# On Automated Verification of Probabilistic Programs

Joël Ouaknine
Oxford University Computing Laboratory, UK

(Joint work with Axel Legay, Andrzej Murawski, James Worrell)

Lorentz Workshop, November 2007

# Verification of Probabilistic Programs

- Probabilistic programs occur in a wide variety of situations:
  - Randomised algorithms, e.g. Miller-Rabin primality testing
  - Symmetry-breaking and fairness in distributed systems
  - Achieving security goals, e.g. anonymity in electronic voting
  - . . .
- Randomisation can improve complexity, even achieve goals that deterministic algorithms cannot
- Probability makes reasoning even harder than in deterministic settings

# Verification of Probabilistic Programs

Many frameworks for modelling/reasoning about probabilistic systems, e.g.:

- Markov chains
- Probabilistic process algebras
- PRISM
- . . .

# Verification of Probabilistic Programs

Many frameworks for modelling/reasoning about probabilistic systems, e.g.:

- Markov chains
- Probabilistic process algebras
- PRISM
- ...

**Our goal: verify probabilistic programs.**

# A Probabilistic Programming Language

- Iteration (`while`), conditionals (`if then else`), ...
- Arrays
- Procedures (with value-passing and reference-passing parameters)
- Global and local variables
- ...
- Randomisation

# A Probabilistic Programming Language

- Iteration (`while`), conditionals (`if then else`), . . .
- Arrays
- Procedures (with value-passing and reference-passing parameters)
- Global and local variables
- . . .
- Randomisation

Some restrictions:

- Finite datatypes
- No pointers
- No recursion

# Probabilistic Programs and Game Semantics

- We model **probabilistic programs** as **probabilistic automata** using **game semantics**
- The probabilistic automata represent **probabilistic strategies** for the programs
- Allows us to model **open programs** (modules with indeterminate components). Enables **compositional reasoning**

# Probabilistic Program Equivalence

- Two programs $P_1$ and $P_2$ are **equivalent** if no context can distinguish them: $P_1 \cong P_2$ iff

$$\forall C \,.\, \mathbf{Prob}(C[P_1] \text{ terminates}) = \mathbf{Prob}(C[P_2] \text{ terminates})$$

  - A **context** is a program with a 'hole' in it, such that $C[P_i]$ are closed programs of type `com`

# Probabilistic Program Equivalence

- Two programs $P_1$ and $P_2$ are **equivalent** if no context can distinguish them: $P_1 \cong P_2$ iff

$$\forall C \,.\, \mathbf{Prob}(C[P_1] \text{ terminates}) = \mathbf{Prob}(C[P_2] \text{ terminates})$$

  - A **context** is a program with a ‘hole’ in it, such that $C[P_i]$ are closed programs of type `com`
  - Note that ‘distinguishing’ is a probabilistic notion: contexts can do statistical sampling

# Probabilistic Program Equivalence

- Two programs $P_1$ and $P_2$ are **equivalent** if no context can distinguish them: $P_1 \cong P_2$ iff

$$\forall C \,.\, \mathbf{Prob}(C[P_1] \text{ terminates}) = \mathbf{Prob}(C[P_2] \text{ terminates})$$

  – A **context** is a program with a 'hole' in it, such that $C[P_i]$ are closed programs of type `com`

  – Note that 'distinguishing' is a probabilistic notion: contexts can do statistical sampling

- **Theorem** [MO 05]. Two programs are equivalent if the probabilistic automata representing their respective strategies accept the same probabilistic languages

