VMC: a Dafny Library for Verified Monte Carlo Algorithms

Fabian Zaiser*   Stefan Zetzsche   Jean-Baptiste Tristan

UNIVERSITY OF OXFORD

aws

*work completed during an internship at AWS

Probabilistic Sampling

… means *generating samples from a desired distribution*

… is important:

- Cryptography
- Differential Privacy

… is hard to do correctly, e.g.

- Fisher Yates shuffle (random permutation)
- Attacks on Differential Privacy

Potential sources of bias [edit]

Care must be taken when implementing the Fisher–Yates shuffle, both in the implementation of the algorithm itself and in the generation of the random numbers it is built on, otherwise the results may show detectable bias. A number of common sources of bias have been listed below.

Implementation errors [edit]

A common error when implementing the Fisher–Yates shuffle is to pick the random numbers from the wrong range. The flawed algorithm may appear to work correctly, but it will not produce each possible permutation with equal probability, and it may not produce certain permutations at all. For example, a

On significance of the least significant bits for differential privacy

Author: Ilya Mironov Authors Info & Claims

CCS '12: Proceedings of the 2012 ACM conference on Computer and communications security • October 2012 • Pages 650–661 • https://doi.org/10.1145/2382196.2382264

Published: 16 October 2012 Publication History

ABSTRACT

We describe a new type of vulnerability present in many implementations of differentially private mechanisms. In particular, all four publicly available general purpose systems for differentially private computations are susceptible to our attack.

The vulnerability is based on irregularities of floating-point implementations of the privacy-preserving Laplacian mechanism. Unlike its mathematical abstraction, the textbook sampling procedure results in a porous distribution over double-precision numbers that allows one to breach differential privacy with just a few queries into the mechanism.

We propose a mitigating strategy and prove that it satisfies differential privacy under some mild assumptions on available implementation of floating-point arithmetic.

Reasoning about Probabilistic Samplers is Hard

```
X ~ Geometric(3/4)
Y ~ Geometric(3/4)
Z ~ Bernoulli(5/9)
T := X + Y + Z
repeat 3 times:
    F ~ Binomial(2 * T, 1/2)
    if F != T:
        return 0
return 1
```

What is the probability of returning 1?

1/π

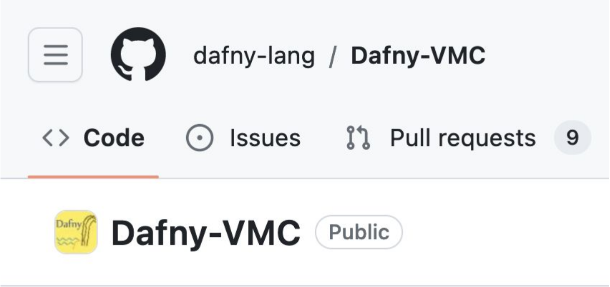
How???

$$\frac{1}{\pi} = \sum_{n=0}^{\infty} \binom{2n}{n}^3 \frac{6n+1}{4^{4n+1}}$$

Example from: Flajolet, Pelletier, Soria: *On Buffon Machines and Numbers*

Dafny-VMC ("Verified Monte-Carlo")

- **Samplers** of various distributions
- Proofs of **correctness**
- Implemented and verified in **Dafny**
- **Interoperability** with Java
- Work in progress (partially axiomatized)
- **Open-source** on Github: https://github.com/dafny-lang/Dafny-VMC/



Structure of Probabilistic Samplers in Dafny-VMC

1. **Functional model**
2. **Correctness proof** of the model

} Focus of this talk

3. **Imperative implementation** (using external randomness source)
4. **Proof of correspondence** between model and implementation
5. **Statistical tests**

Randomized Functions in Dafny

- Dafny's functions are **deterministic**
- → need to get infinitely many **random bits as input**
- Compute **random value** and return **unused bits**
- *"Bitstream transformers"*

```
type Bits = nat -> bool

function CoinModel(s: Bits): (bool, Bits) {
  (s(0), (n: nat) => s(n + 1))
}
```

