Formally proving crypto properties of pseudorandom number generators

Katherine Ye '16
advised by Andrew Appel and Matt Green


DILBERT By Scott Adams
TOUR OF ACCOUNTING
OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.
scottadams@aol.com
www.dilbert.com
NINE NINE NINE NINE NINE NINE
© 2001 United Feature Syndicate, Inc.
ARE YOU SURE THAT'S RANDOM?
THAT'S THE PROBLEM WITH RANDOMNESS: YOU CAN NEVER BE SURE.

- Most modern cryptosystems rely on random numbers
- e.g. RSA generates random big primes that become a private key
- Reducing the entropy of a cryptosystem's pseudo-random number generator (PRG) is an easy way to break the entire cryptosystem

Random number generator

RNG

↓

1011100110101011000010110010000011111011110001111100110111010000000010

Pseudo-random number generator

1100101

PRG

111111011111101001010110011010001000111101111110101110001010101000110000

Pseudo-random number generator

1100101

$$\underbrace{\texttt{1100101}}$$

PRG

1111110111111010010101100110100010001111011111101011100010101010 0011000

≈

0010111001101010110000101100100000111111011110001111100110111010000000001

Pseudo-random number generator

1100101

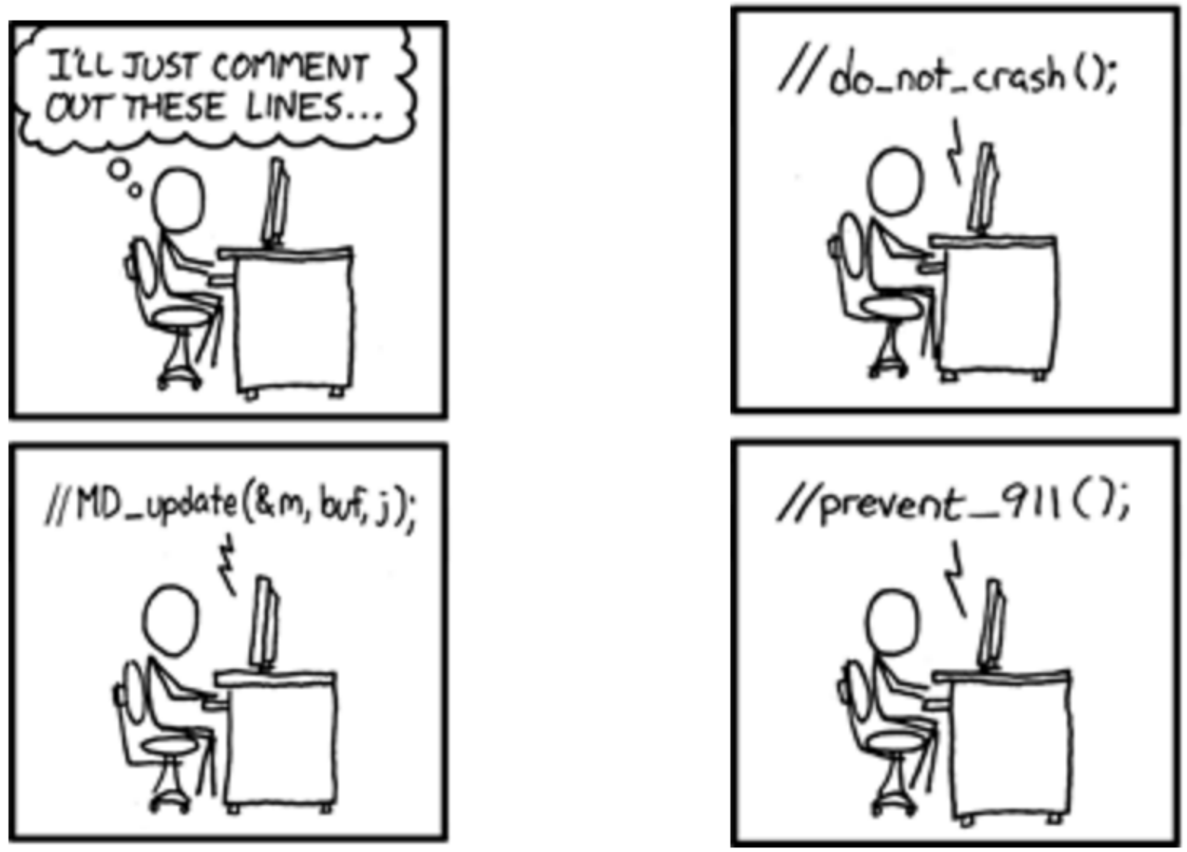
PRG

11111101111110100101011001101000100011110111111010111000101010100011000

! ≈

0010111001101010110000101100100000111111011110001111100110111010000000001

Debian OpenSSL PRG



https://www.xkcd.com/424/

- Removed sources of system entropy → only 32,767 choices
- Predictable SSL/SSH keys (Spotify, Yandex...)
- Can read encrypted traffic, log into remote servers, forge messages
- Have to patch servers AND replace weak keys

https://freedom-to-tinker.com/blog/kroll/software-transparency-debian-openssl-bug/

- **We need secure PRGs**

- But surprisingly little work exists on proving PRGs secure, either on paper or formally

Until now!

