Verified Correctness and Security of mbedTLS HMAC-DRBG

Katherine Ye, Matthew Green, Naphat Sanguansin,
Lennart Beringer, Adam Petcher, and Andrew Appel

*Princeton / CMU, Johns Hopkins, Princeton / Dropbox,
Princeton, Oracle, Princeton*

ACM CCS 2017

Most modern cryptosystems rely on high-quality randomness.

e.g. RSA generates random big primes that are used to compute a private key

Pseudorandom number generator

1100101

PRG

0111111011111101001010110011010001000111101111110101110001010101000110001110100

Pseudorandom number generator

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

PRG

$$\overbrace{\texttt{01111111011111110100101011001101000100011110111111010111000101010100011000111010 0}}$$

≈

0000101110011010101100001011001000001111110111100011111001101110100000000 1000011

Pseudorandom number generator

$$\underbrace{1100101}$$

PRG

0111111101111111010010101100110100010001111011111110101110001010101000110001110100

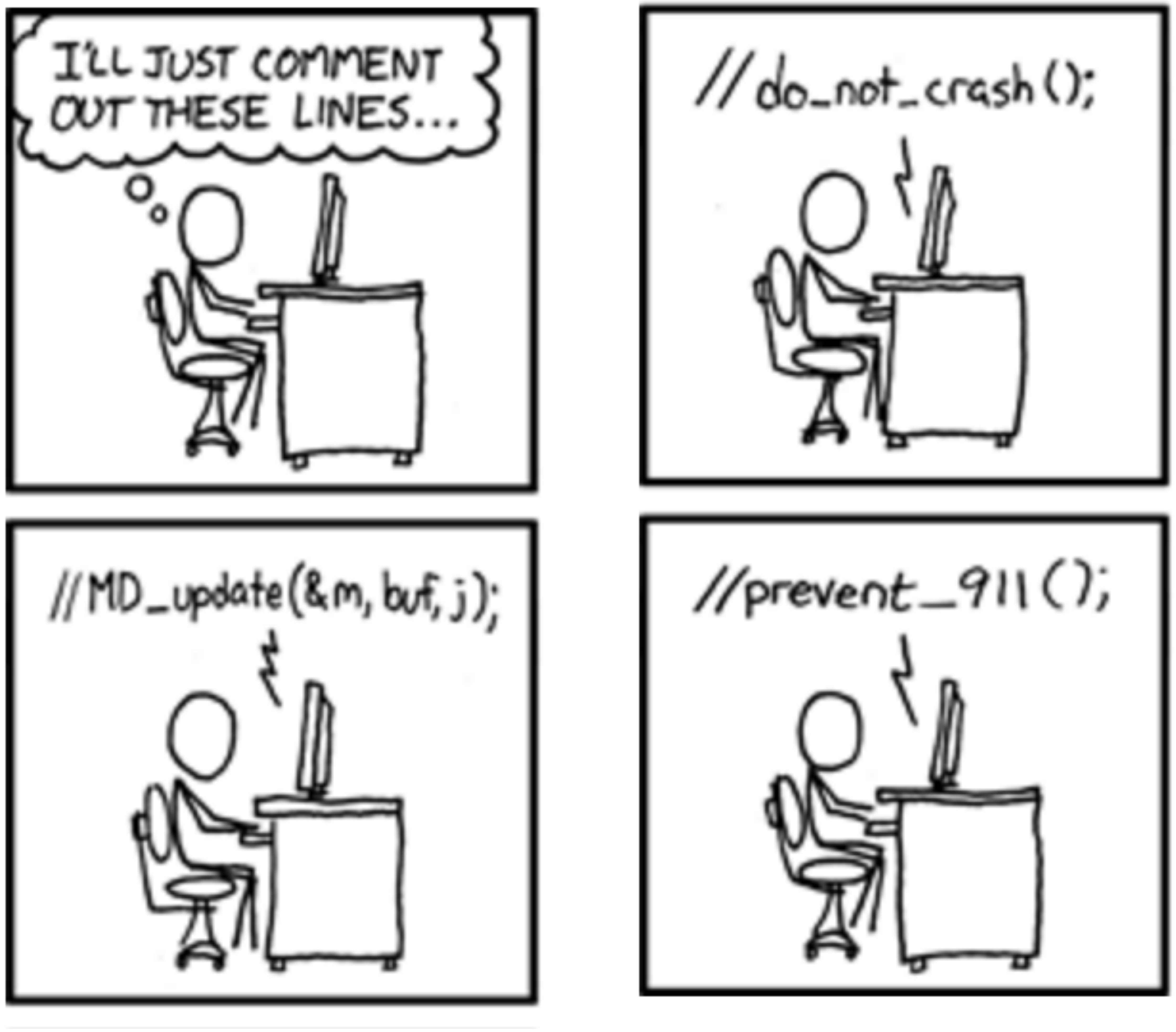
! ≈

0000101110011010101100001011001000001111110111110001111100110111010000000001000011

Reducing the entropy of a cryptosystem's pseudorandom number generator (PRG) is an easy way to break the entire cryptosystem.

Dual-EC-DRBG

Debian OpenSSL PRG



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

- Removed sources of system entropy → only 32,767 choices (process ID!)
- Predictable SSL/SSH keys (Spotify, Yandex...)
- Can read encrypted traffic, log into remote servers, forge messages

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

We need secure PRGs

But how?


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.

Our work

Foundational Crypto Framework

Verified Software Toolchain

Proved **functional correctness** and **cryptographic security** of a widely used implementation of a PRG

mbedTLS

HMAC-DRBG

PRG security property

- Proved that output is indistinguishable from random to a computationally bounded adversary, subject to assumptions
- Typical real/ideal indistinguishability proof in the computational model, using a hybrid argument on number of PRG calls
- Derived a concrete bound on advantage

Foundational
Cryptography
Framework

Coq framework for reasoning about the security of cryptographic schemes using a probabilistic programming language

The Foundational Cryptography Framework, Petcher and Morrisett (POST '15)

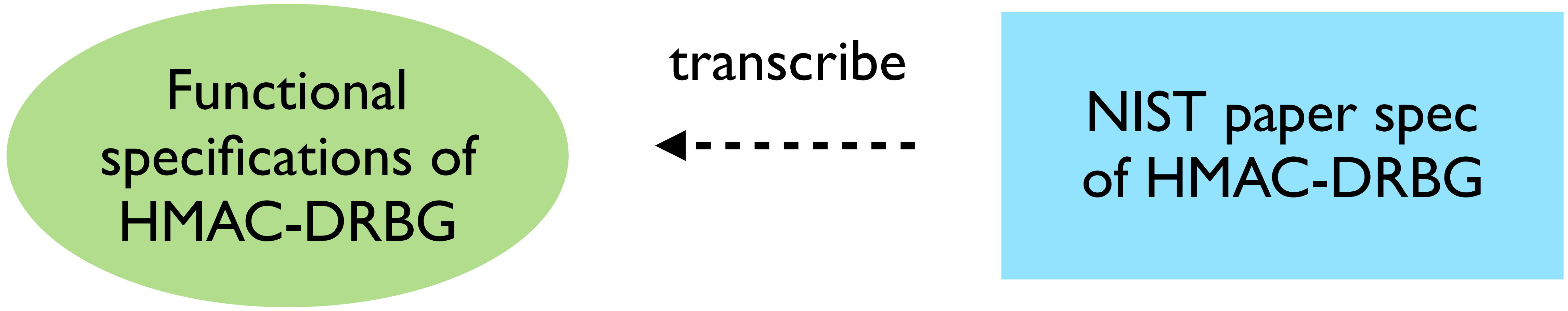
Verified
Software
Toolchain

Coq framework for verifying the correctness of C programs (pointers, mutable state, etc.)

