# Control and data dependency-based automated analysis of security protocols for confidentiality

Ilja Tšahhirov
(joint work with Peeter Laud)

# Problem statement

- Given the program (performing computations and exchanging messages over public channel),
- Working with some secret data
- No active adversary should be able to learn anything about the secret data

- Automatically determine whether the protocol is secure or not.

# Existing Solutions

- Program with operational semantics.
- Adversary running the program and observing its outputs.
- Secrecy definition – adversary cannot distinguish between two identical programs working with different secret data.

- Properties of cryptographic primitives enable modification of protocol text
- It is possible to achieve the point when “secret” value is not used in the protocol anymore.
- Bruno Blanchet technique – uses this approach.
- Syntax trees or flow graphs are not the best program representation for transformations.

# Our Technique Outline

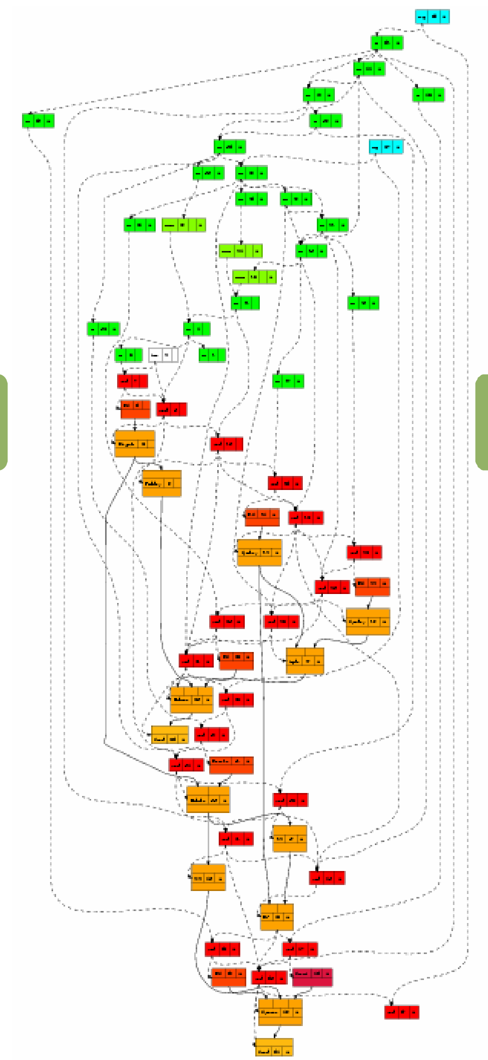
- **Program in WHILE-style language**.
- Operational semantics.
- Adversary running the program and observing its outputs.
- Secrecy definition – adversary cannot distinguish between two identical programs working with different secret data.

- **Program dependency graph**
- Graph semantics – functional dependency of outputs on the inputs.
- Adversary supplying the inputs and observing the outputs.
- Secrecy definition – no functional dependency of the outputs on the secret data.

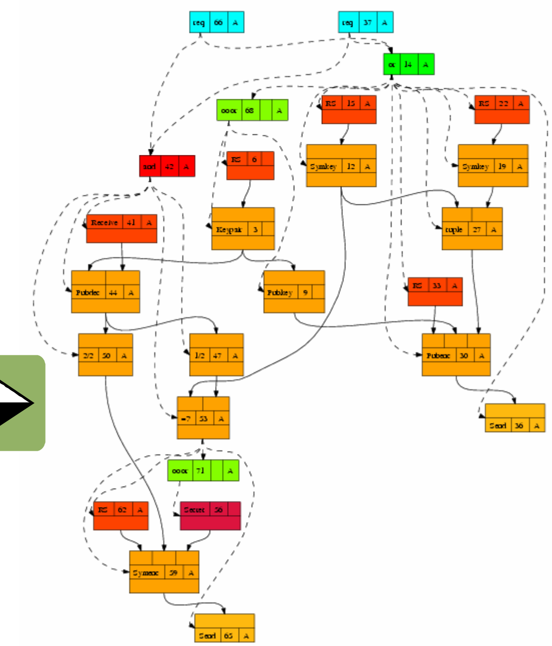
# Technique Outline 2

```
kp:=gen_key_pair
pk:=get_pub_key(kp)

Replicate
(
 sk_1:=gen_s_key
 sk_2:=gen_s_key
 sks:=(sk_1,sk_2)
 sks_e:=aenc(pk,sks)
 send(sks_e)
 rsks_e:=receive
 rsks:=adecd(kp,rsks_e)
 rsk_1:=proj_1^2(rsks)
 rsk_2:=proj_2^2(rsks)
 check(rsk_1=sk_1)
 M_e:=senc(sk_2,M)
 send(M_e)
)
```



…



# Program language semantics

- Structural operational semantics
- Program execution is set of transitions on the configuration set
- Configuration captures computation state at a given moment

The configuration is a tuple $< S, s, out >$, where

- $S \in Stm \cup \{\epsilon\}$: unexecuted yet statements,
- $s \in State$: current state $(Var \rightarrow Val)$,
- $out \in Val \cup \{ok\} \cup \{\perp\}$: output to the adversary

# Program language semantics 2

$$\frac{\mathbb{A}[\![a]\!]s \neq \bot}{< PAsgn(x,a), s, \_ > \Rightarrow < \epsilon, s[x \mapsto \mathbb{A}[\![a]\!]s], ok >}(\text{Asgn}^{\text{ok}})$$

$$\frac{\mathbb{A}[\![a]\!]s = \bot}{< PAsgn(x,a), s, \_ > \Rightarrow < PStopped, s, stuck >}(\text{Asgn}^{\text{e}})$$

