-based proving is an intuitive way to prove the security of symmetric primitives. Therefore, a tool is required that would give security guarantees for reduction schema and would not allow mistakes during the game rewritings. Thus, the improvement and development of such tools is important to the researchers.

As a result of this thesis, I improved a proof assistant tool ProveIt that is used for game-based proving. The language used in ProveIt did not have semantics and therefore we gave a small-step semantics for ProveIt. Based on the semantics, we proved the security of several reduction schema and implemented the corresponding game transformations in ProveIt. We analyzed and implemented the following game transformations: Dead Code Elimination, Statement Switching, Remove Condition, Replace Function Call, Wrap and Random Function Simulation. These transformations use the abstract syntax tree of ProveIt to modify the security games and are used in reduction based proofs. To guarantee correct security bounds of the transformations we had to give proofs for the security definitions of the transformations. Therefore, the new transformations are guaranteed to be correct. As a future work we need to implement additional transformations that would allow to prove the security of complex primitives.

Tööriist sümmeetriliste primitiivide formaalseks analüüsiks

Magistritöö (30 EAP)

Kristjan Krips

Resümee

Tänapäeva maailmas on krüpograafia laialt kasutusel, et turvata elektroonilist sidet. Seega on oluline, et vastavad krüptograafilised algoritmid oleksid tõestatavalt turvalised. Krüptograafiliste protokollide ehitamiseks kasutatakse krüptograafilisi primitiive ja terve protokolli turvalisus tõestatakse vastavate primitiivide turvalisusle tuginedes. Sümmeetrilised primitiivid kasutavad ühte salajast võtit nii krüpteerimiseks kui ka dekrüpteermiseks ja on kiiremad kui asümmeetrilised primitiivid. Seetõttu kasutatakse sümmeetrilisi primitiive paljudes protokollides ja seega on vaja leida vastavatele primitiividele turvatõestused.

Üks võimalus turvatõestuste kirjutamiseks on kasutada krüptograafiliste mängude põhist tõestamist. Krüptograafilised mängud modelleerivad primitiive kindlas keskkonnas, kus leidub vastane. Mängude põhise tõestamise abil kirjutatakse esialgne mäng ümber nii, et primitiivi turvalisust oleks lihtsam tõestada. Selline tõestamise meetod on keeruline, kuna see nõuab palju ümberkirjutamist. Lisaks sellele võib mängude ümberkirjutamisel tekkida tõestusse vigu. Antud olukorra lahendamiseks kasutatakse tööriistu, mis abistavad tõestuse läbiviimist.

Antud magistritöö teemaks on tööriista arendamine, mis aitab mängude põhiseid tõestusi läbi viia sümmeetriliste primitiivide jaoks. Vastava tööriista nimi on ProveIt ja see lubab modifitseerida krüptograafilisi mänge, kasutades kindlaid tões-

tuse skeeme. Vastava tõestusskeemi rakendamist nimetatakse transformatsiooniks. Minu eesmärgiks oli lisada antud programmis kasutatavale keelele semantika ja kasutada seda semantikat erinevate transformatsioonide lisamiseks programmi ProveIt. Magistritöö käigus implementeerisin ProveIt jaoks järgnevad transformatisoonid: Dead Code Elimination, Statement Switching, Remove Condition, Replace Function Call, Wrap ja Random Function Simulation. Surnud koodi eemaldamiseks oli vaja implementeerida elusate muutujate analüüs. Selleks, et implementeerida transformatsiooni tuleb kõigepealt tõestada vastav turvalisuse definitsioon. Seega on mu magistriöös vastavate tõestusskeemide turvatõestused. Edasise tööna tuleb ProveIt lisada keerukamaid transformatisoone, et võimaldada keerukamate turvatõestuste läbiviimist.

Appendix A Source Code

The source code of the implemented game transformations is available in the following SVN repository: `svn://ats.cs.ut.ee/u/proveit`.
To get the instructions for running the code, contact the project leader of ProvIt: Liina Kamm, `kamm@ut.ee` .

Bibliography

[Kam11] Liina Kamm. *ProveIt - Cryptographic Protocol Proving Assistant Protocol Language*. Cybernetica AS, University of Tartu, October 2011.

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

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.