Program Logics for Certified Compilers, Appel et al. (2014)

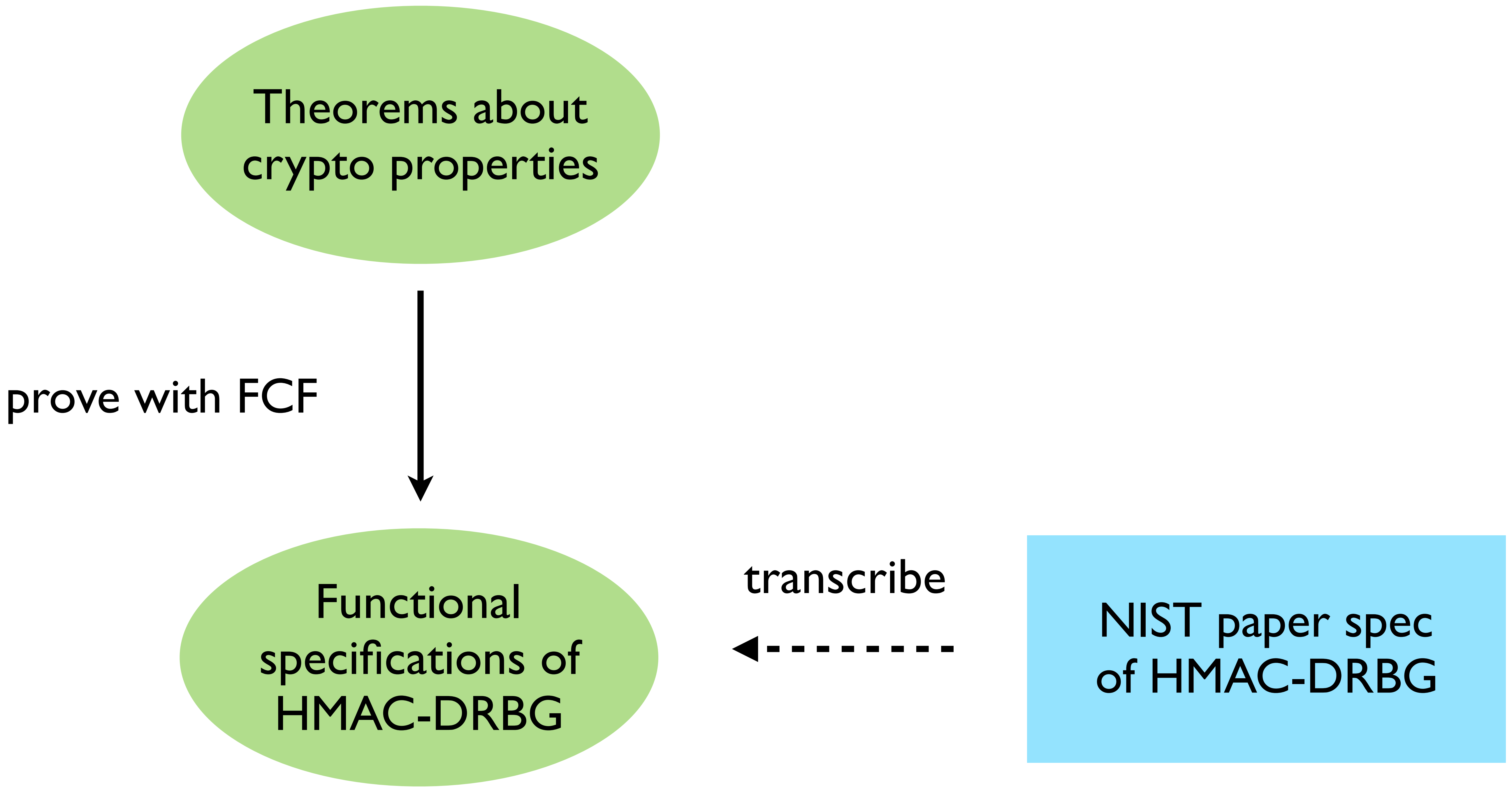
Our work

x → y:
x implements y

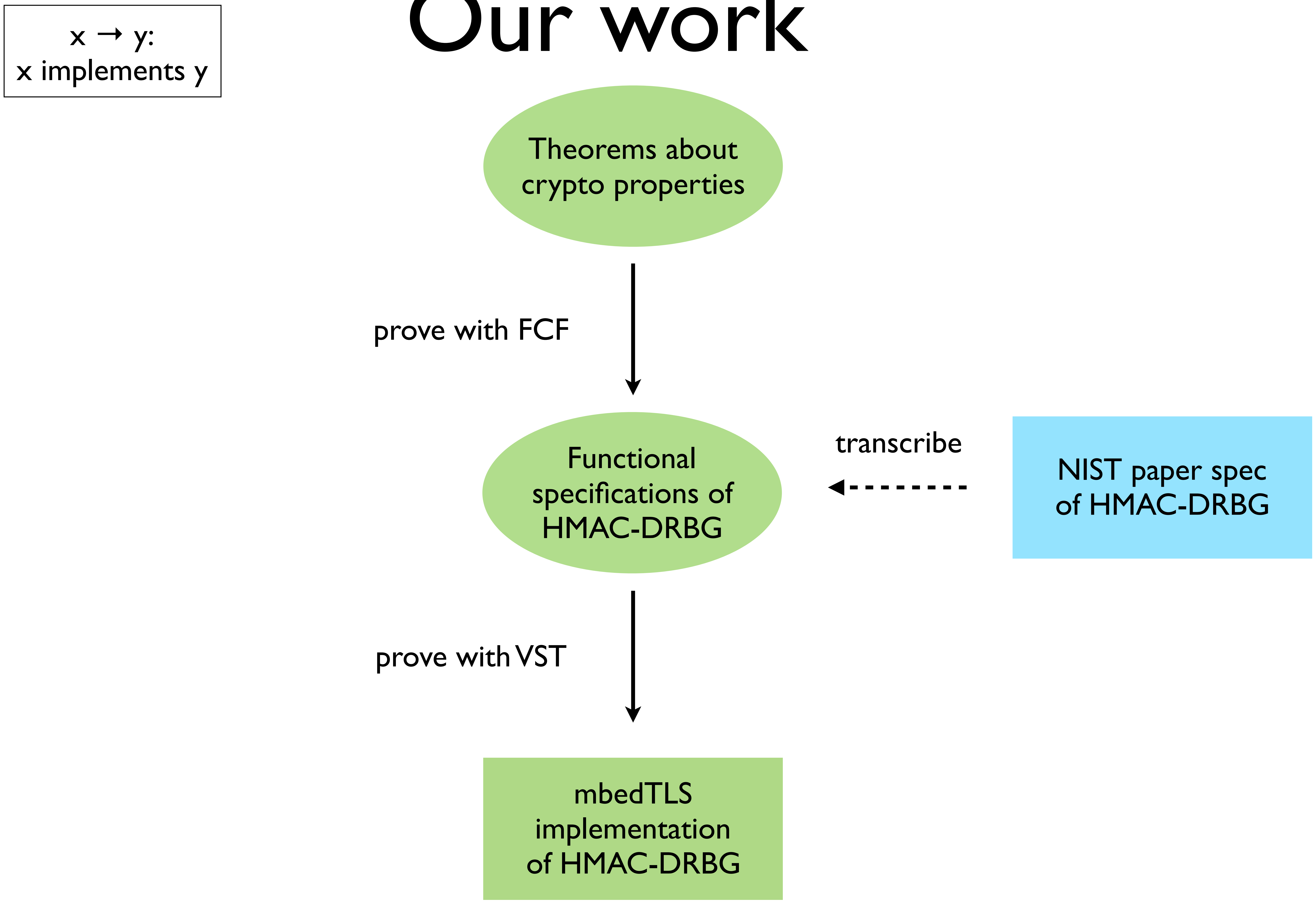


Our work

x → y:
x implements y

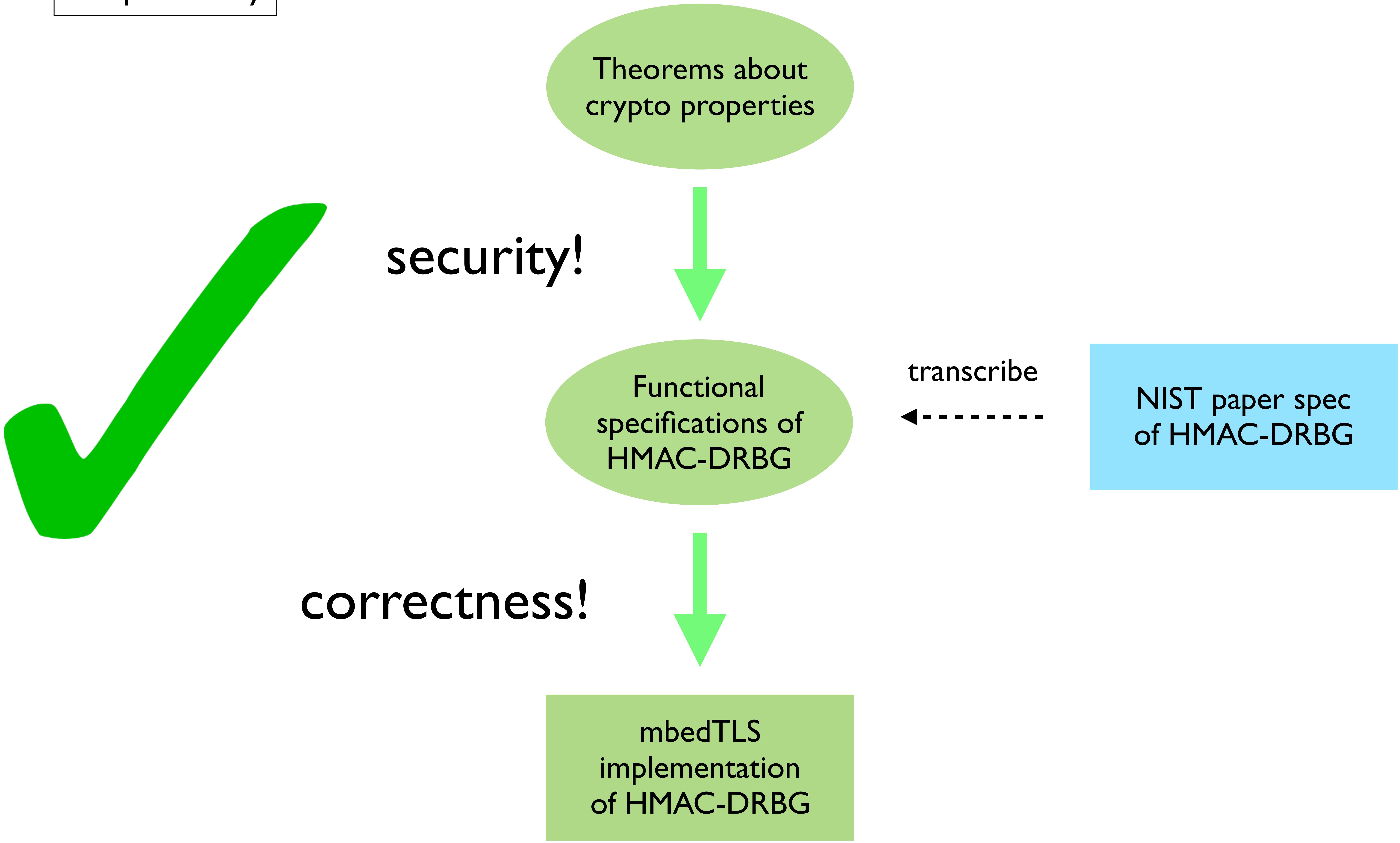