Goal: formally prove **functional correctness** and **cryptographic security** of a widely-used implementation of a PRG

Goal: prove **functional correctness** and **cryptographic security** of a widely-used implementation of a PRG

mbedTLS

HMAC-DRBG

Verified Software Toolchain

Foundational Crypto Framework

Goal: prove **functional correctness** and **cryptographic security** of a widely-used implementation of a PRG

mbedTLS

HMAC-DRBG

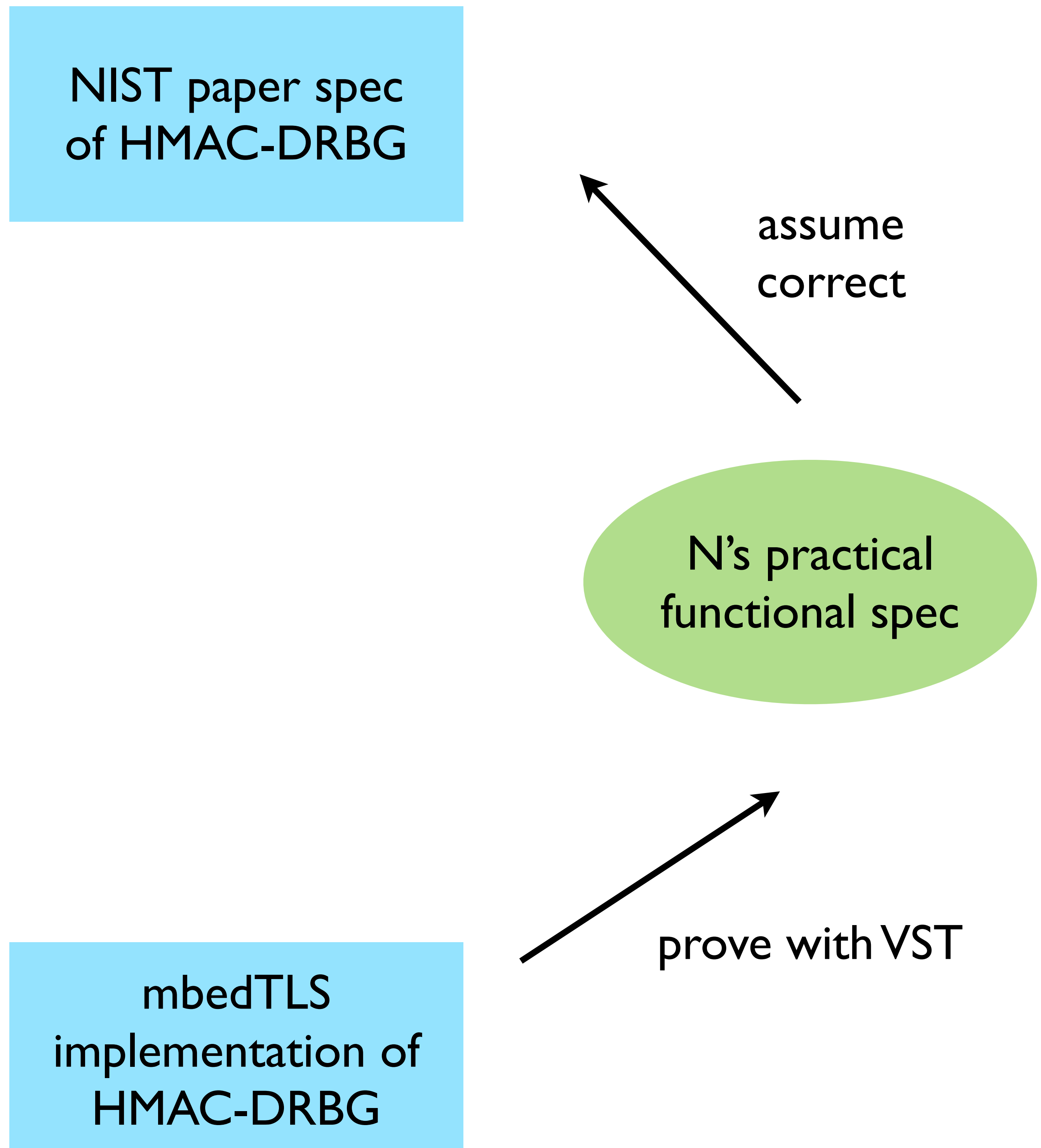
Our project

NIST paper spec
of HMAC-DRBG