## Probabilistic Program Equivalence

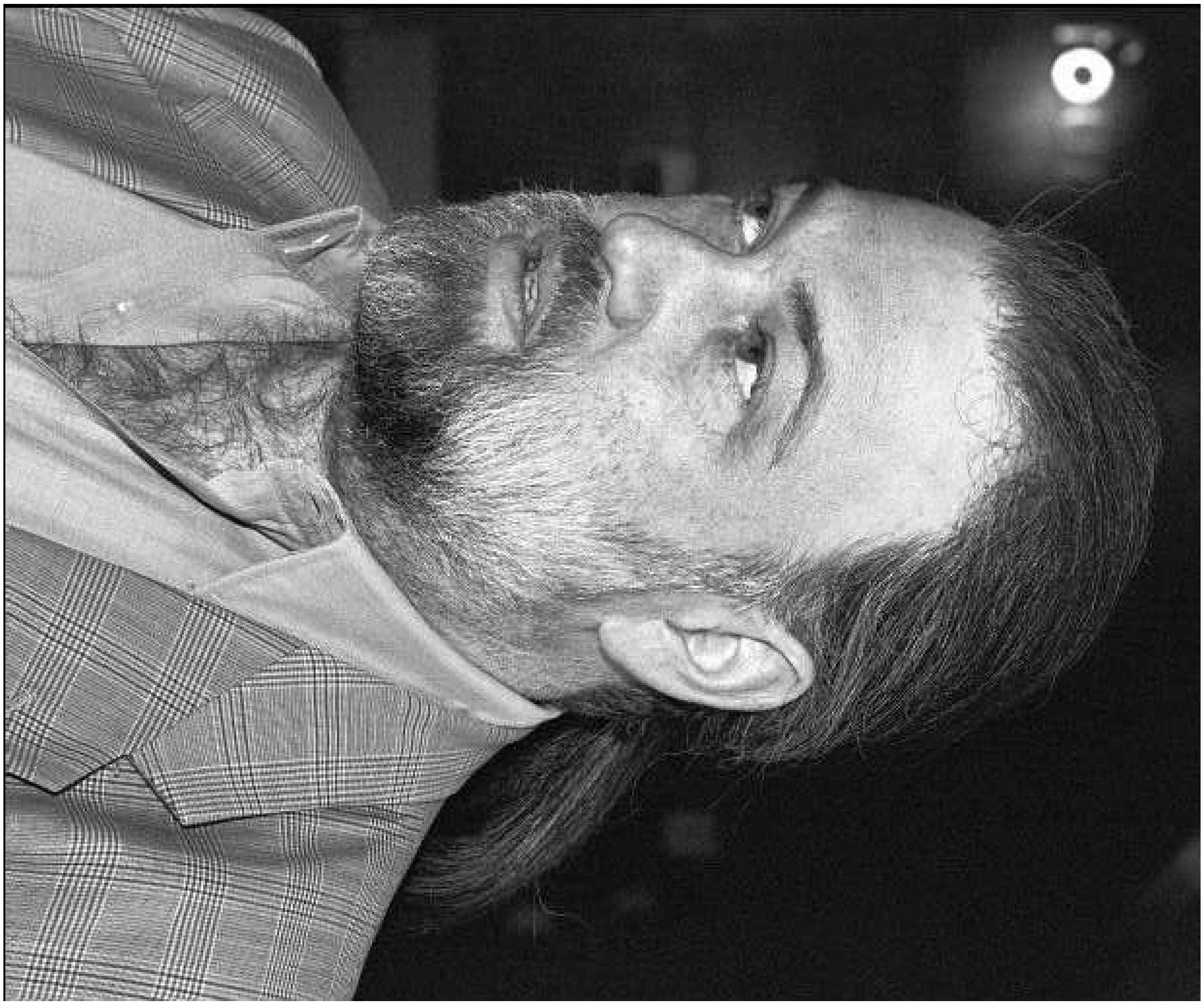
- Two programs $P_1$ and $P_2$ are **equivalent** if no context can distinguish them: $P_1 \cong P_2$ iff

$$\forall C \,.\, \mathbf{Prob}(C[P_1] \text{ terminates}) = \mathbf{Prob}(C[P_2] \text{ terminates})$$