Our work

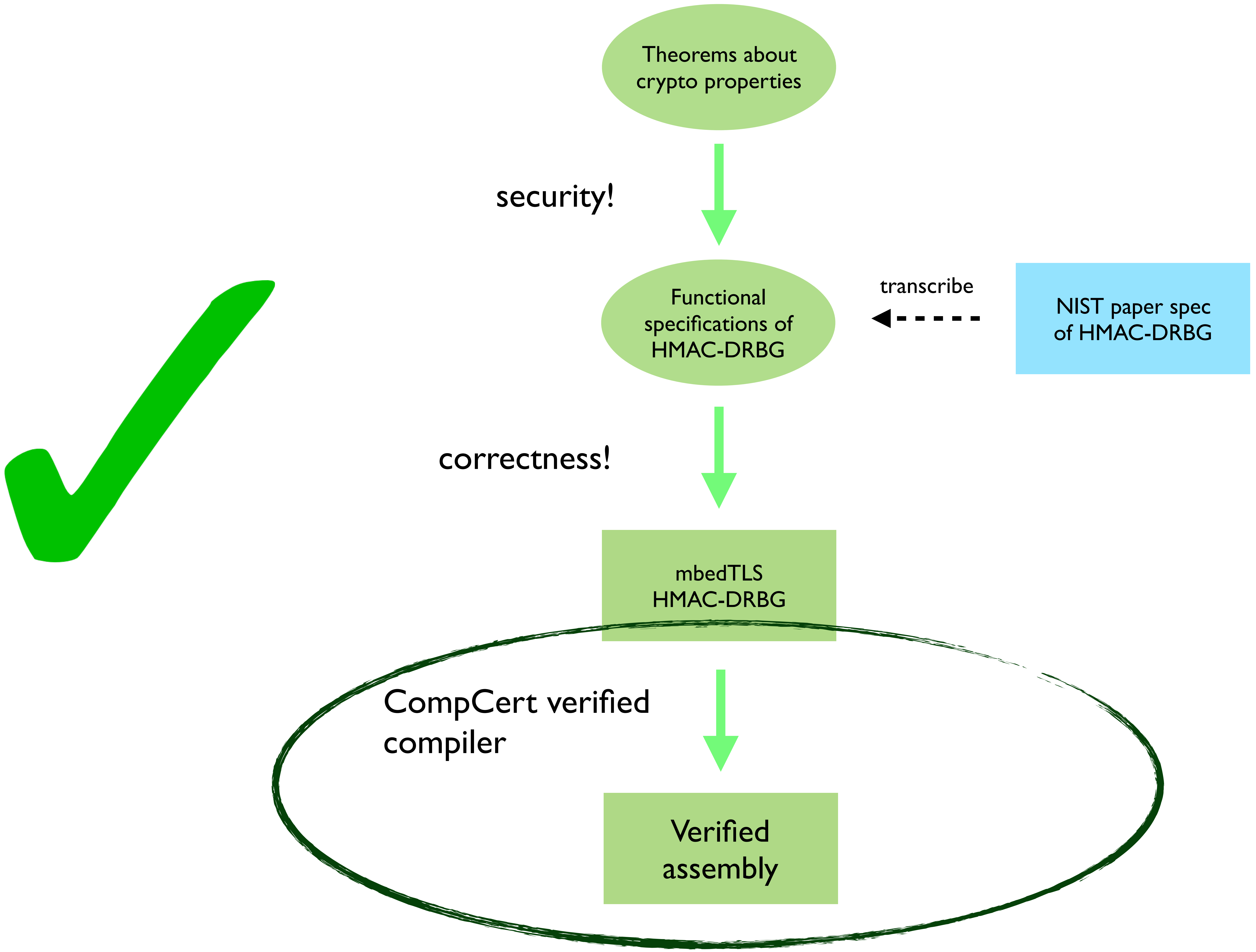

x → y:
x implements y
Theorems about crypto properties
prove with FCF
Functional specifications of HMAC-DRBG
transcribe
NIST paper spec of HMAC-DRBG
prove with VST
mbedTLS implementation of HMAC-DRBG