Compositionality

- Passing around bitstreams is error-prone
- Joe Hurd introduced a monad **abstraction**
- Small set of **combinators**: `Coin`, `Return`, `Bind`, `While`
- Can be used to model all our samplers

```
type Hurd<A> = Bits -> (A, Bits)

function Coin(): Hurd<bool> {
  (s: Bits) => (s(0), (n: nat) => s(n + 1))
}

function Return<A>(a: A): Hurd<A> {
  (s: Bits) => (a, s)
}

function Bind<A, B>(
  h: Hurd<A>, f: A -> Hurd<B>
): Hurd<B>

function While<A>(
  cond: A -> bool,
  body: A -> Hurd<A>
): A -> Hurd<A>
```

Probability in Dafny

- **Probability measure** on bitstreams ("independent & uniformly distributed bits")
- Hurd proved that bitstreams are a **probability space** (currently axiomatized)

```
ghost const prob: iset<Bits> -> real

ghost function probMass<A>(
  h: Hurd<A>, result: A
): real {
  prob(iset s | h(s).0 == result)
}

lemma CoinIsCorrect()
  ensures probMass(Coin(), false) == 0.5
  ensures probMass(Coin(), true) == 0.5
```

Bernoulli(exp(−γ)) Distribution

- Returns `true` with **probability exp(−γ)** for γ in [0, 1]
- "Source" of **irrational** probabilities
- **Building block** for other samplers

$$\mathbb{P}[k > n] = \frac{\gamma}{1} \cdot \frac{\gamma}{2} \cdots \frac{\gamma}{n} = \frac{\gamma^n}{n!}$$

$$\mathbb{P}[k = n] = \frac{\gamma^{n-1}}{(n-1)!} - \frac{\gamma^n}{n!}$$

$$\mathbb{P}[k \text{ odd}] = \Sigma_{n=0}^{\infty} \left(\frac{\gamma^{2n}}{(2n)!} - \frac{\gamma^{2n+1}}{(2n+1)!}\right) = \Sigma_{n=0}^{\infty} \frac{(-\gamma)^n}{n!} = e^{-\gamma}$$

```
method BernExp(gamma: real): bool
  # for gamma in [0,1]
  k := 0
  a := true
  while a:
    k += 1
    a := Bernoulli(gamma / k)

  return k % 2 == 1
```

Bernoulli(exp(−γ)) Distribution

```
function BernExp(gamma: real): Hurd<bool>
  requires 0.0 <= gamma <= 1.0
{
  Bind(
    While(
      (ak: (bool, nat)) => ak.0,
      (ak: (bool, nat)) =>
        var k' := ak.1 + 1;
        Bind(
          Bernoulli(gamma / k' as real),
          a' => Return((a', k'))
        )
    )((true, 0)),
    (ak: (bool, nat)) => Return(ak.1 % 2 == 1)
  )
}
```

```
method BernExp(gamma): bool
  # for gamma in [0,1]
  k := 0
  a := true
  while a:
    k += 1
    a := Bernoulli(gamma / k)

  return k % 2 == 1
```

Probabilistic Loops

- Some samplers require loops (e.g. **rejection sampling**)
- **Cannot** sample from Uniform{0,1,2} with **bounded number of bits**
- Loops in samplers **terminate almost surely**

```
function While<A>(
  cond: A -> bool,
  body: A -> Hurd<A>
): A -> Hurd<A> {
  (state: A) =>
  if cond(state)
  then Bind(
    body(state),
    While(cond, body))
  else Return(state)
}
```

Error: cannot prove termination

Tracking Nontermination

- We need to track **nontermination** explicitly
- Change our **probability monad**!

→ can talk about the **probability of nontermination**!

```
type Hurd<A> = Bits -> (A, Bits) // old
type Prob<A> = Bits -> Result<A> // new

datatype Result<A> =
| Diverging
| Result(value: A, rest: Bits)

function Coin(): Prob<bool>

function Return<A>(a: A): Prob<A>

function Bind<A, B>(
 p: Prob<A>,
 f: A -> Prob<B>
): Prob<B>
```

Probabilistic While Loops – Take 2

```
function WhileBounded<A>(
  fuel: nat, cond: A -> bool, body: A -> Prob<A>, init: A
): Prob<A> {
  if fuel == 0 then s => Diverging
  else if !cond(init) then Return(init)
  else Bind(
    body(init),
    state' => WhileBounded(fuel - 1, cond, body, state'))
}
```

While loop with bounded fuel

Out of fuel

Normal recursion

```
ghost function While<A>(
  cond: A -> bool, body: A -> Prob<A>
): A -> Prob<A> {
  (init: A) => (s: Bits) =>
  if fuel: nat :|
    !WhileBounded(fuel, cond, body, init)(s).Diverging?
  then WhileBounded(fuel, cond, body, init)(s)
  else Diverging
}
```

Unbounded while loop

Does the loop terminate for some amount of fuel?