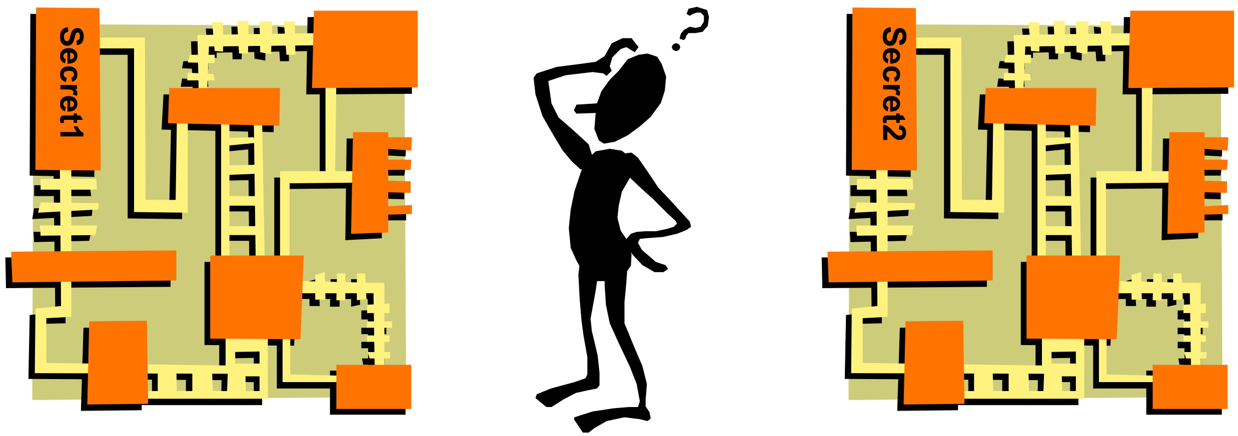
$$\frac{< S_1, s, \_ > \Rightarrow < S_1', s', out >}{< PParal(S_1, S_2), s, \_ > \Rightarrow < PParal(S_1', S_2), s', out >}(\text{Paral}^1)$$

$$\frac{< S_1, s, \_ > \Rightarrow < \epsilon, s', out >}{< PParal(S_1, S_2), s, \_ > \Rightarrow < S_2, s', out >}(\text{Paral}^2)$$

$$\frac{< S_2, s, \_ > \Rightarrow < S_2', s', out >}{< PParal(S_1, S_2), s, \_ > \Rightarrow < PParal(S_1, S_2'), s', out >}(\text{Paral}^3)$$

$$\frac{< S_2, s, \_ > \Rightarrow < \epsilon, s', out >}{< PParal(S_1, S_2), s, \_ > \Rightarrow < S_1, s', out >}(\text{Paral}^4)$$

# Security definition

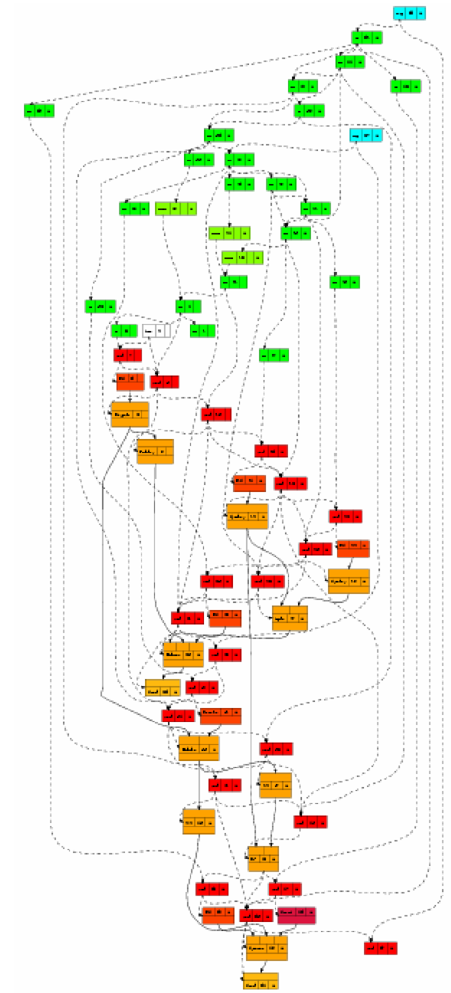



The protocol is considered secure if there's no non-negligible correlation between the final state of the adversary and the secret message

# Translation to graphs

```
kp:=gen_key_pair
pk:=get_pub_key(kp)

Replicate
(
    sk_1:=gen_s_key
    sk_2:=gen_s_key
    sks:=(sk_1,sk_2)
    sks_e:=aenc(pk,sks)
    send(sks_e)
    rsks_e:=receive
    rsks:=adecd(kp,rsks_e)
    rsk_1:=proj_1^2(rsks)
    rsk_2:=proj_2^2(rsks)
    check(rsk_1=sk_1)
    M_e:=senc(sk_2,M)
    send(M_e)
)
```



Adversary "playing" with graph should observe at least same set of info as running the program.

# Graph execution rules

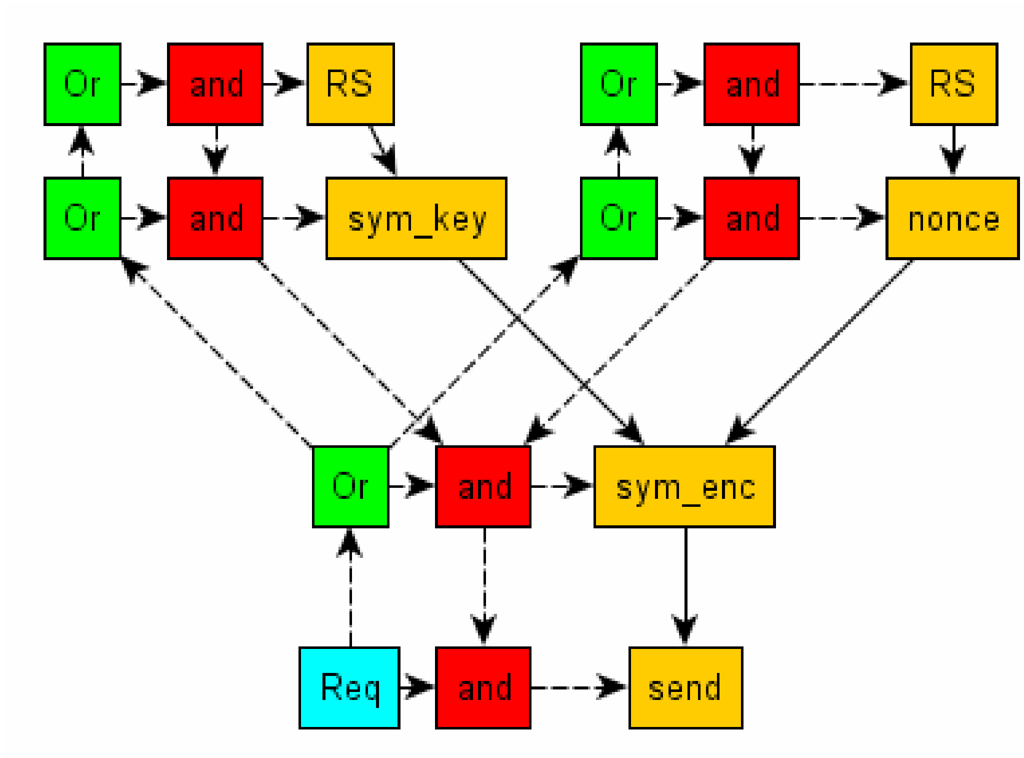
- Graph represents:
  - Operations occurring in the protocol (nodes)
  - Data dependencies (edges)
  - Control dependencies (edges)
  - Adversary interface (special type of nodes - Req)
  - Note: If replication is present, the Graph is infinite (each replicated operation is present in $\mathbb{N}$ copies).