Our work

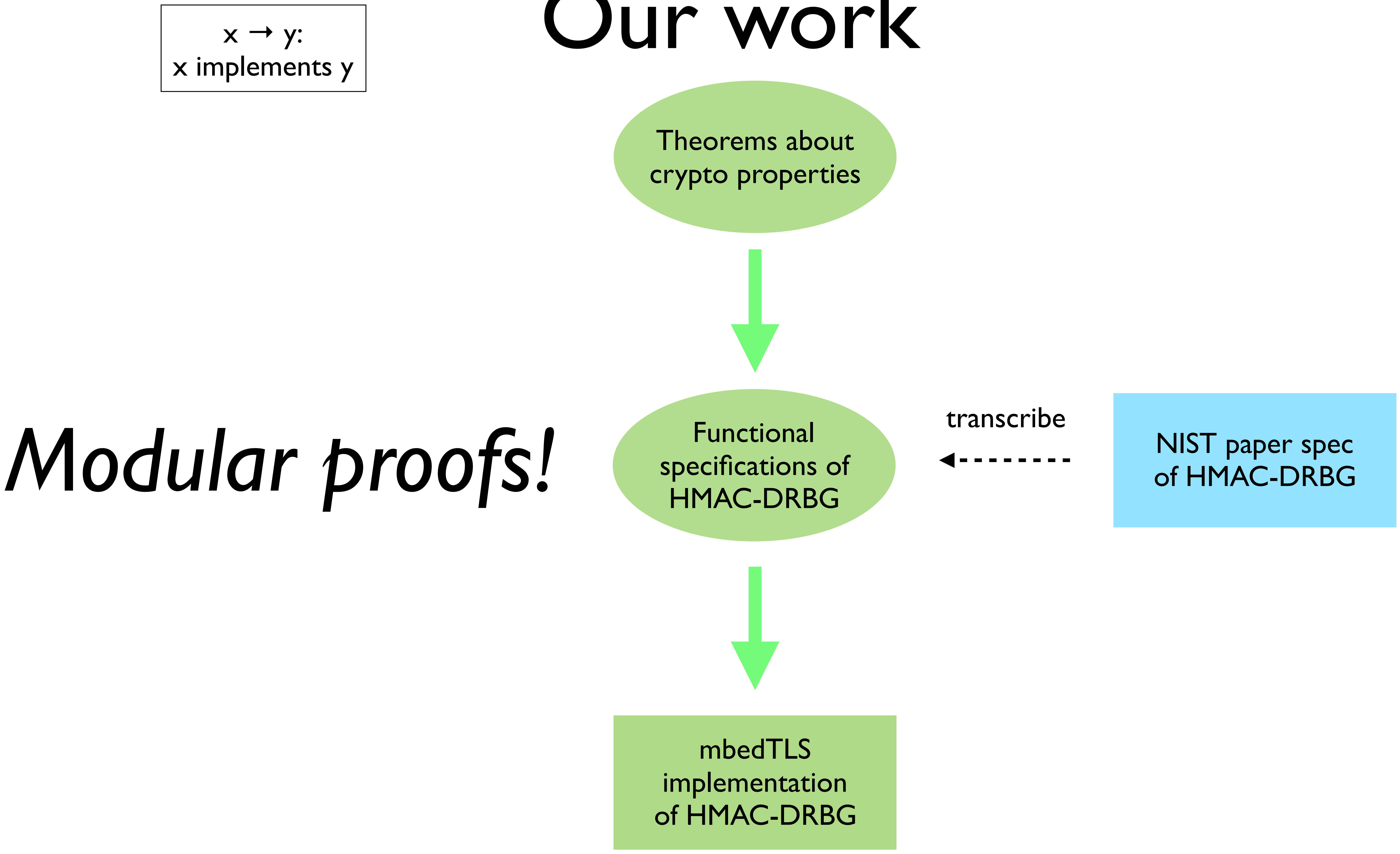
x → y:
x implements y



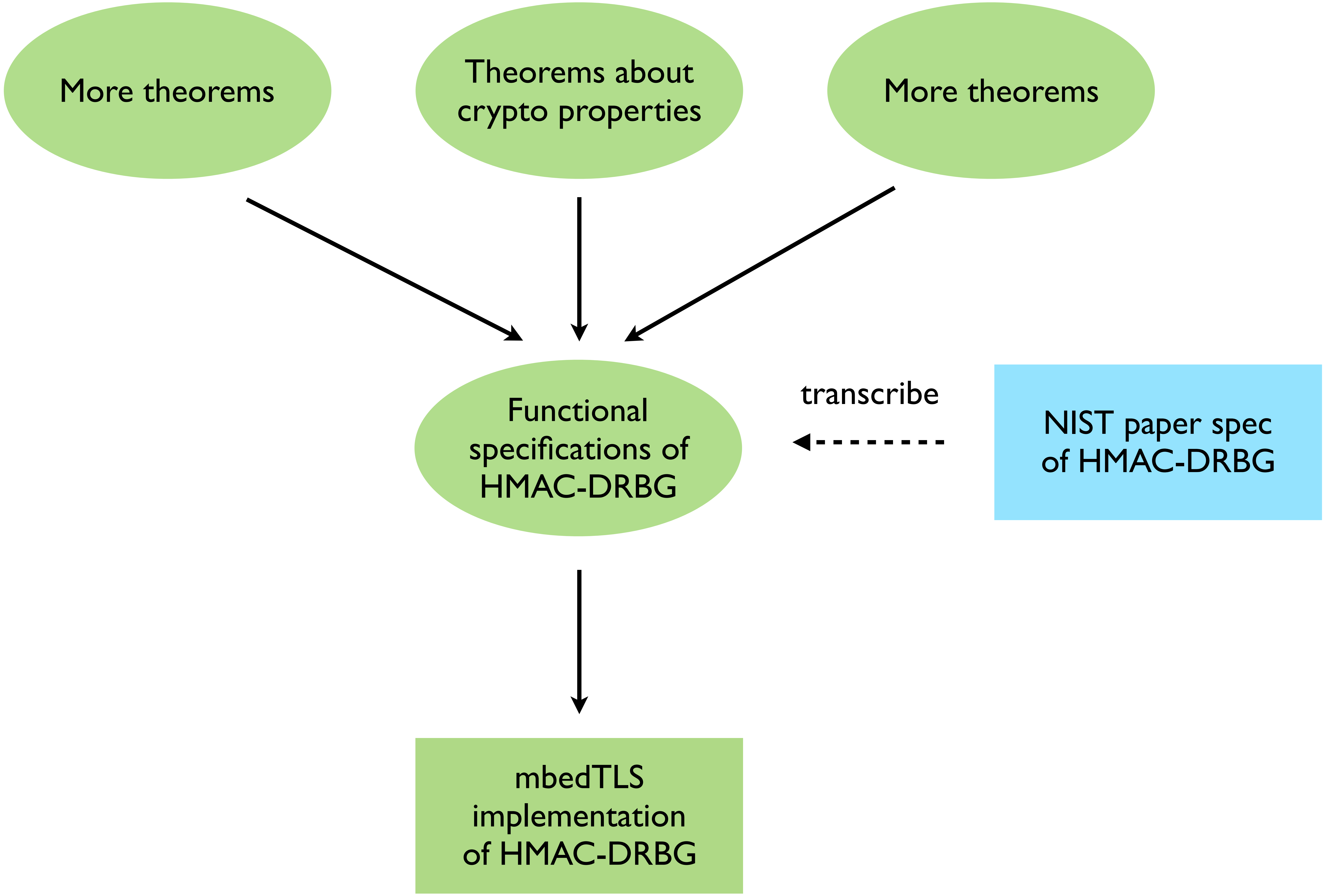
Our work


Theorems about crypto properties
security!
Functional specifications of HMAC-DRBG
transcribe
NIST paper spec of HMAC-DRBG
correctness!
mbedTLS HMAC-DRBG
CompCert verified compiler
Verified assembly

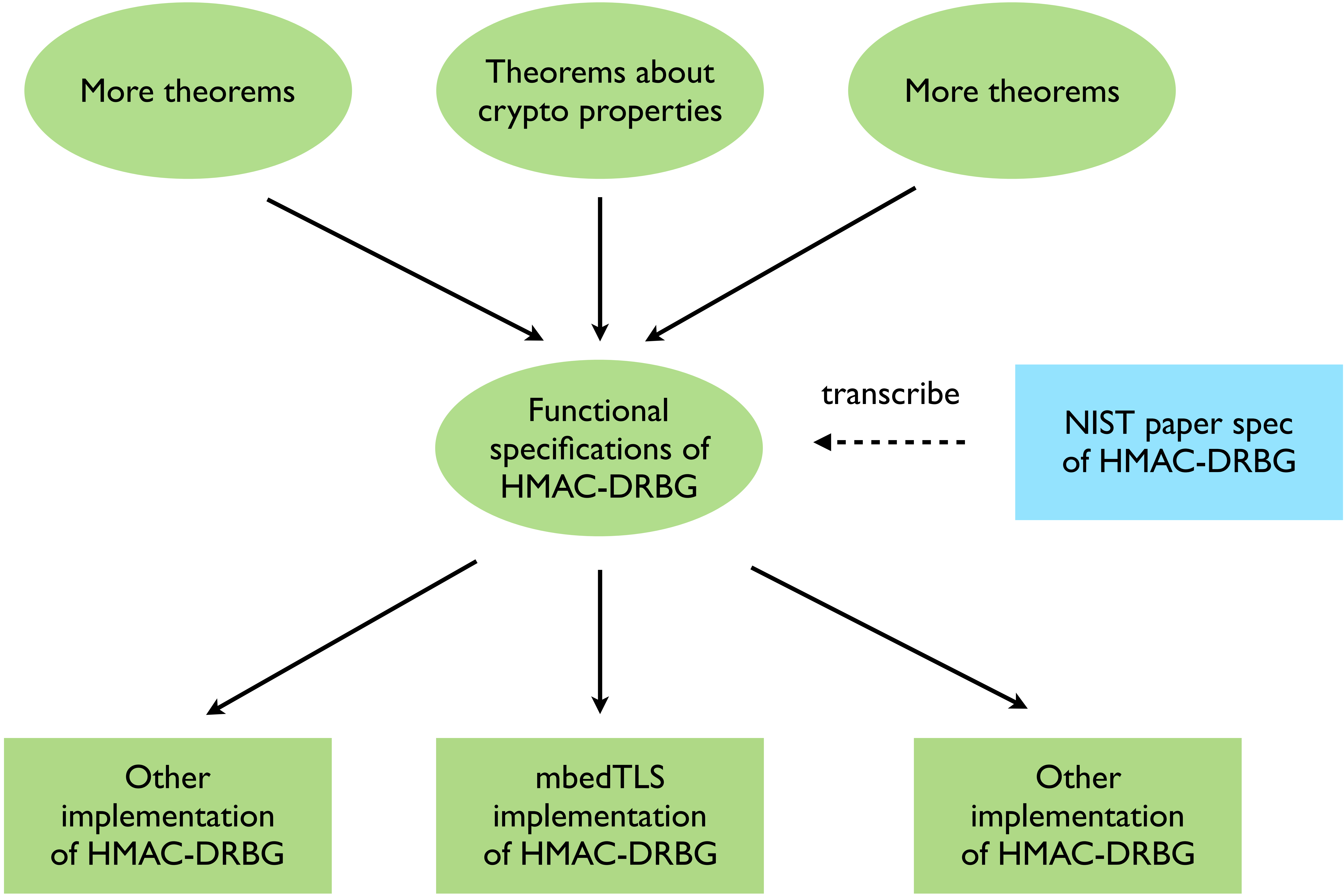
Our work


x → y:
x implements y
Theorems about crypto properties
Modular proofs!
Functional specifications of HMAC-DRBG
transcribe
NIST paper spec of HMAC-DRBG
mbedTLS implementation of HMAC-DRBG