Verifying While Loops?

- How can we prove that a loop produces `res` with probability `p`?
- **Idea:** reason about the bounded version (via induction)
- Take the **limit** `fuel` $\to \infty$

```
lemma {:axiom} WhileProbability<A>(
  cond: A -> bool,
  body: A -> Prob<A>,
  init: A,
  res: A,
  p: real
)
  requires !cond(res)
  requires ConvergesTo(
    (fuel: nat) => probMass(
      WhileBounded(fuel,cond,body,init), res),
    p)
  ensures probMass(While(cond, body)(init), res)
    == p
```

Required formalizing some real analysis in Dafny

Correctness Proof for Bernoulli(exp(−γ))

Can be proved with the previous lemma and a limit argument!

```
lemma BernExpCorrectness(gamma: real)
  requires 0.0 <= gamma <= 1.0
  ensures probMass(BernExp(gamma), true)
    == Exp(-gamma)
  ensures probMass(BernExp(gamma), false)
    == 1.0 - Exp(-gamma)
```

Required defining the exponential function in Dafny (partially axiomatized)

Dafny-VMC

- **Samplers** of various distributions
- Proofs of **correctness**
- https://github.com/dafny-lang/Dafny-VMC/

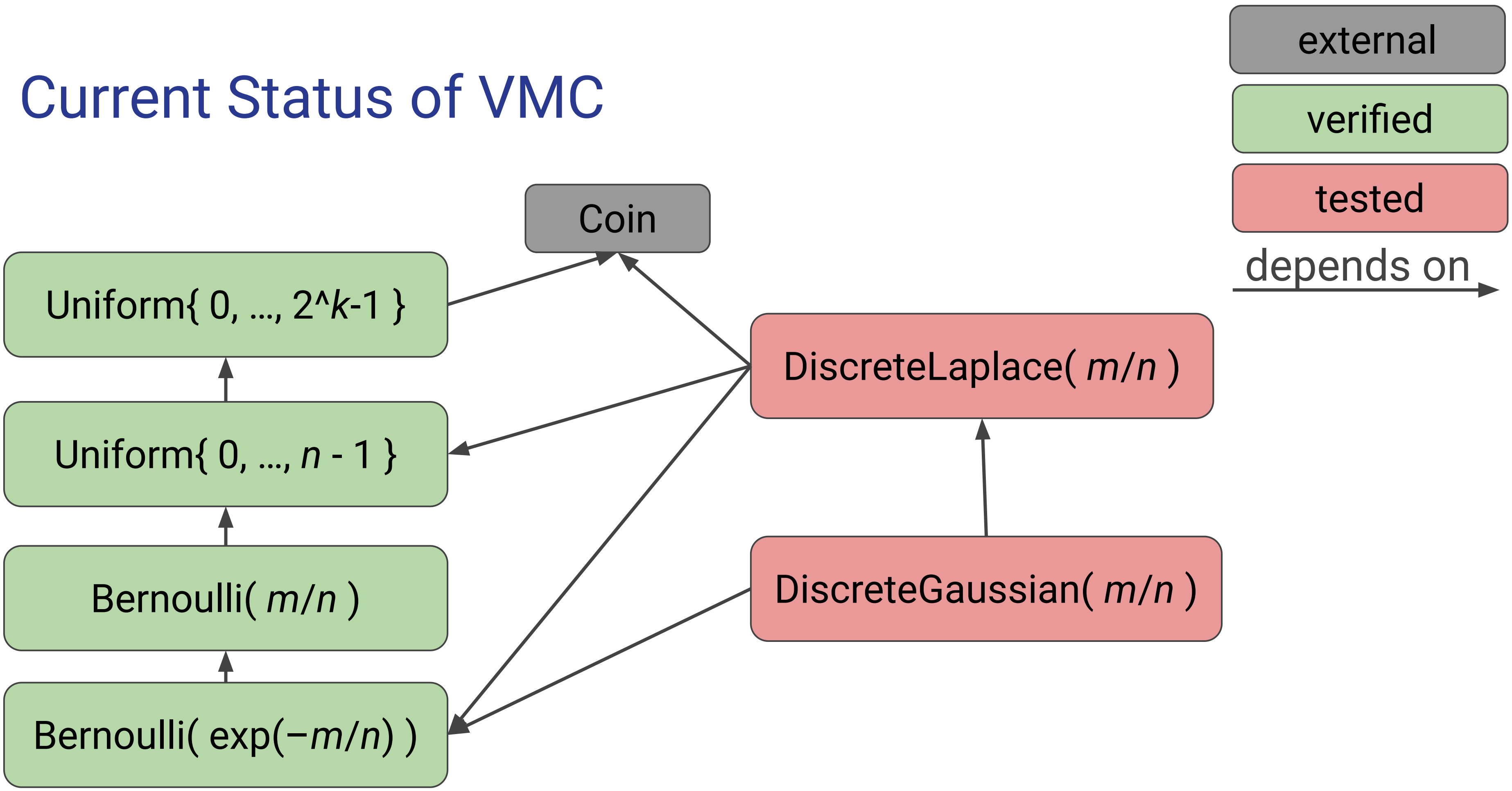
Verification of Bernoulli(exp(−γ)):