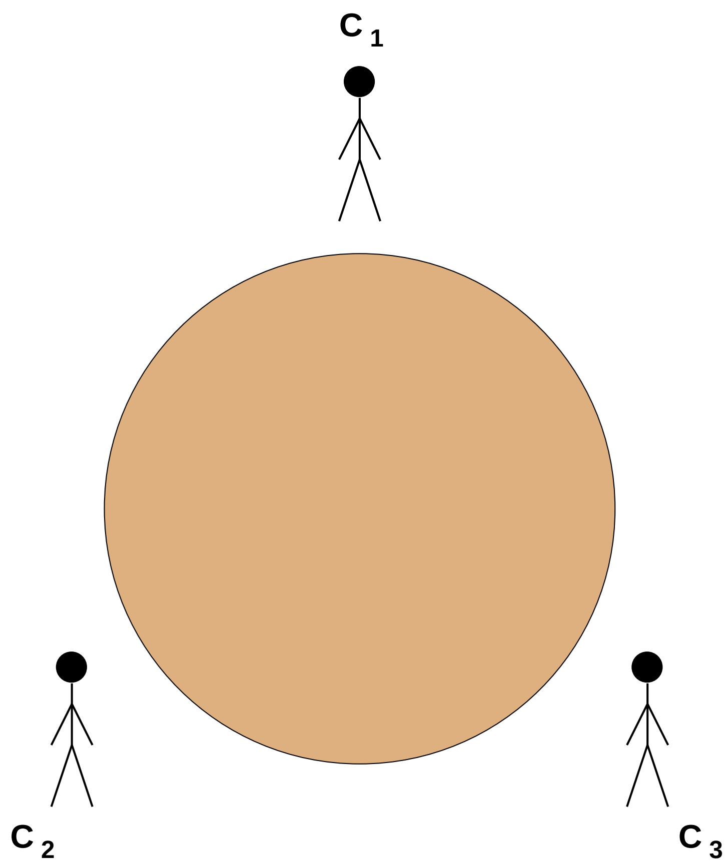
  - A **context** is a program with a ‘hole’ in it, such that $C[P_i]$ are closed programs of type `com`
  - Note that ‘distinguishing’ is a probabilistic notion: contexts can do statistical sampling

- **Theorem** [MO 05]. Two programs are equivalent if the probabilistic automata representing their respective strategies accept the same probabilistic languages

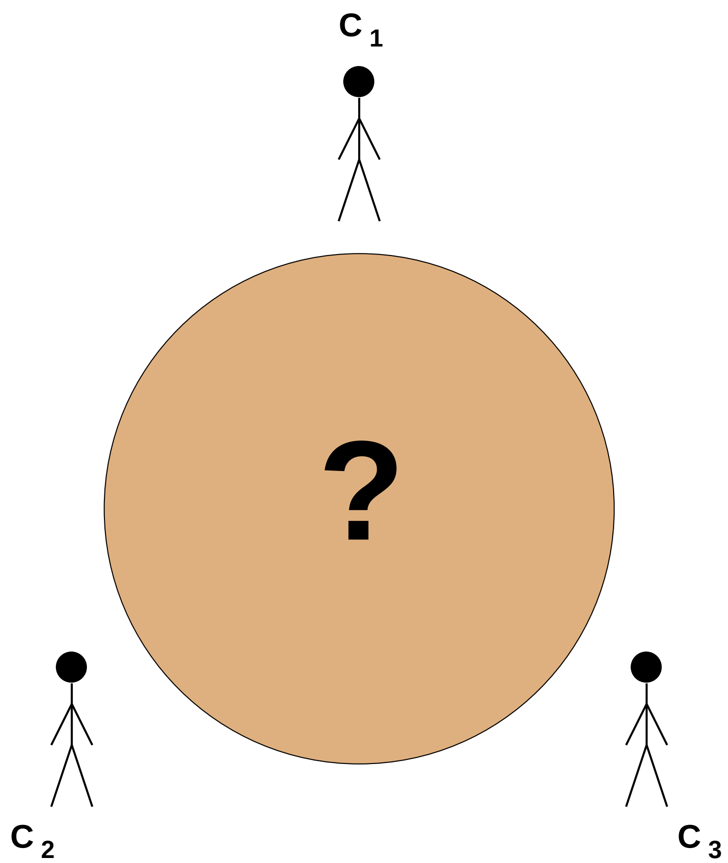
- Language equivalence for probabilistic automata is decidable in polynomial time [Tzeng 92]