mbedTLS
implementation of
HMAC-DRBG

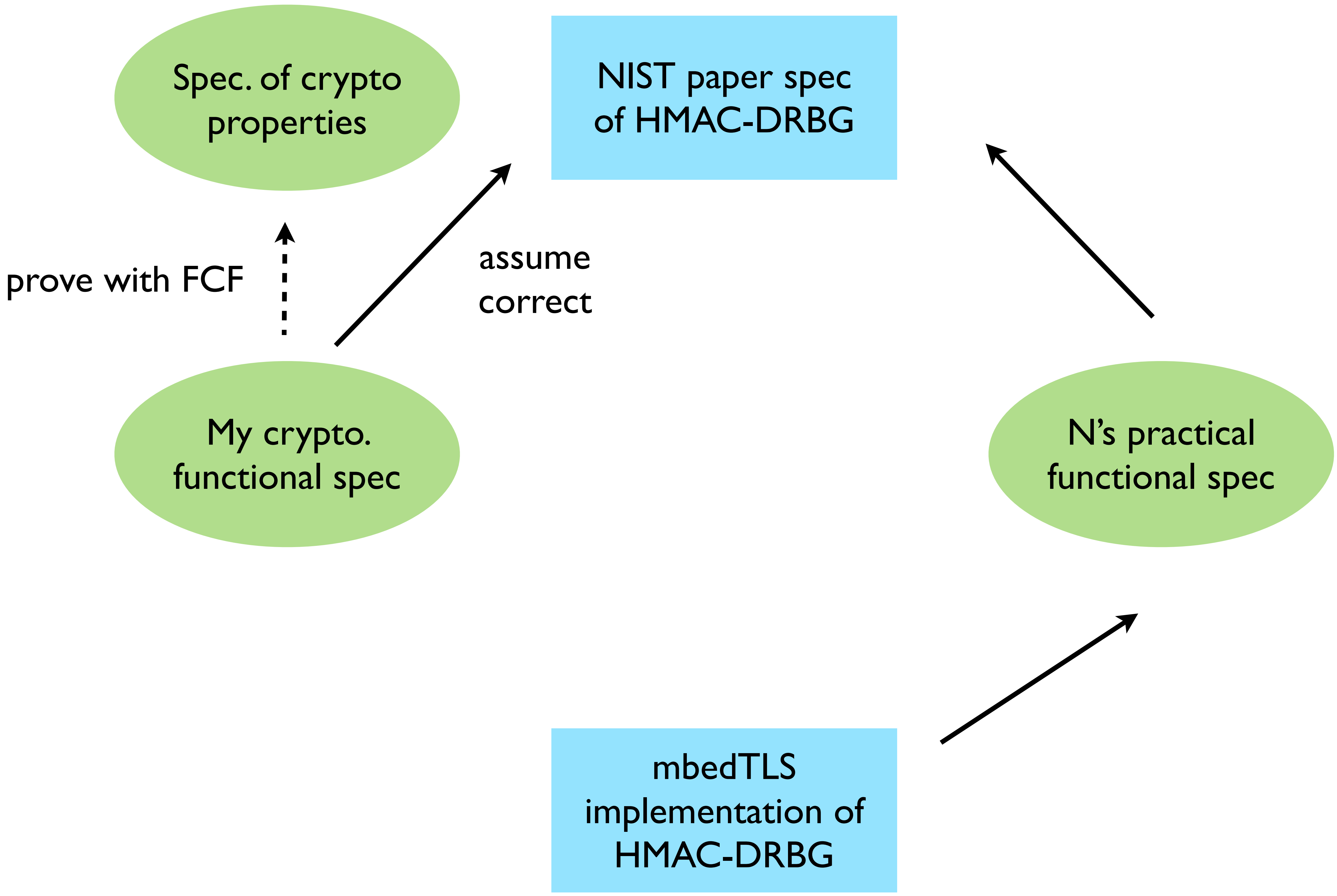
Our project

x → y:
x implements y



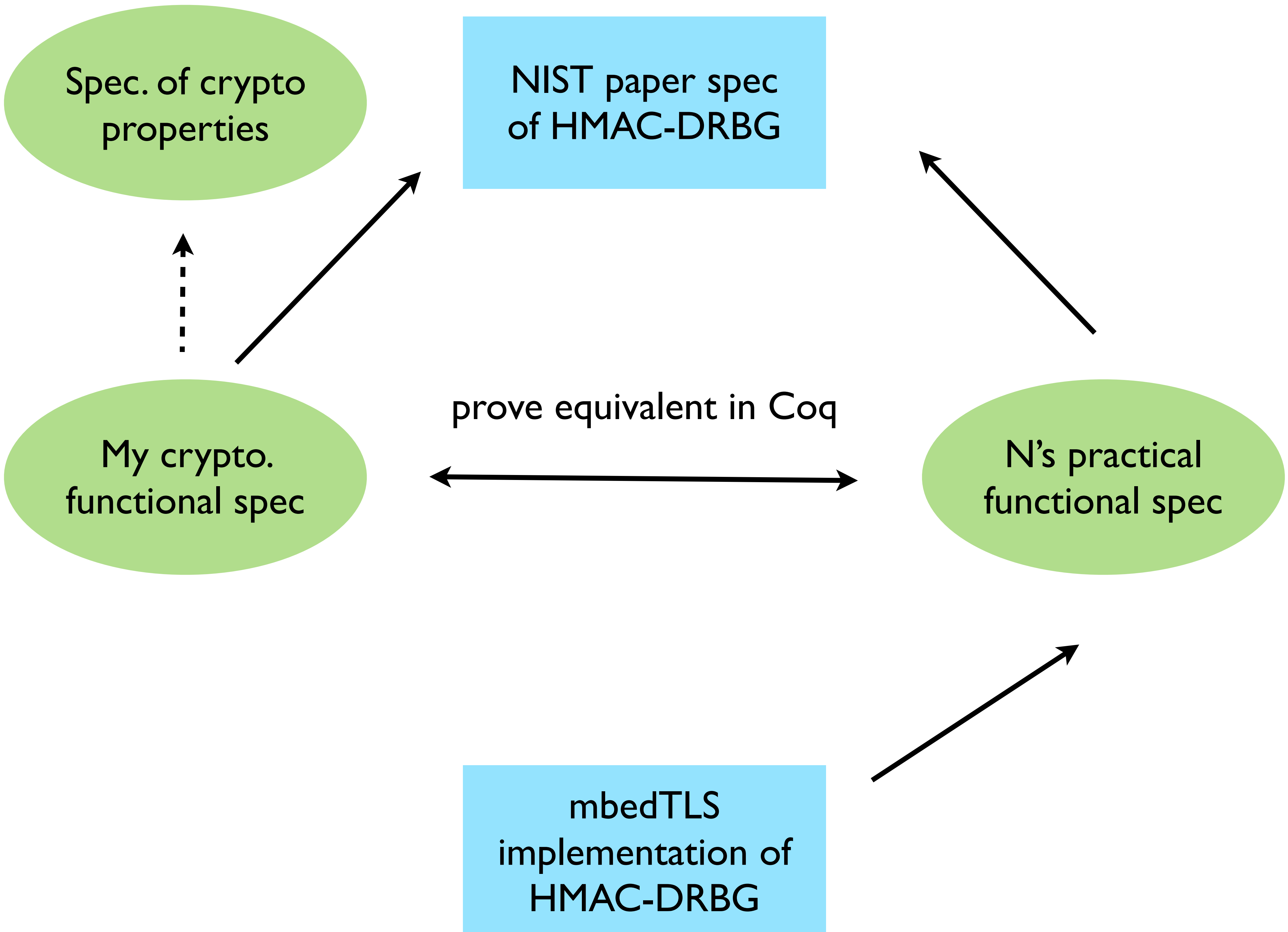
Our project

x → y:
x implements y

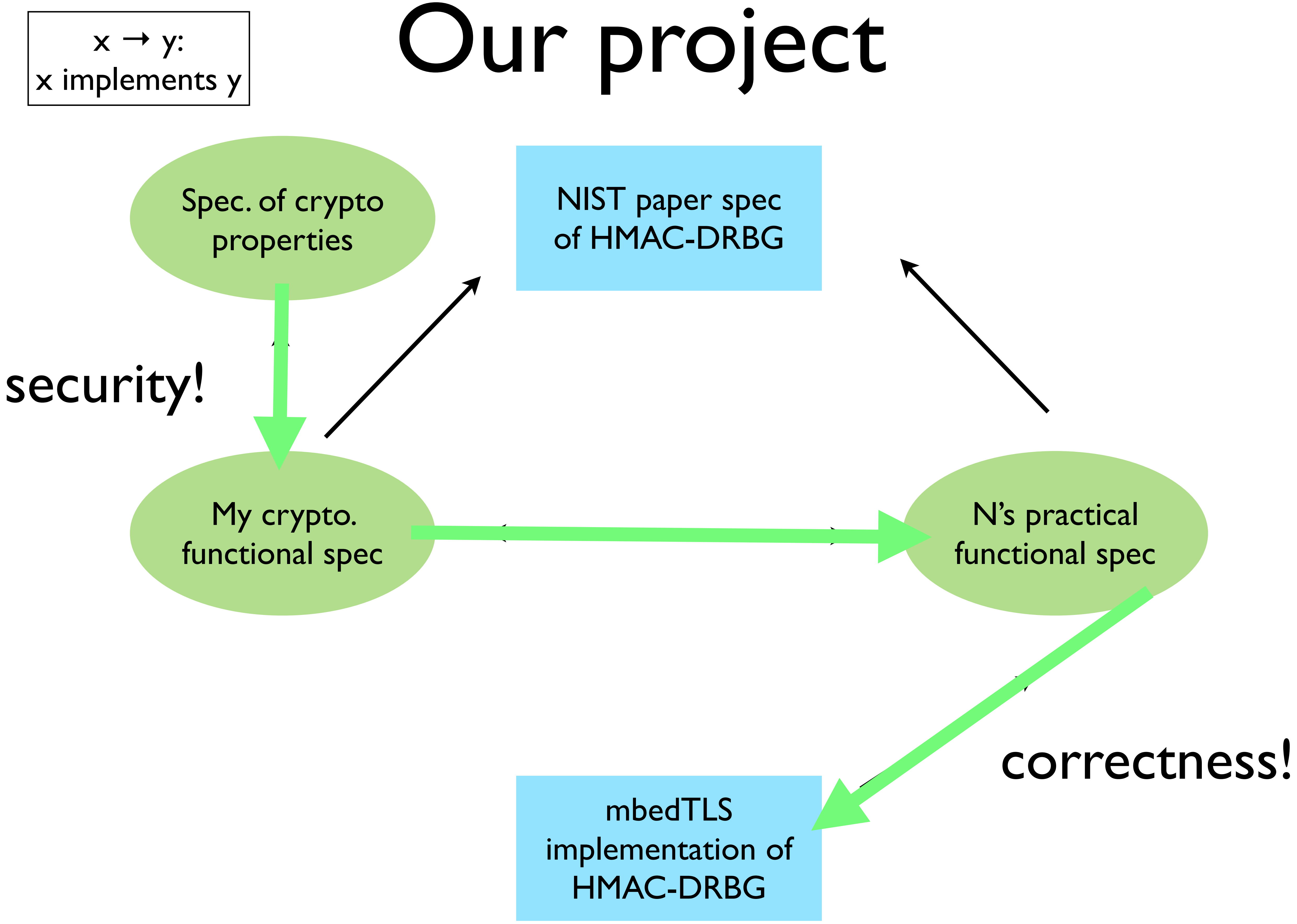


x → y:
x implements y