# Graph execution rules 2

- Execution rules
  - Adversary defines which outputs he'd like to see (sets true/false values to Req nodes), and which values are supplied to the inputs (values program gets from communication channels)
  - Now find the values of the outputs (using the graph semantics).

# Graph execution rules - example

```
x:= gen_sym_key
y:= random
z:= sym_enc (y,x)
send (z)
```



# Graph semantics

- Edges (dependencies)
- Nodes (operations)
- Set of equations of type:

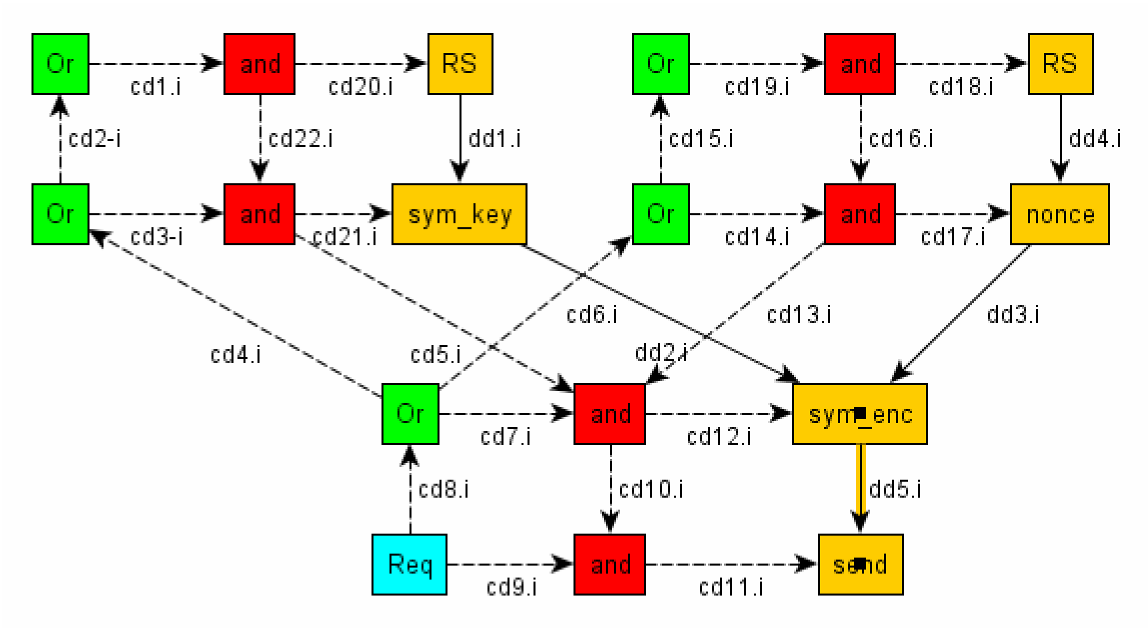
  $Dep_i$ = Operation ($Dep_1$, … , $Dep_n$)

  - One equation per node
  - Specifies how the dependencies are related – each dependency is a function of other dependencies

- Smallest solution of the set of equations – final values, subset of them visible to Adversary.
  - All Operations are monotone – so the smallest solution should exist.

- Security criteria – no functional dependency of outputs on secret.