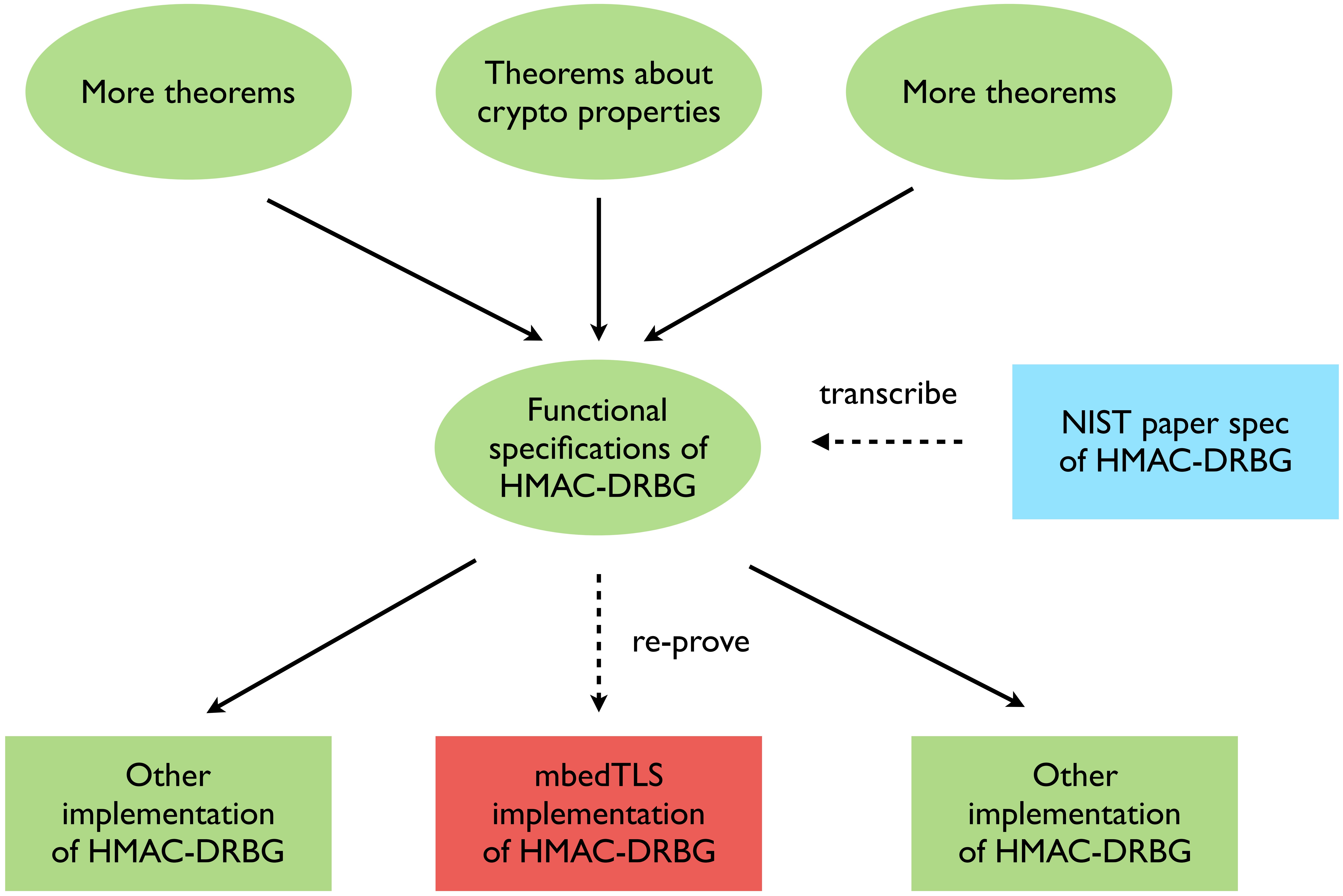
Modular proofs



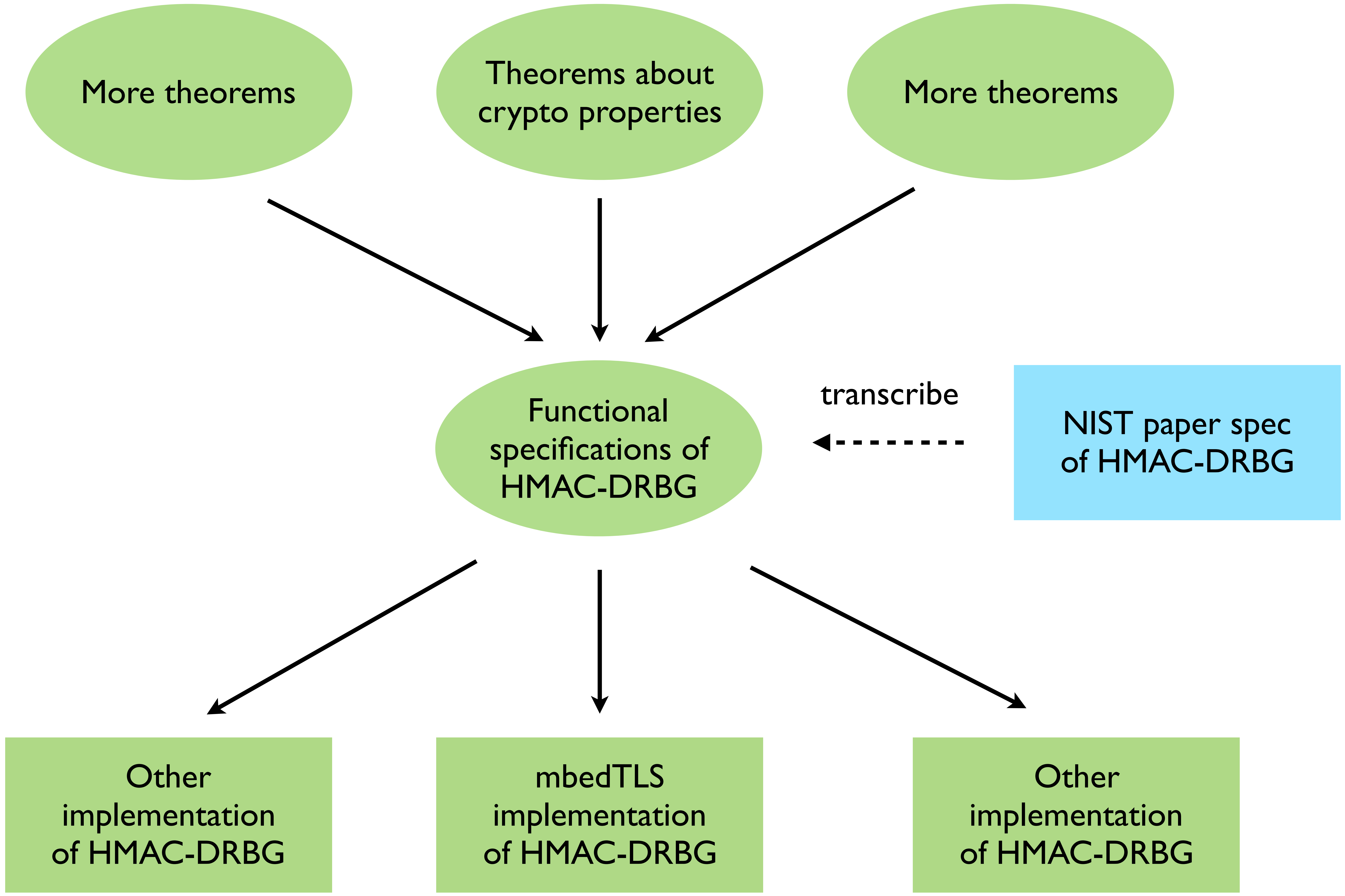
Modular proofs



Modular proofs



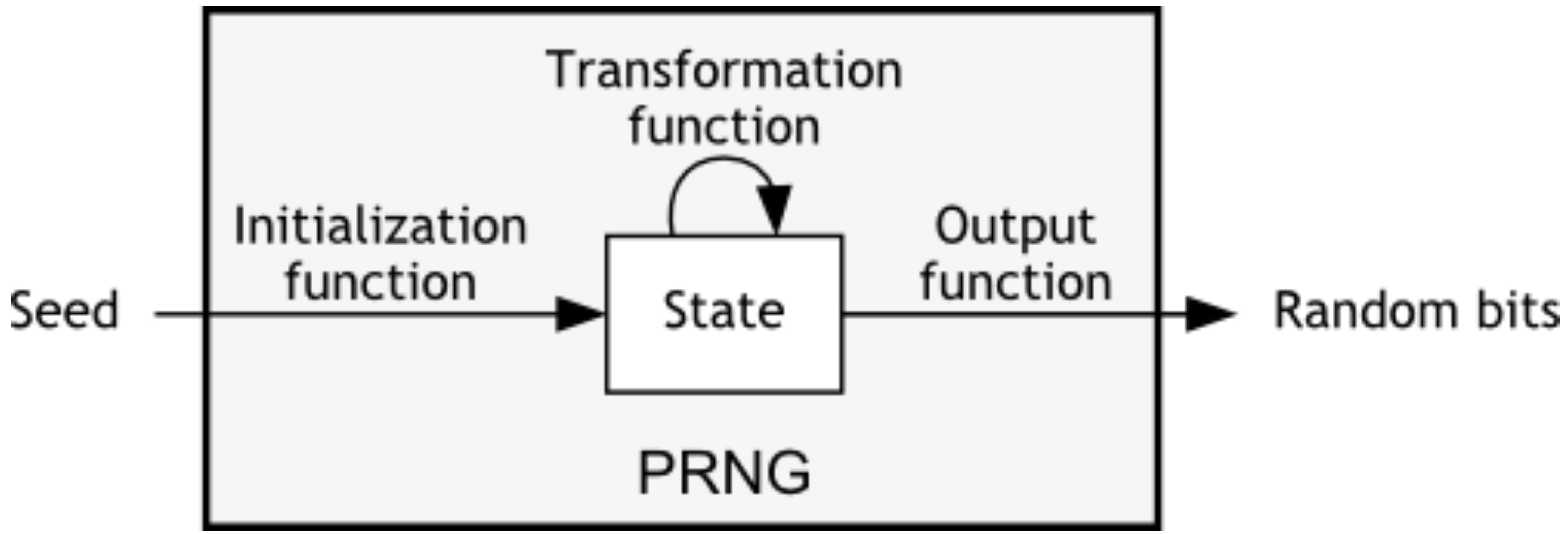
Modular proofs



HMAC–DRBG

(keyed–hash message authentication code
deterministic random bit generator)

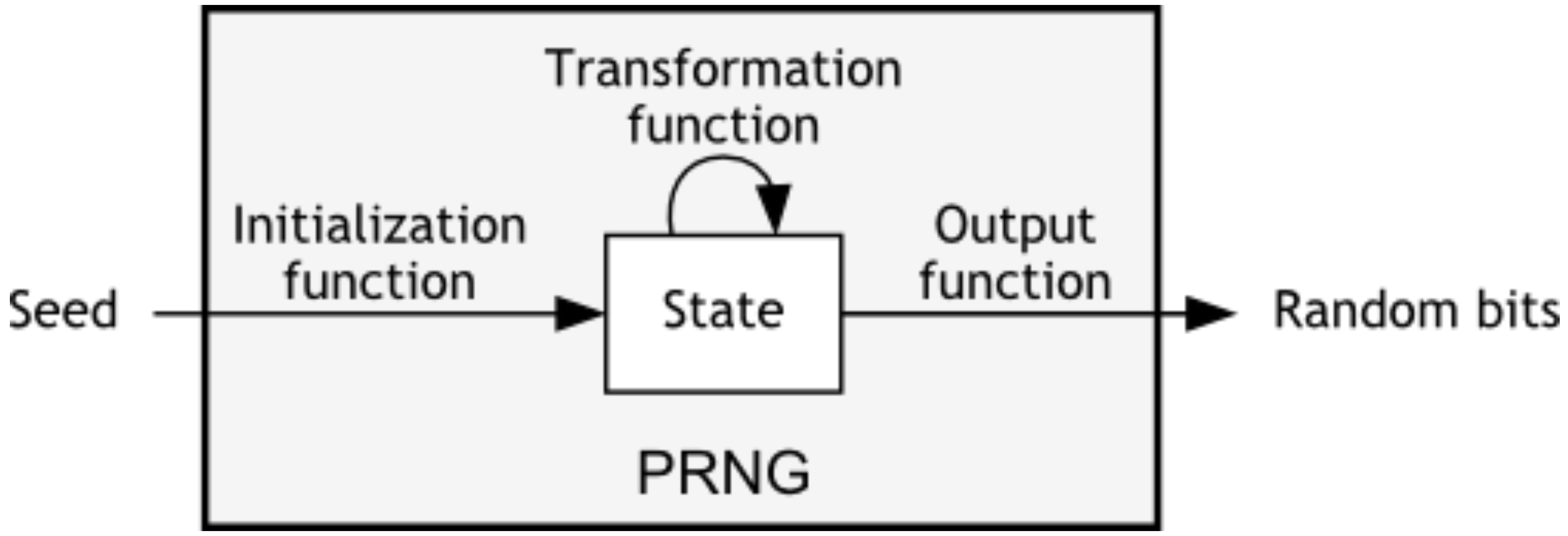
Generic pseudorandom number generator