Our project

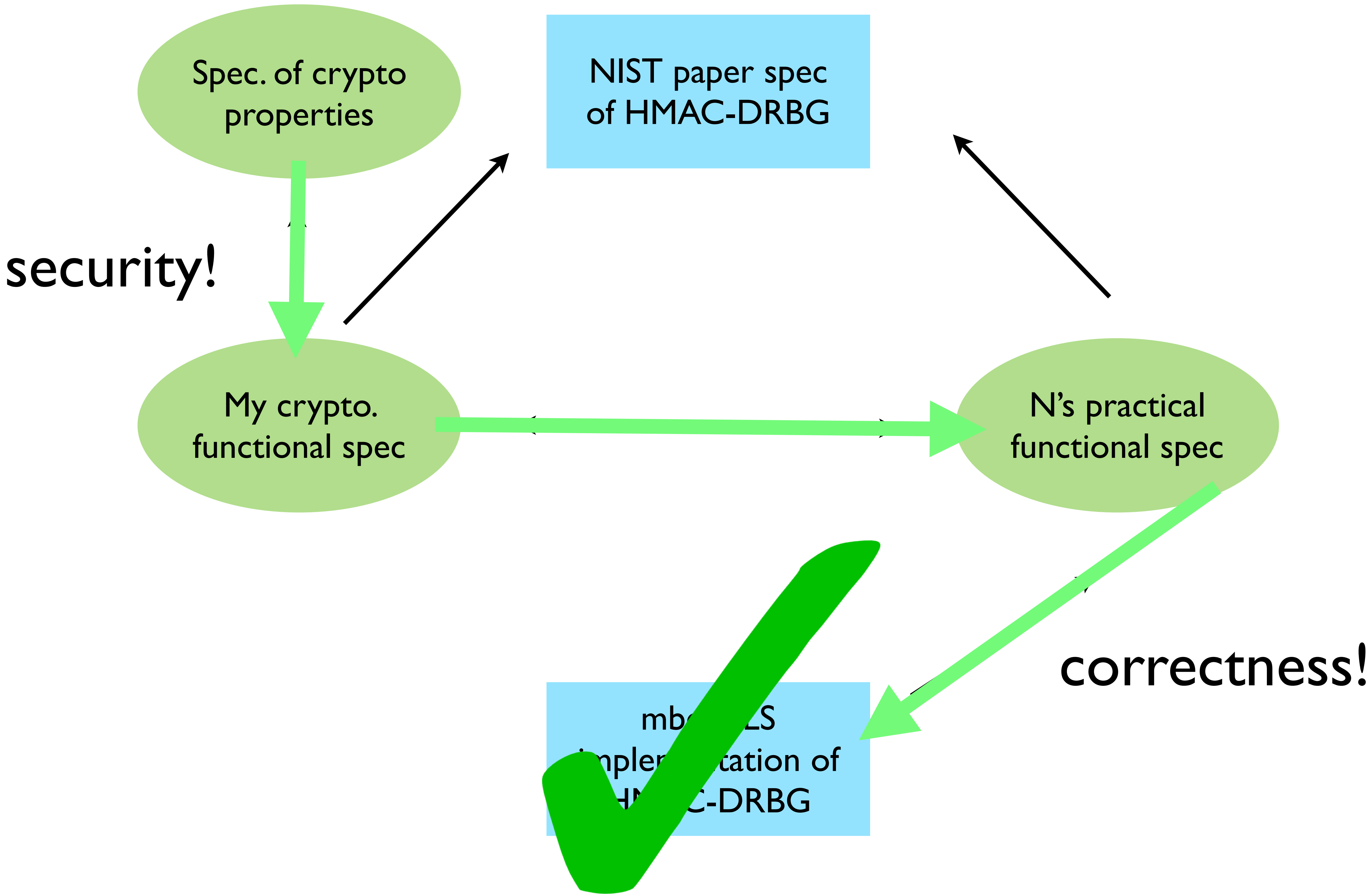


Our project


x → y:
x implements y
Spec. of crypto properties
NIST paper spec of HMAC-DRBG
security!
My crypto. functional spec
N's practical functional spec
correctness!
mbedTLS implementation of HMAC-DRBG