# Equation examples

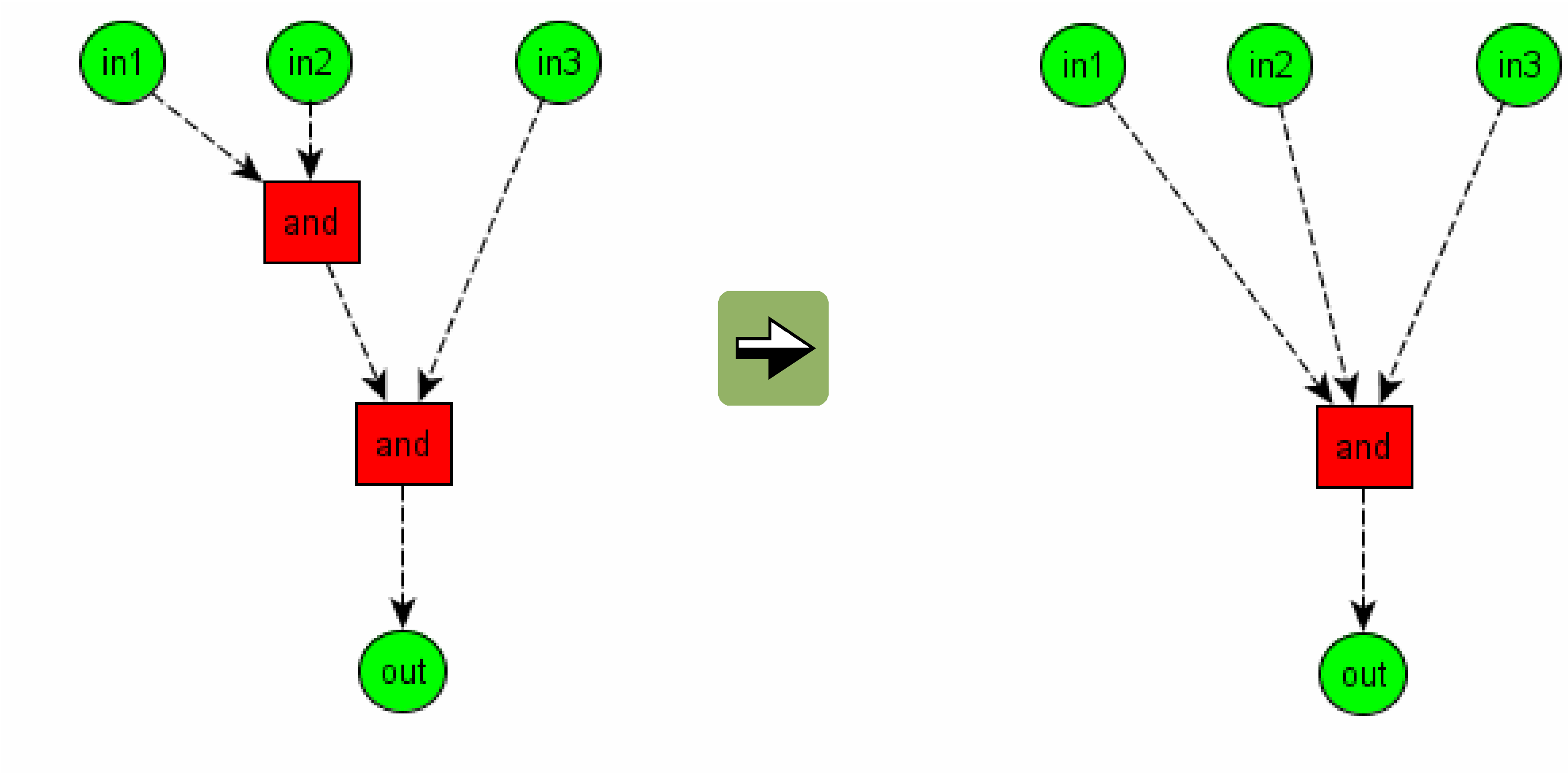



- Equation examples:
  - dd3.i = Nonce( cd17.i, dd4.i)
  - cd17.i = And( cd14.i, cd16.i)
  - cd8.i = Req ← This value is supplied by Adversary.

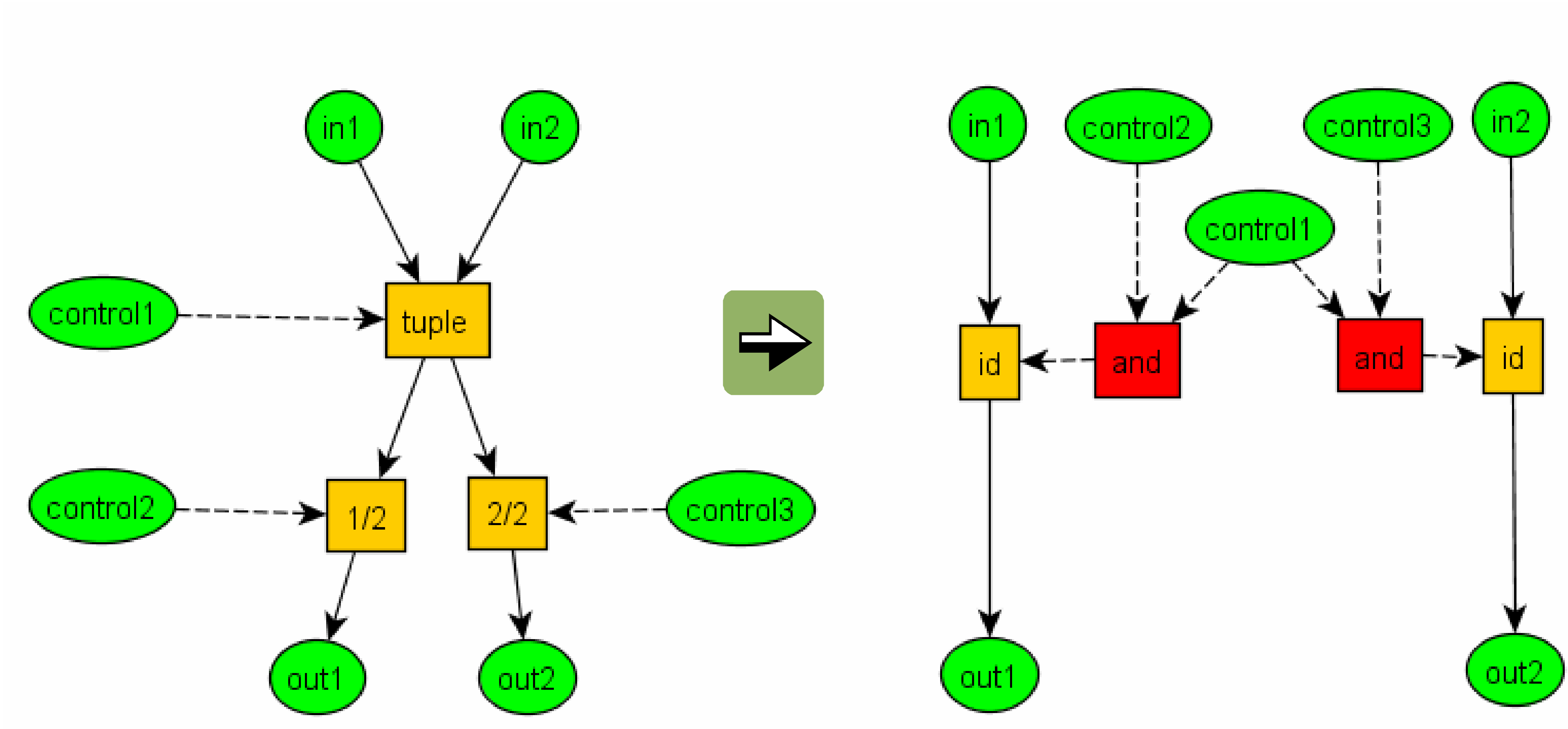
# Graph transfromation

- Transforming the graph / equations
- The smallest solution should stay indistinguishable from the original one.
- Transformations are based on the properties of the operation (including cryptographic primitives)