(PRG is synonymous with PRNG)

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

Generic pseudorandom 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/

Typical PRG use

```
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,
...
```

Generate

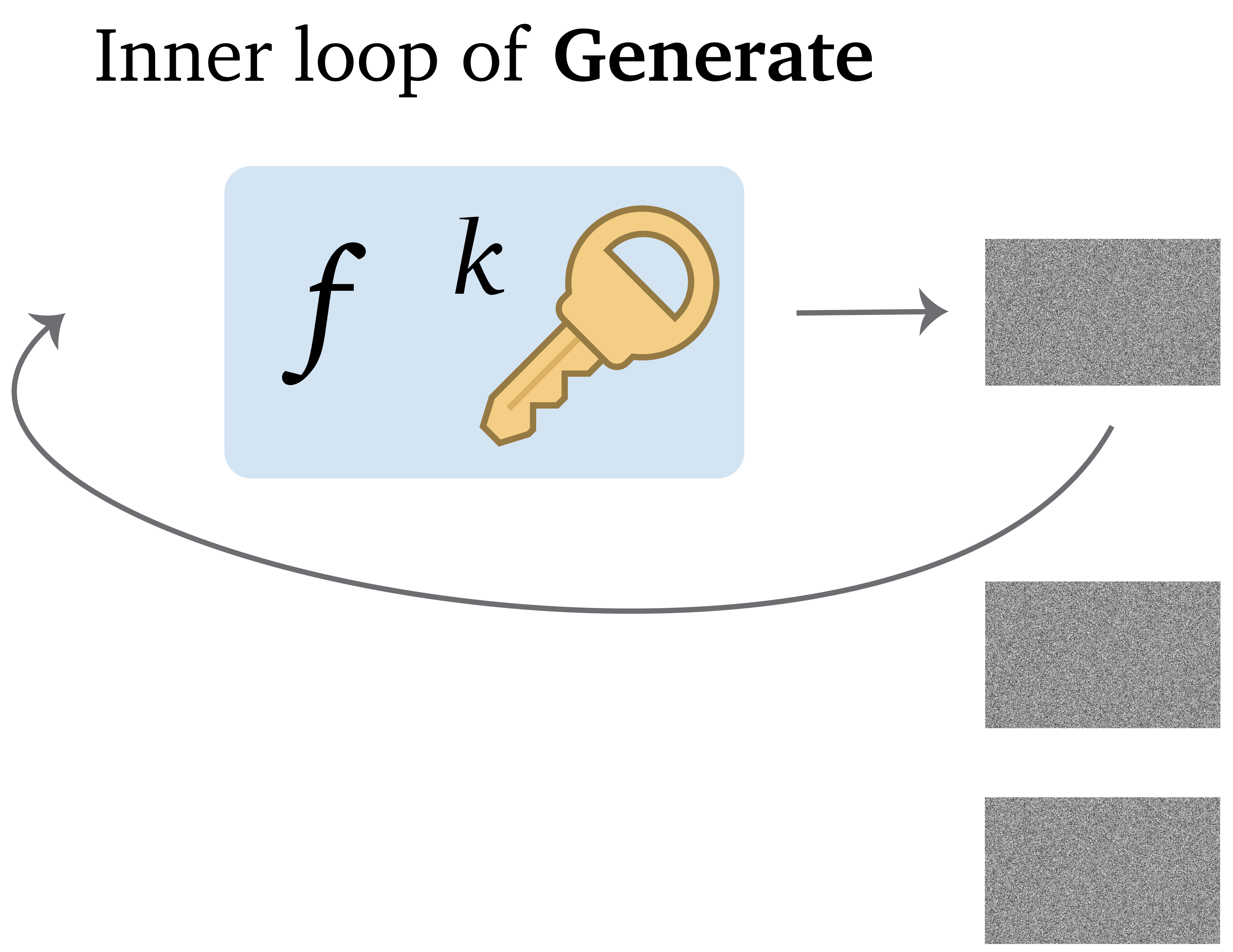
k = secret key; v = initialization vector (internal state)

f = hash function (e.g. HMAC)