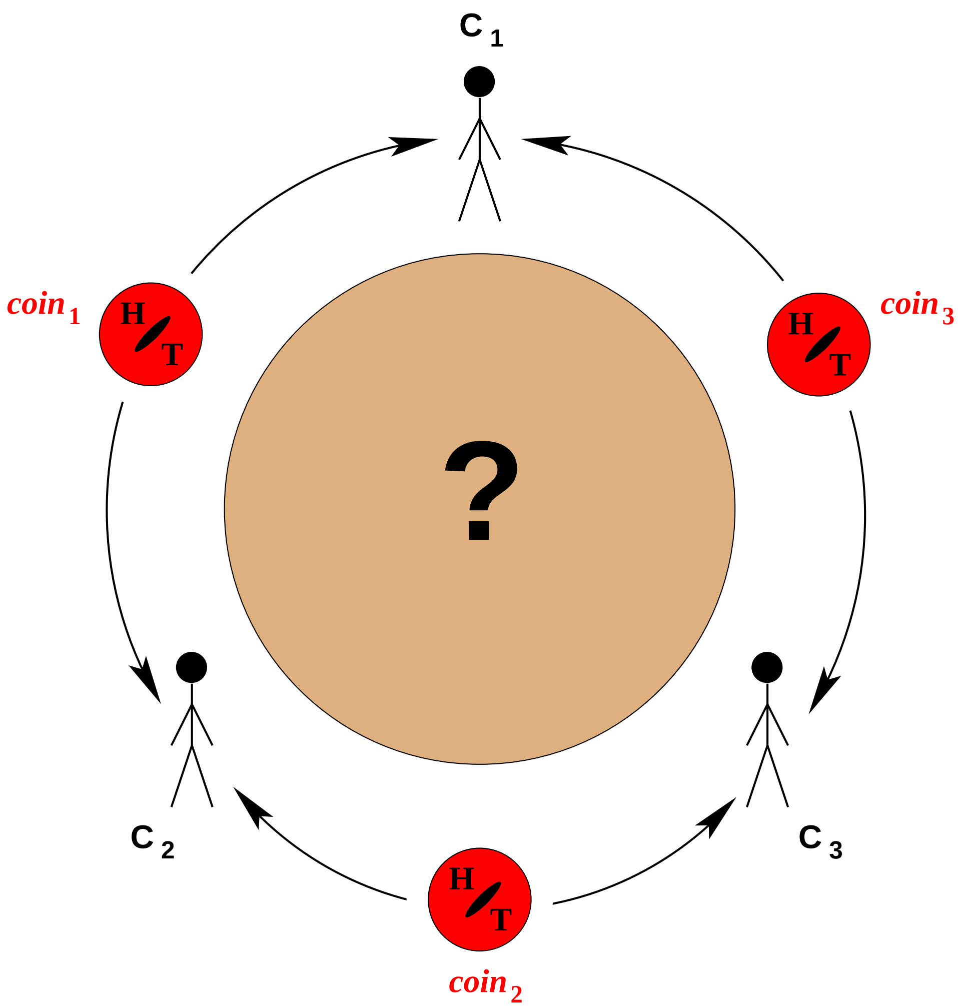
# The Dining Cryptographers



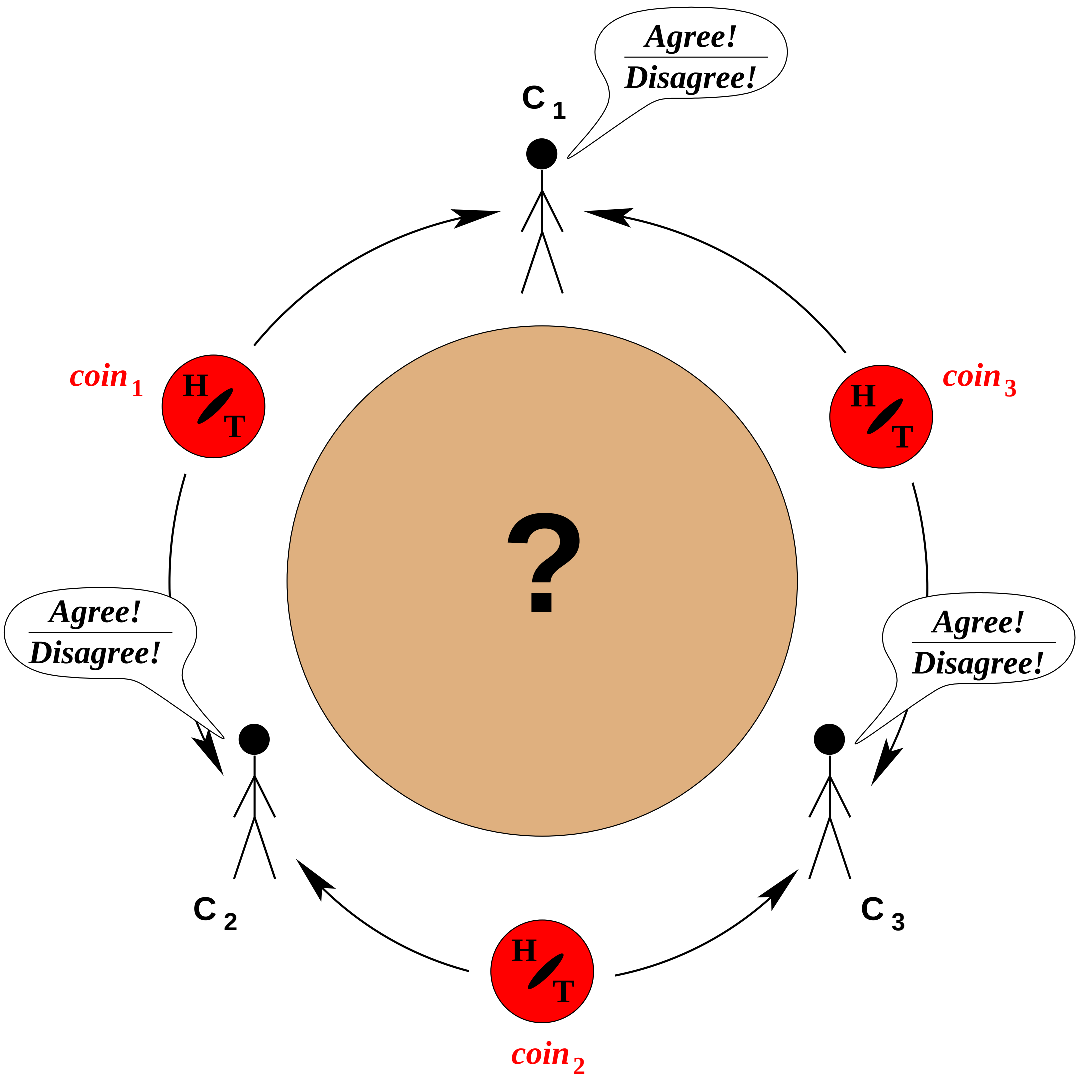
# The Dining Cryptographers



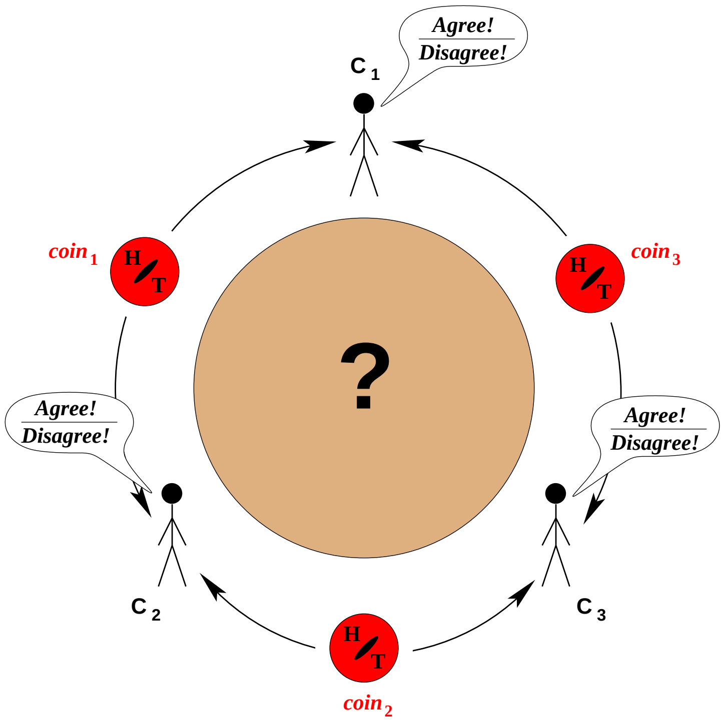
# The Dining Cryptographers



# The Dining Cryptographers