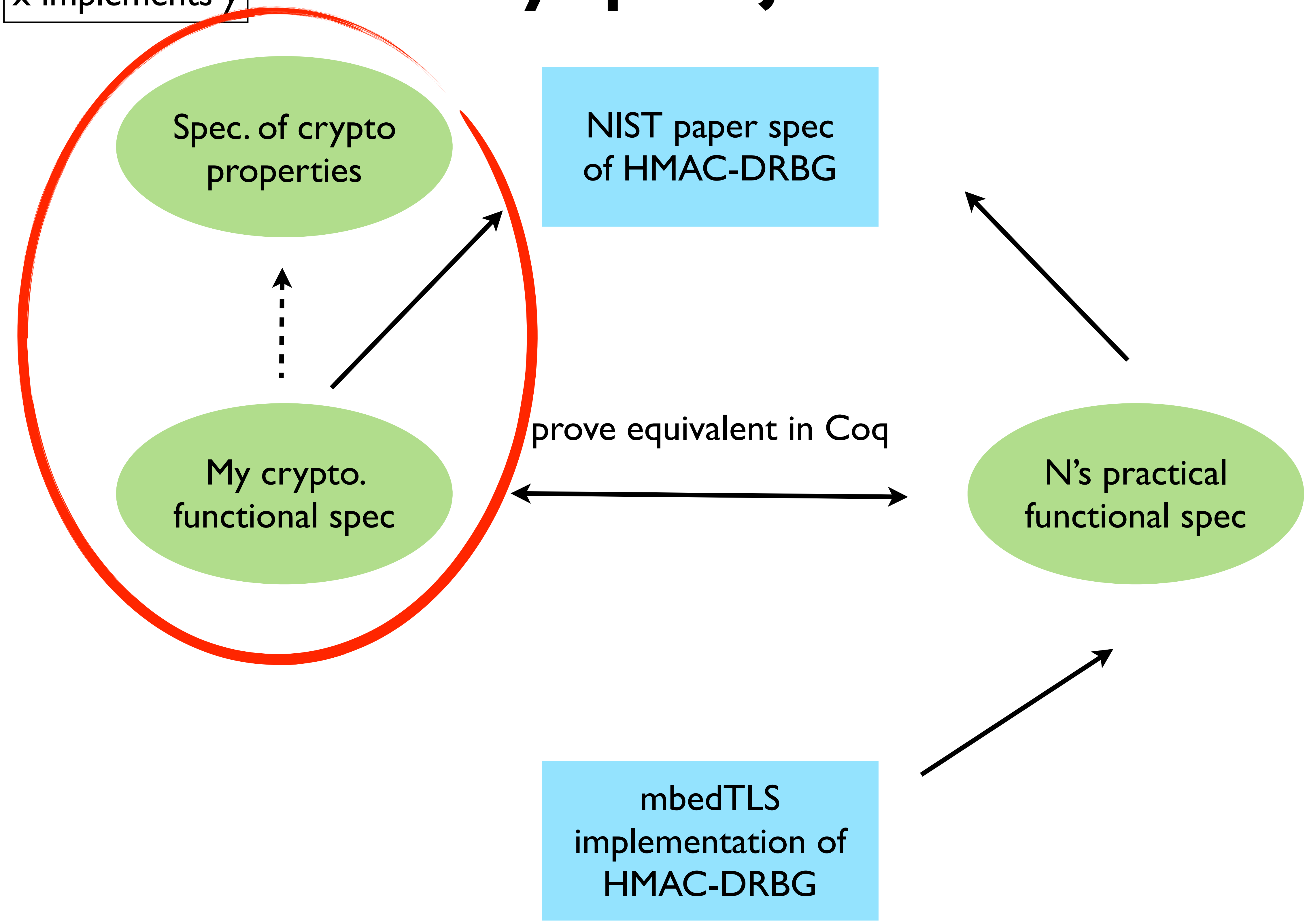
x → y:
x implements y

Our project



My project

x → y:
x implements y



Security properties of PRGs

- Output indistinguishable from random to a computationally-bounded adversary

Security properties of PRGs

- Backtracking-resistant (compromise at time t does not compromise output from time < t)
- Eventually recovers from compromises of internal state

Related work

(there isn't much)

Our group

- Appel (2015) does the first "full formal machine-checked verification of a C program: the OpenSSL implementation of SHA-256."
- Petcher, Beringer, Ye, and Appel (2015) do the same for HMAC, adding a proof of crypto security depending on SHA

[Verification of a Cryptographic Primitive: SHA-256]

[Verified Correctness and Security of OpenSSL HMAC]

Our group

- Appel (2015) does the first "full formal machine-checked verification of a C program: the OpenSSL implementation of SHA-256."
- Petcher, Beringer, Ye, and Appel (2015) do the same for HMAC, adding a proof of crypto security depending on SHA

hence HMAC-DRBG

[*Verification of a Cryptographic Primitive: SHA-256*]

[*Verified Correctness and Security of OpenSSL HMAC*]