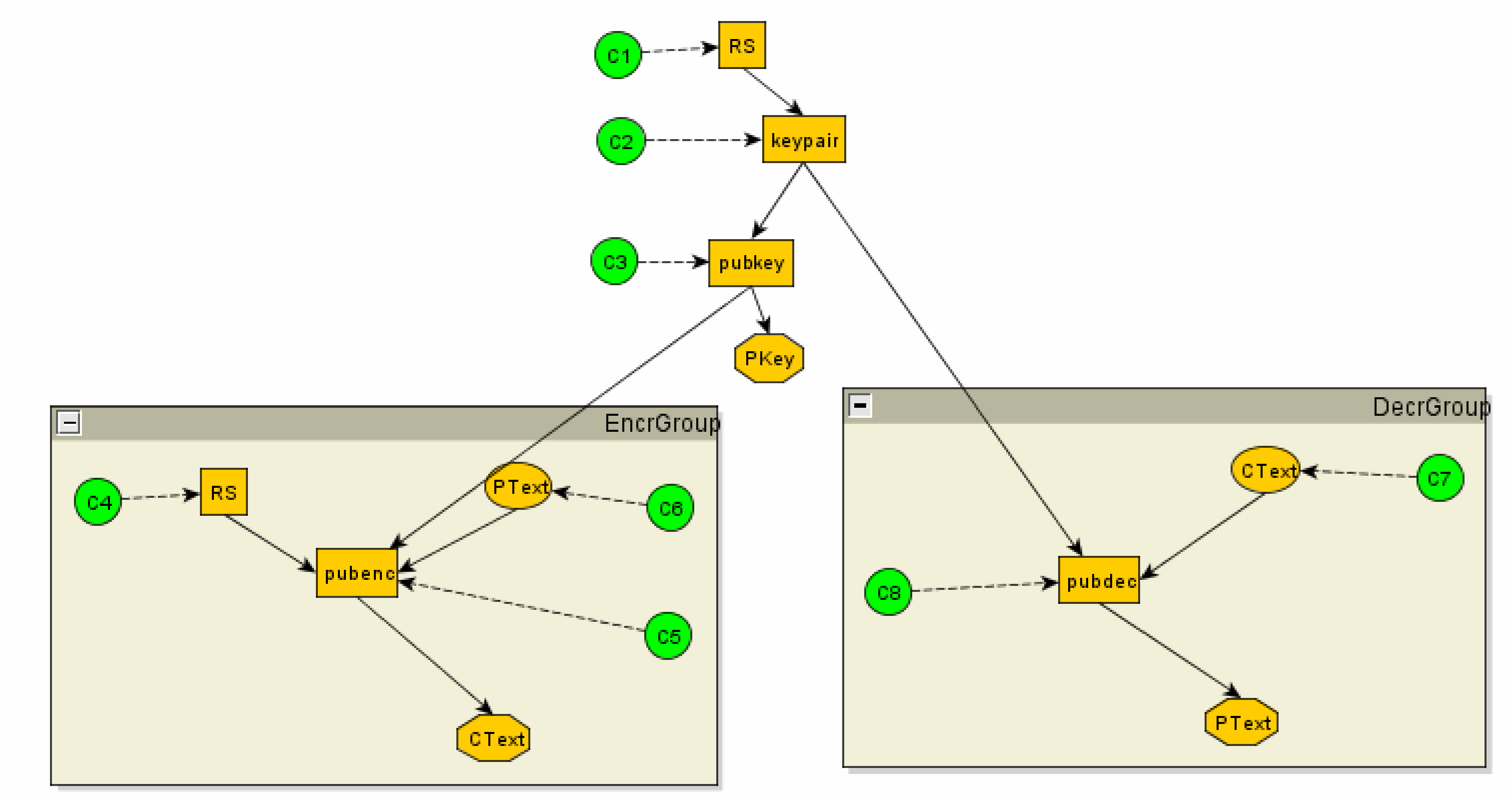
# Control flow transformation example



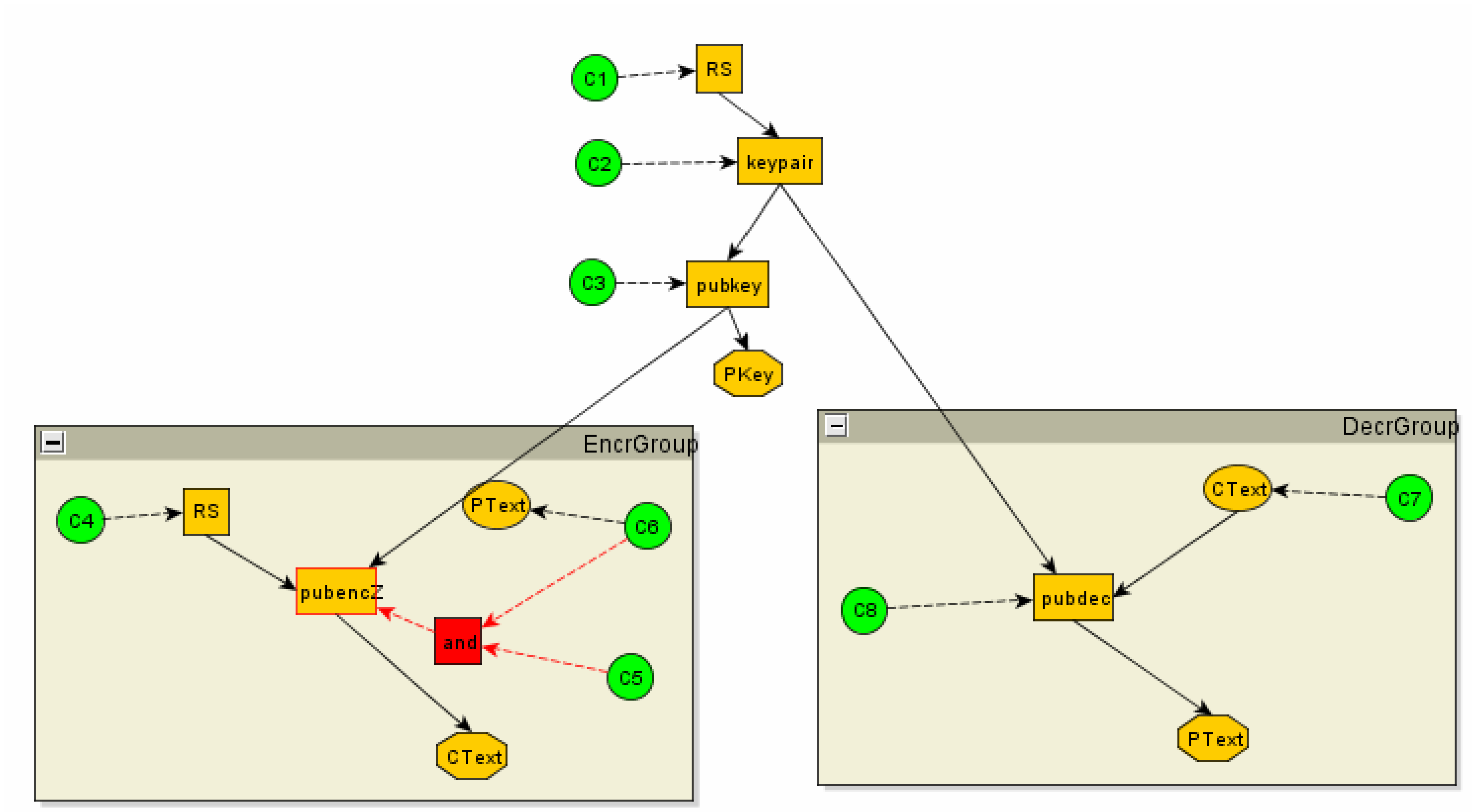
# Operation transformation example



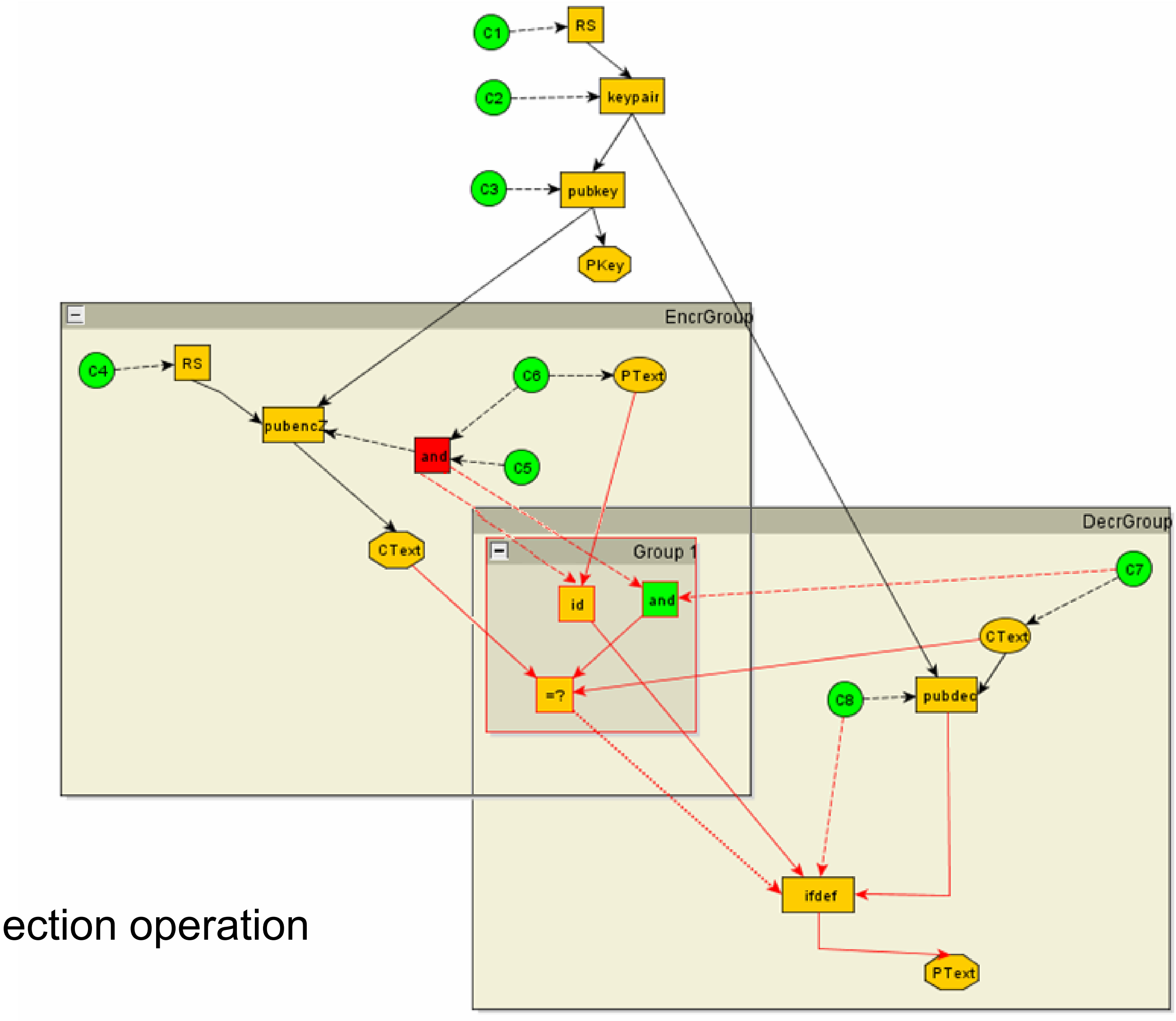
# Pubenc transformation example - 1