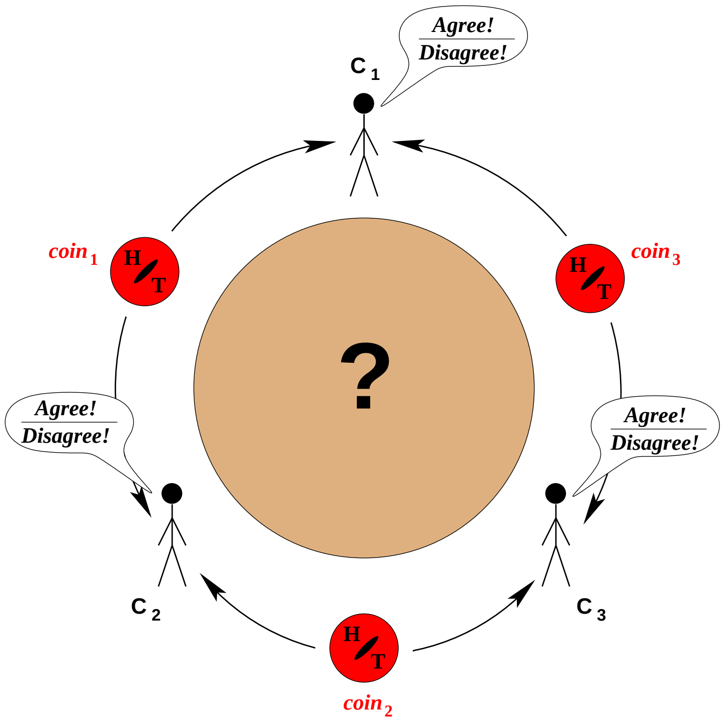


# The Dining Cryptographers




- **Correctness:** If the number of **“Disagree!”** is odd, then one of them paid, otherwise the NSA paid.

# The Dining Cryptographers

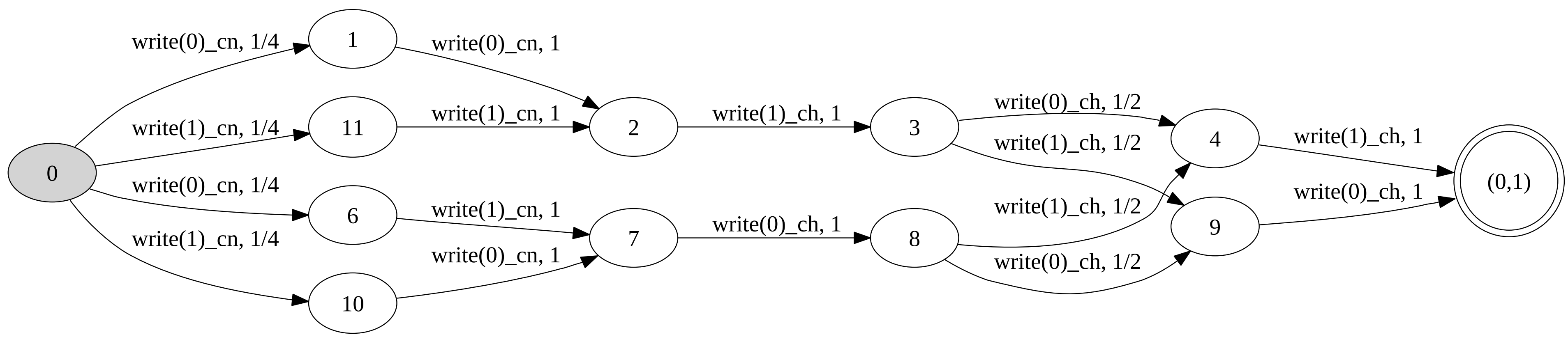



- **Correctness:** If the number of **“Disagree!”** is odd, then one of them paid, otherwise the NSA paid.
- **Anonymity:** If one of them paid, then neither of the other two cryptographers can deduce who it is.