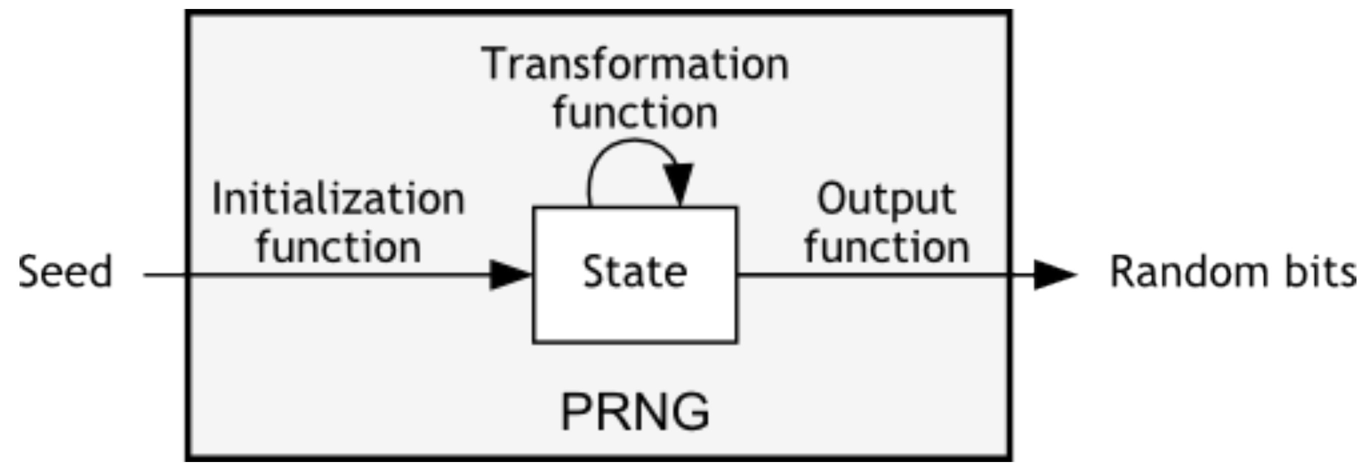
Paper proofs

- One proof by Hirose (2009) about HMAC-DRBG; not peer-reviewed
- Several crypto papers analyze the security of PRGs and propose new security properties, e.g. Dodis et al.

http://repo.flib.u-fukui.ac.jp/dspace/bitstream/10098/2126/1/art.pdf
https://eprint.iacr.org/2013/338.pdf

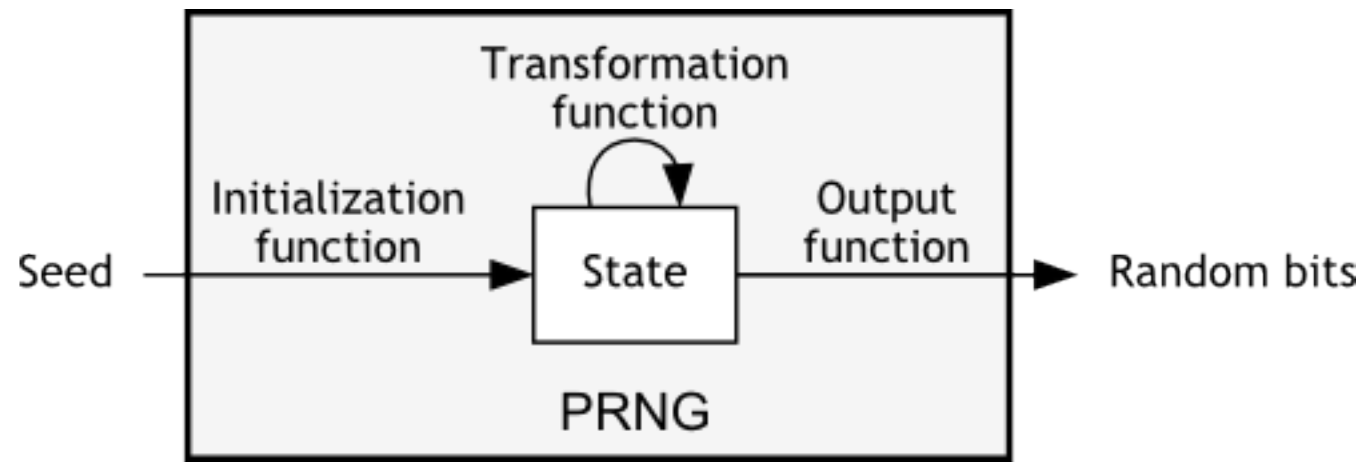
PRG internals

Pseudo-random number generator



http://pit-claudel.fr/clement/blog/how-random-is-pseudo-random-testing-pseudo-random-number-generators-and-measuring-randomness/

Pseudo-random number generator




```
Instantiate
Generate (bits)
Reseed (add entropy)
Update (internal state)
```

http://pit-claudel.fr/clement/blog/how-random-is-pseudo-random-testing-pseudo-random-number-generators-and-measuring-randomness/

Generate (simplified)

Chaining:
K = secret key; V = initialization vector;
H = hash function (e.g. HMAC); || = concatenate

```
rand_bits =
H(K, V)
|| H(K, H(K, V))
|| H(K, (H(K, H(K, V)))...
```

outputs used again as inputs

Generate (simplified)

n blocks of output: recursion

```
rec loop K V n =
  if n = 0 then ([], V)
  else
    let (result, V’) := loop K V (n-1) in
    let V’’ := HMAC K V’ in
    (result ++ V’’, V’’)

fun Generate K V n reseed_ctr =
  if reseed_ctr >= max then reseed_required
  else
    let (bits, V’) := loop K V n in
    let (K’, V’’) := Update K V’ in
    (K’, V’’, bits)
```

PRG run

```
User/Adversary:

Instantiate,
Generate 10 blocks,
Generate 20 blocks,
Generate 1 block,
Generate 10000000 blocks,
Generate 1 block,
...
```

Another loop

PRG run

```
User/Adversary:

Instantiate,
Generate 10 blocks,
  Update K and V
Generate 20 blocks,
  Update K and V
Generate 1 block,
  Update K and V,
Generate 10000000 blocks,
  Update K and V,
  RESEED,
Generate 1 block,
  Update K and V,
...
```