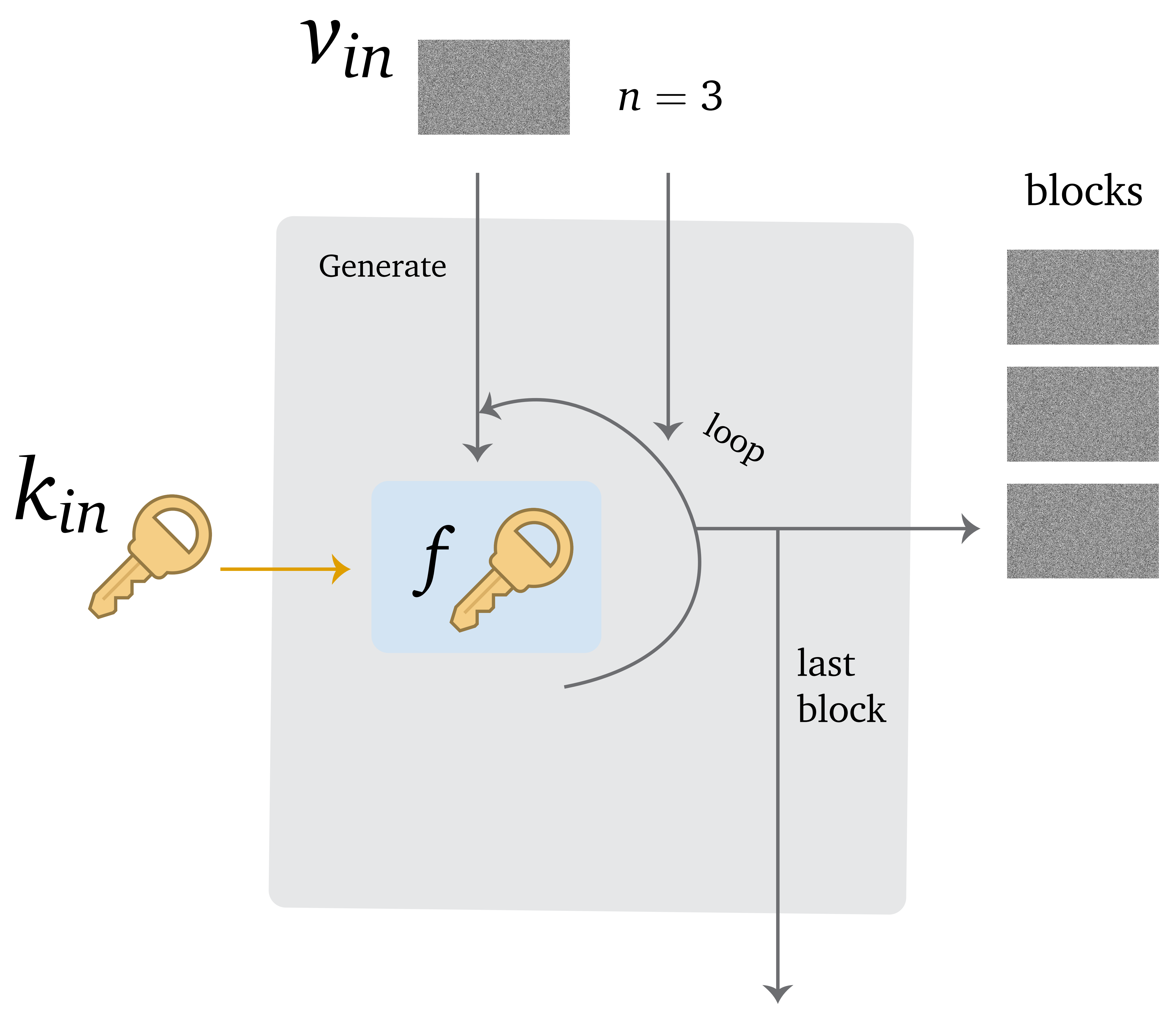
```
rand_bits =
f(k, v)
|| f(k, f(k, v))
|| f(k, (f(k, f(k, v)))...
```