- Formalizing real analysis (limits & series)
- Probabilistic loops and nontermination

Questions?

Backup slides

Current Status of VMC



Probability in Dafny

- σ-algebra on bitstreams ("allowed events")
- Probability measure on bitstreams ("independent & uniformly distributed bits")
- Definition of probability spaces
- Hurd proved that bitstreams are a probability space (currently axiomatized)
- Can state correctness!

```
ghost const eventSpace :
iset<iset<Bits>>

ghost const prob: iset<Bits> -> real

ghost function probMass<A>(
  h: Hurd<A>, result: A): real {
  prob(iset s | h(s).0 == result)
}

ghost predicate IsProbSpace<A>(
  eventSpace: iset<iset<A>>,
  prob: iset<A> -> real)

lemma BitsIsProbSpace()
  ensures IsProbSpace(eventSpace, prob)

lemma CoinIsCorrect()
  ensures probMass(Coin(), false) == 0.5
  ensures probMass(Coin(), true) == 0.5
```

Imperative Sampler

- `SampleCoin` relies on an external random source
- E.g.: Java random number generator
- Other imperative samplers use `SampleCoin` as a primitive

```
trait CoinSampler {
  ghost var s: Rand.Bitstream

  method {:extern} SampleCoin()
    returns (b: bool)
    modifies this
    ensures Coin(old(s))==(b, s)
}
```

Assumption: external random source behaves like the model!

Structure of Probabilistic Samplers in Dafny

1. **Functional Model**
2. **Correctness proof**
3. **Imperative implementation**
4. **Proof of correspondence**
5. **Statistical tests**

```
function Uniform(n: nat): Hurd<nat>
  requires n >= 1
{ ... }

lemma UniformCorrect(n: nat)
  ensures forall i: nat :: 0 <= i < n ==>
    probMass(Uniform(n), i) == 1.0 / n as real
{ ... }

trait UniformSampler {
  ghost var s: Bitstream

  method SampleUniform(n: nat)
    returns (i: nat)
    modifies this
    requires n >= 1
    ensures Uniform(n)(old(s))==(i, s)
  { ... }
}

method {:test} TestUniform() { ... }
```

Axiomatizations

- Measure theory
- Construction of the probability space on `Bits`
- Measurability and independence of probabilistic primitives
- Properties of the exponential function
    - Functional equation: $\exp(x)\exp(y) = \exp(x + y)$
    - Convergence of its power series

Future Work: Representing Probabilistic Computations

Hurd monad:

- **Pros:** easy to relate imperative code and functional model
- **Cons:** hard to prove correctness (in particular, independence), need to thread bitstreams through the proof

Can we use a different probability monad?

- splittable RNG?
- Giry monad?