Complications with Updating key and Reseed

First steps

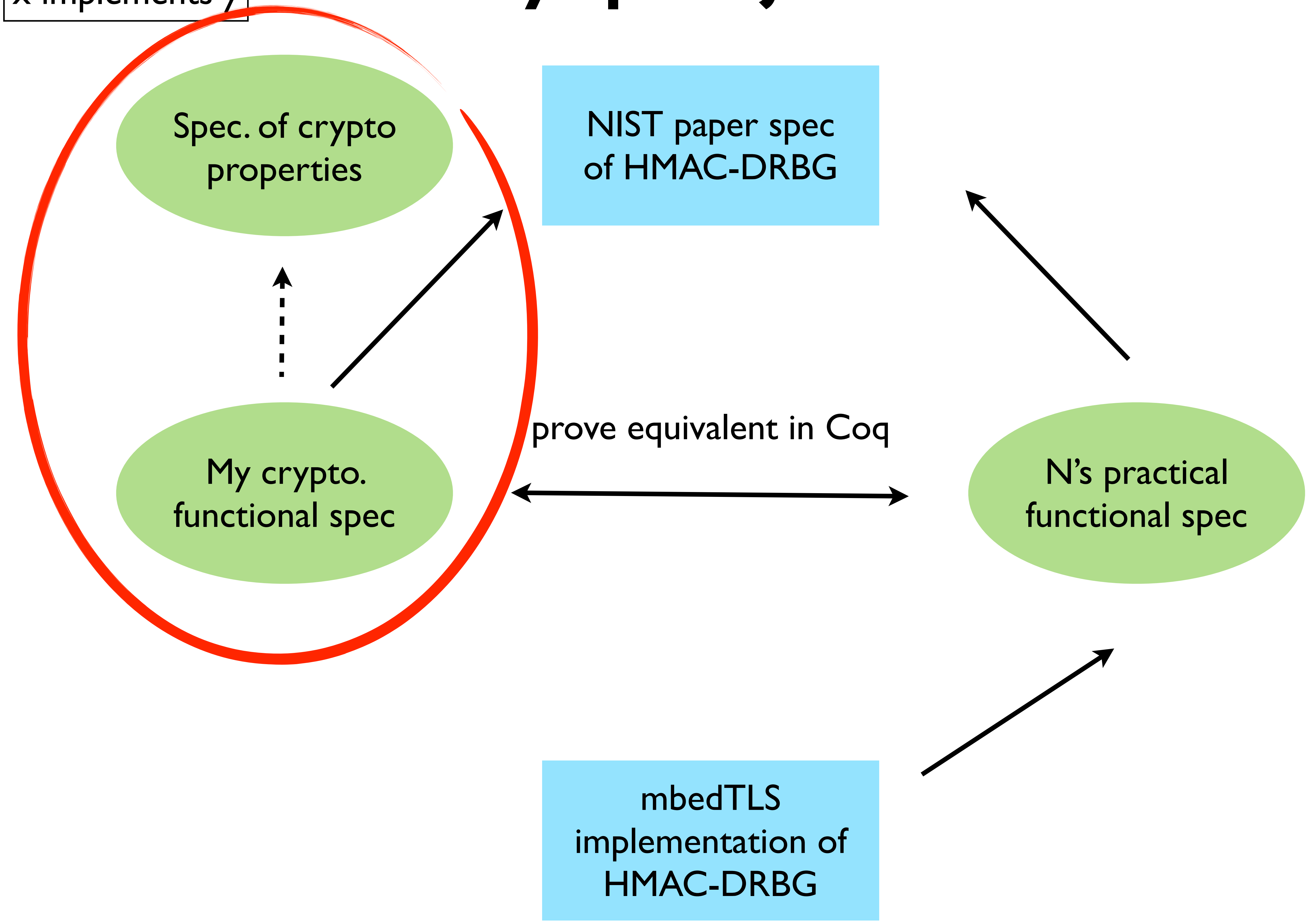
- Proof of indistinguishability for inner loop of PRG (Generate function): done by collaborator
- Extend to proof of indistinguishability for outer loop of PRG (multiple Generate calls with Update): working on

Method

- Proofs in the "sequence of games" style
- Bound probability of adversary distinguishing correctly by 1/2 + negligible amount
- Done in FCF in Coq, so correctness is verified

My project

x → y:
x implements y



Progress to date

Progress

- Matt and I have a proof outline for HMAC-DRBG indistinguishability written in text
- Started (separate) proof of entropy as a function of user number of calls

Progress

- Steep learning curve for FCF; also learning crypto as I go
- Collaborator wrote a bare-bones proof for the inner loop
- Collaborator gave us an outline for hybrid game strategy
- Studying the above; started formalizing our proof!

To do

- Formalize simplified HMAC-DRBG proof
- Add features to proof (e.g. additional input)
- Write full functional spec of HMAC-DRBG
- Connect it with concrete functional spec

Measure of success

Questions

- How automated?
- How much effort? (time, lines of code?)
- Did we contribute original math?
- Note: we hope to publish a paper on the whole system.

Measures of success

- How many properties were we able to verify, and how important are they?
- Is our verification actually right?

Measures of success

- What attacks can be definitively ruled out by our verification? What attacks are still possible?
- Are the security and formal verification communities excited about using or building on our work?

Thanks!