# Pubenc transformation example - 2

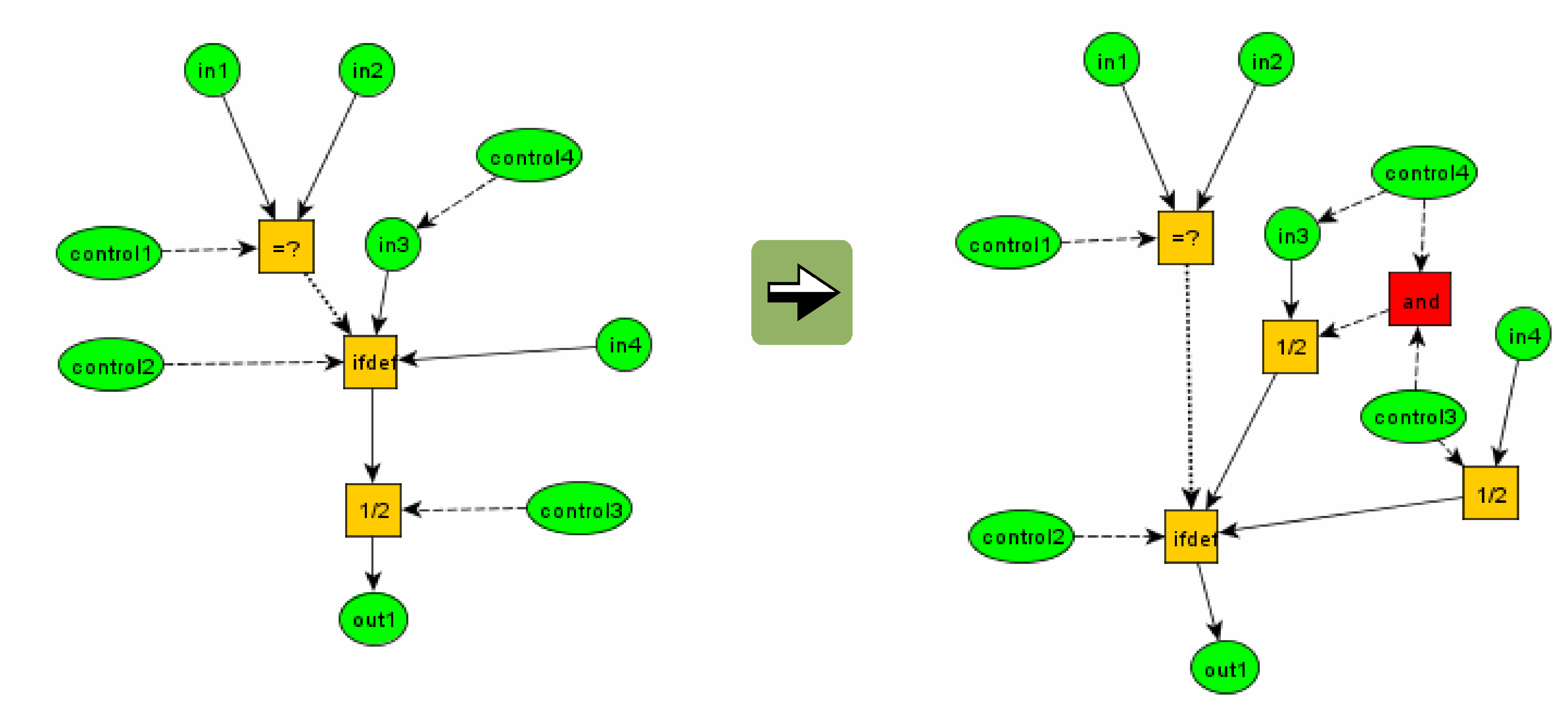


# Pubenc transformation example - 3

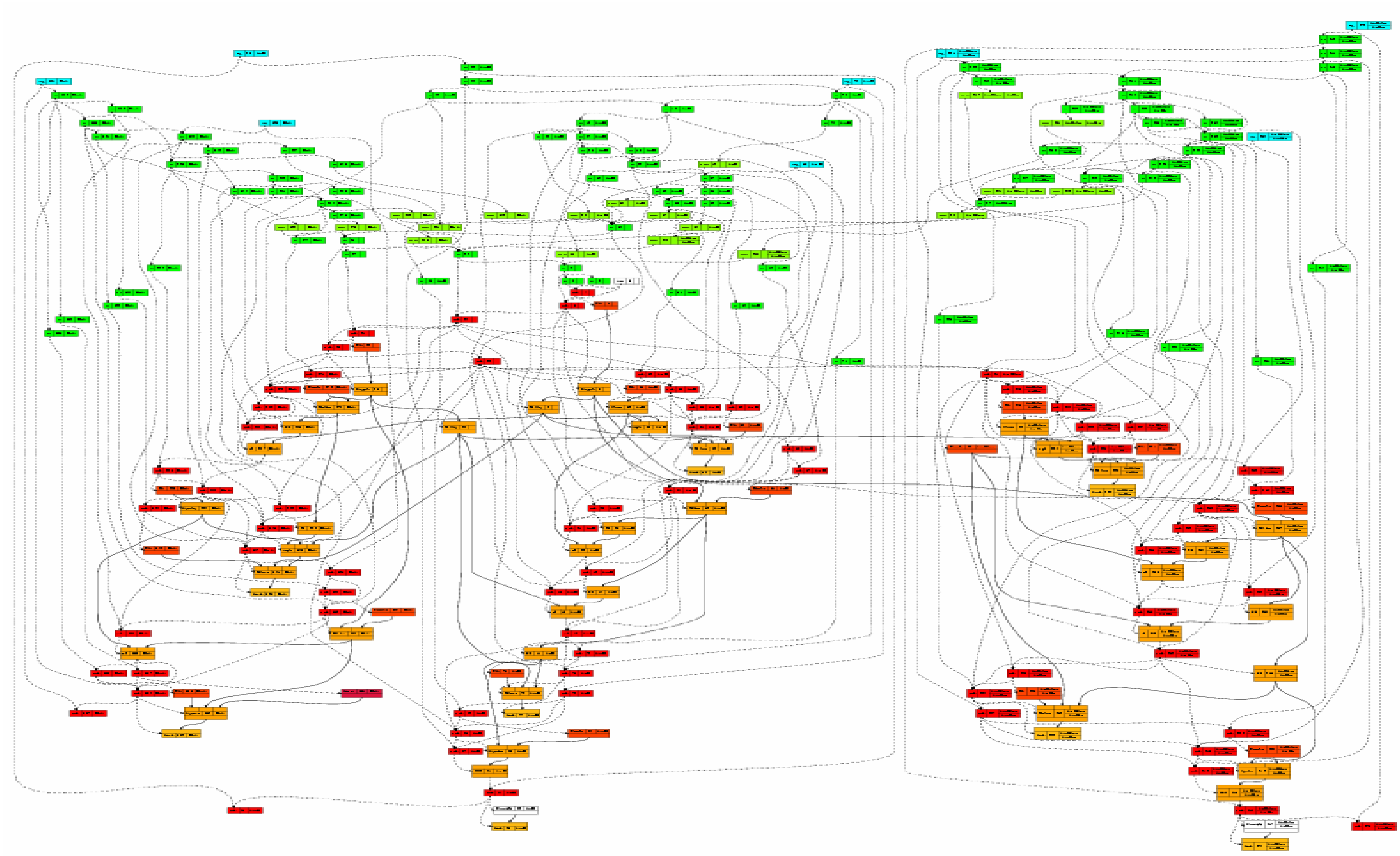



Ifdef – selection operation

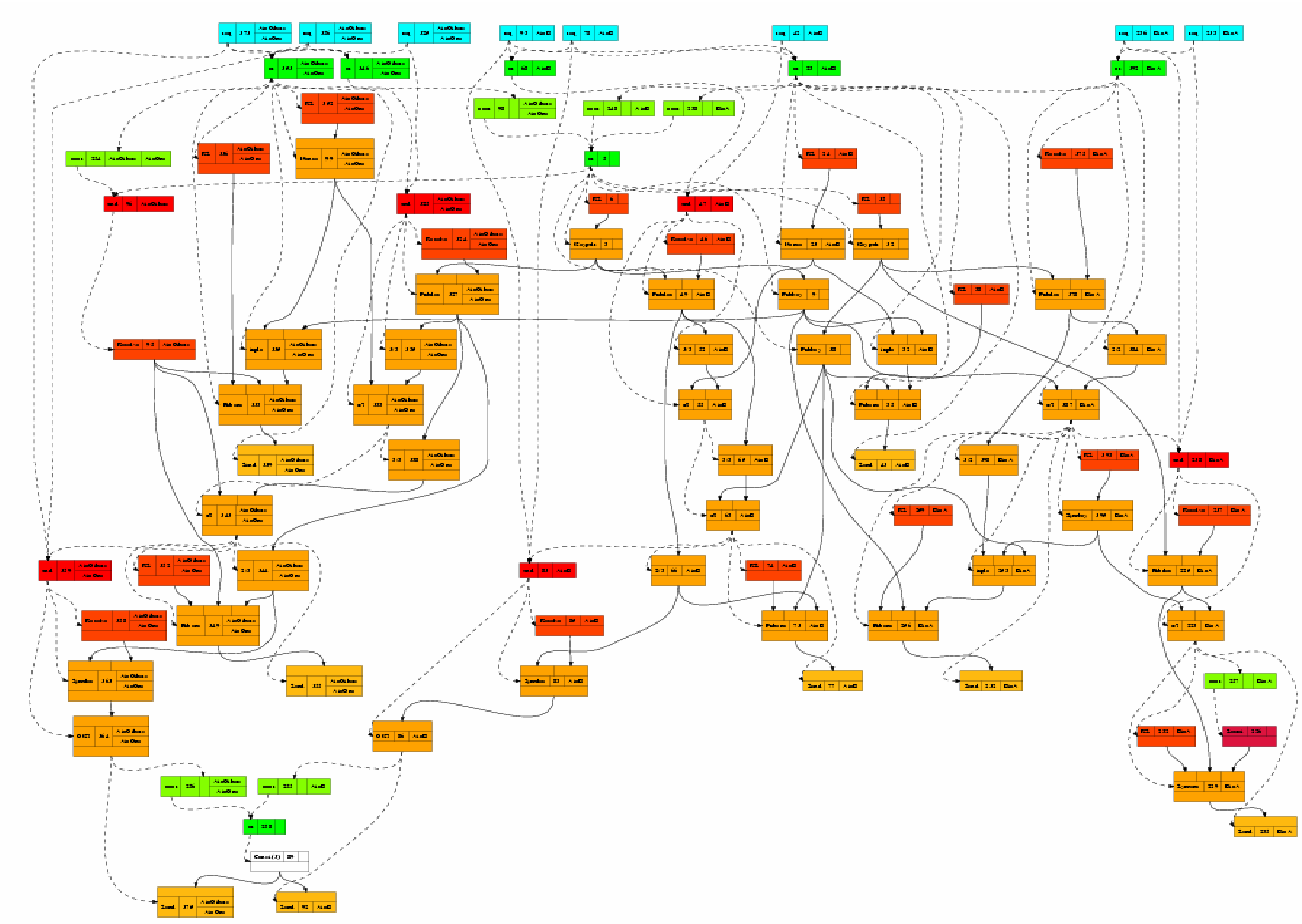
# Operation transformation example



# “Real” graph – a bit more complex ☺



# “Real” graph – after some transforms applied – still complex.



# Summary – what’s done, what’s left…

- Ready
  - A nice idea
  - Conceptual framework
  - Programming language semantics
  - Part of the graph semantics
  - Analyzer prototype
- To be done
  - Program -> graph translation correctness proof
  - Graph transformation correctness proof
  - Complete graph semantics
  - Fully functional analyzer

Thank you