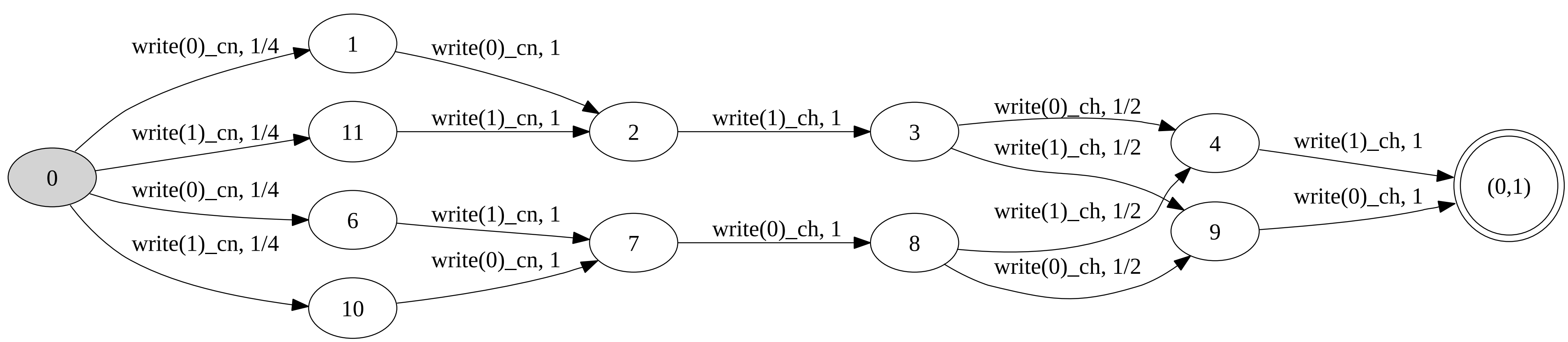
# Anonymity

```
void main(var%2 ch, var%2 cn) {

  int%4 whopaid; int%2 first; int%2 right; int%2 left; int%4 i;

  whopaid:=3;
  first:=coin;
  right:=first;
  i:=1;

  while (i) do
  {
    left := if (i=3) then first else coin;
    if (i=1) then { cn:=right; cn:=left };
    if ((left=right)+(i=whopaid)) then ch:=1 else ch:=0;
    right:=left;
    i:=i+1
  }
}
```

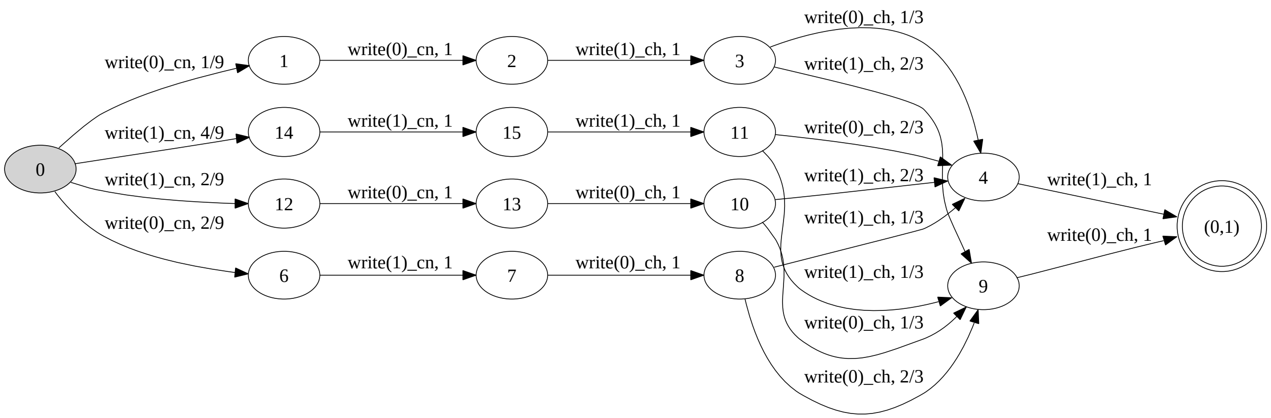
## Crypto no. 2 paid:



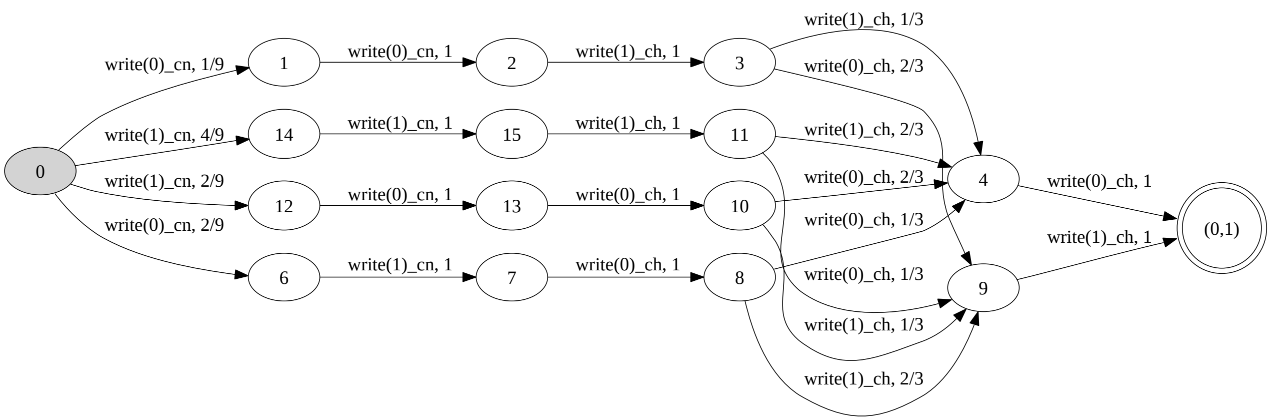

## Crypto no. 3 paid:



## Biased coins ($\frac{1}{3}/\frac{2}{3}$), Crypto no. 2 paid:



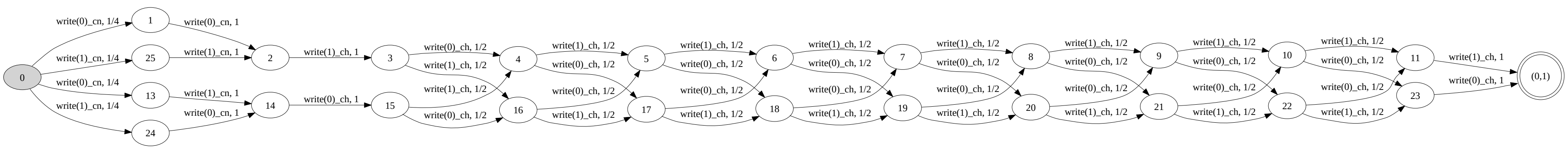

## Biased coins ($\frac{1}{3}/\frac{2}{3}$), Crypto no. 3 paid:




$$\mathbf{Prob}(0,0,1,0,1) = \frac{1}{27} \text{ vs. } \frac{2}{27}$$

# Many Dining Cryptographers

It is straightforward to model more cryptographers, e.g.:



| # crypt. | PRISM | APEX |
|---|---|---|
| 3 | 4 | 7 |
| 4 | 4 | 8 |
| 5 | 7 | 8 |
| 6 | 39 | 9 |
| 7 | 95 | 9 |
| 8 | 282 | 10 |
| 9 | 964 | 10 |
| 10 | $>$ 1h | 11 |
| 15 | OOM | 13 |
| 50 | OOM | 56 |
| 100 | OOM | 125 |

# Summary and Future Work

- Verification of probabilistic **programs**
- First fully automated verification of **anonymity** in Dining Cryptographers protocol

Future work:

- Symbolic state-space representation, predicate abstraction, . . .
- Support for pointers
- Analysing probabilistic strategies
- Automatic counterexample generation
- Case studies: anonymity for electronic voting protocols, . . .