outputs used again as inputs

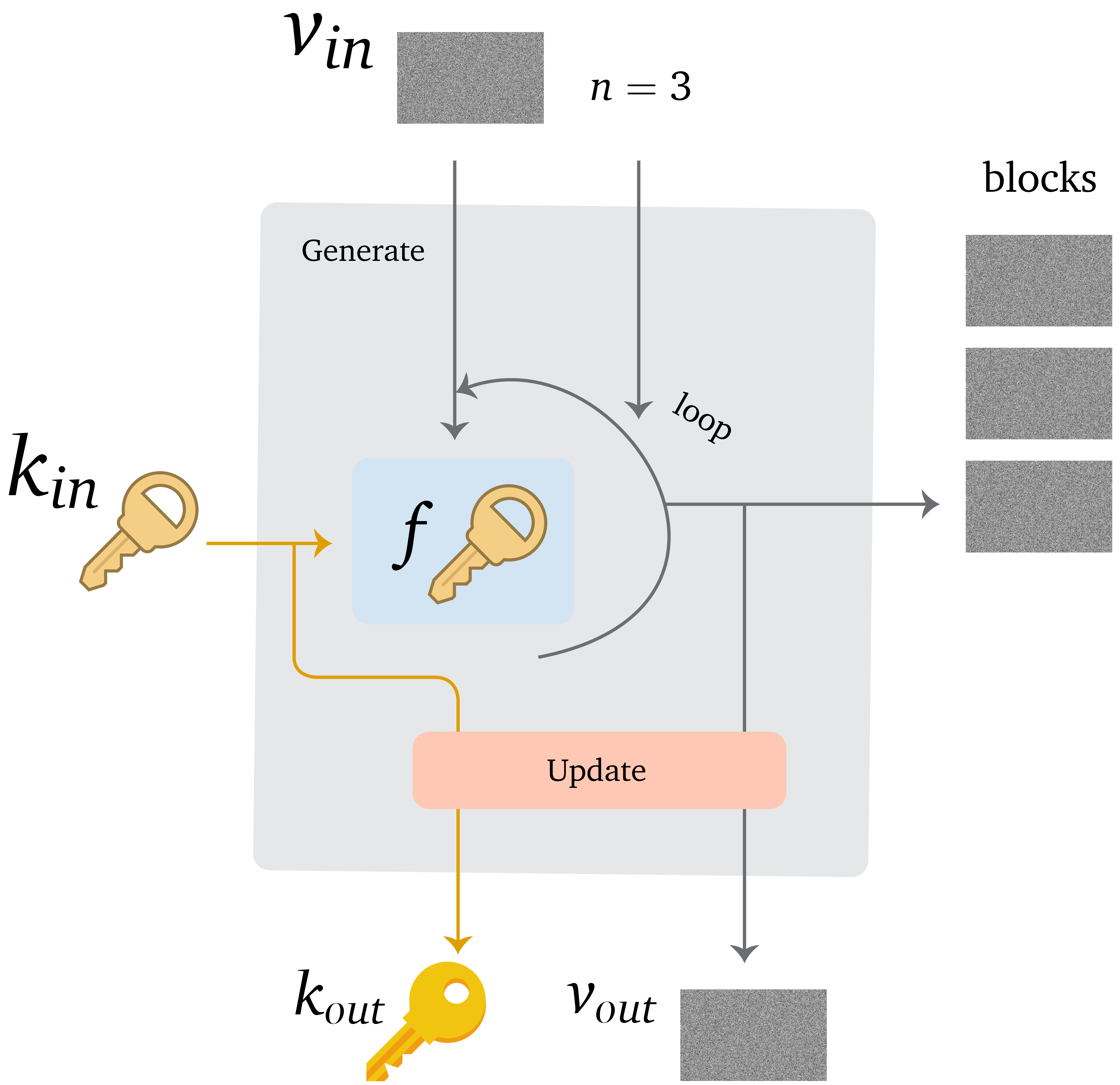
The outputs are used as inputs



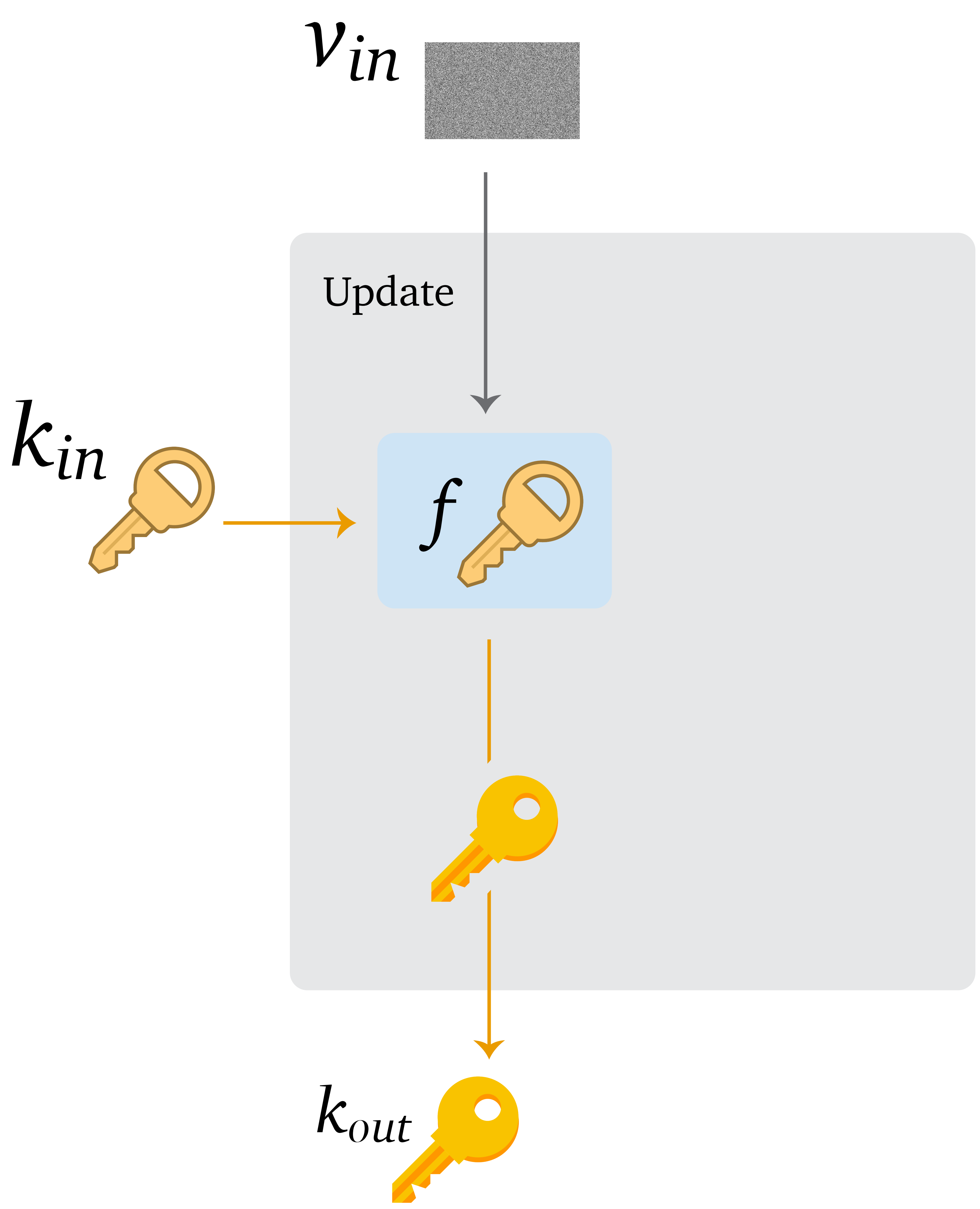
Generating bits



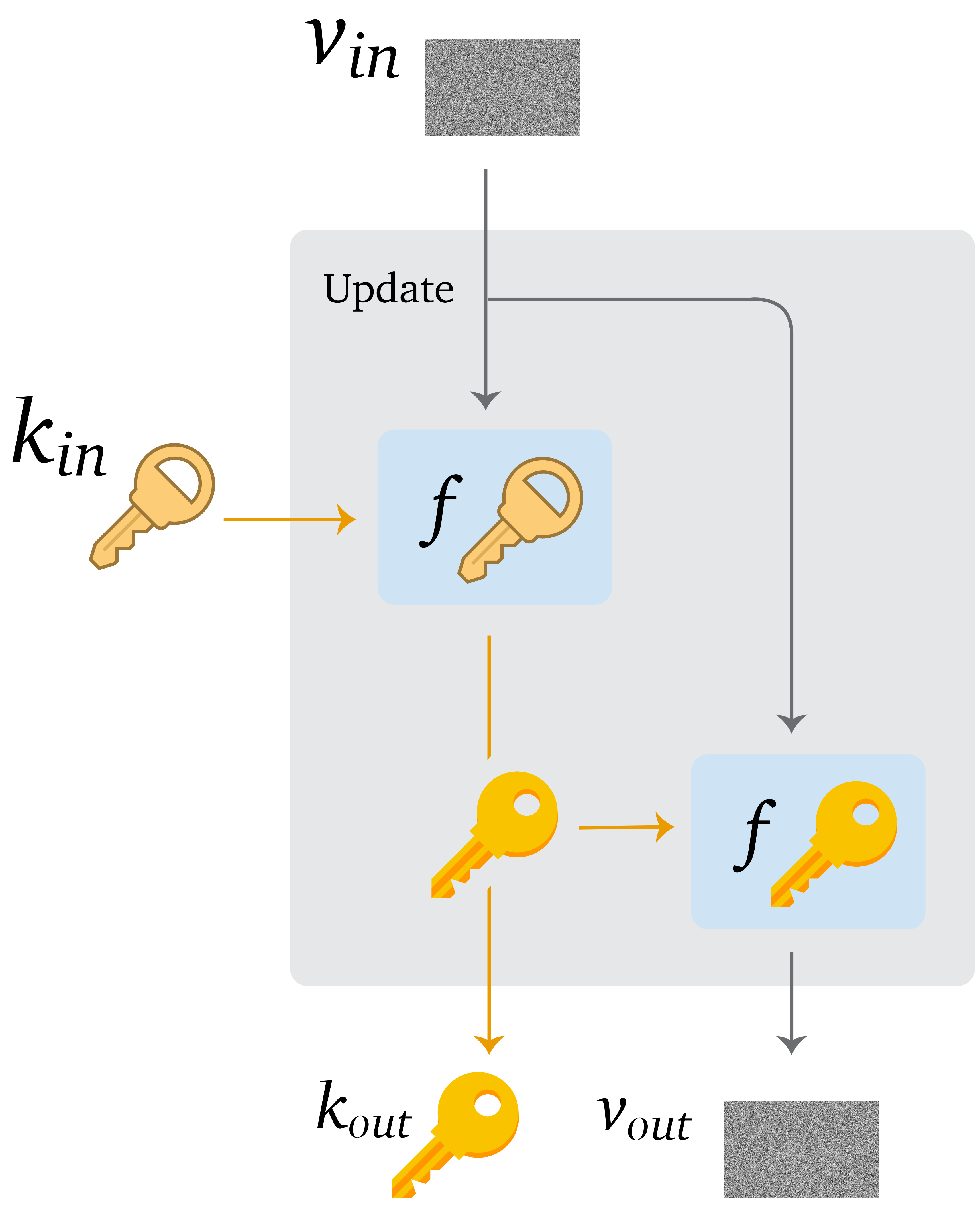
Generating bits



Updating the internal state

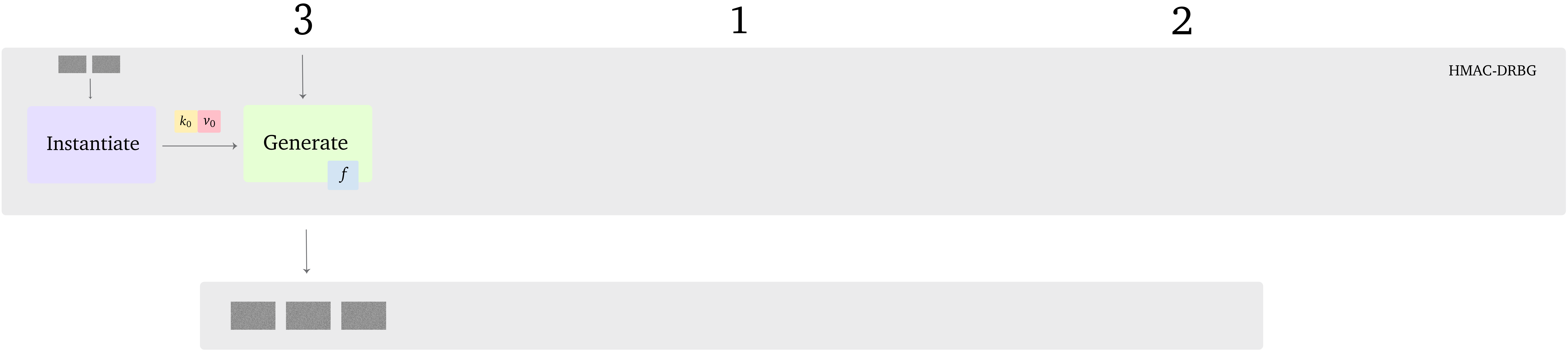


Updating the internal state

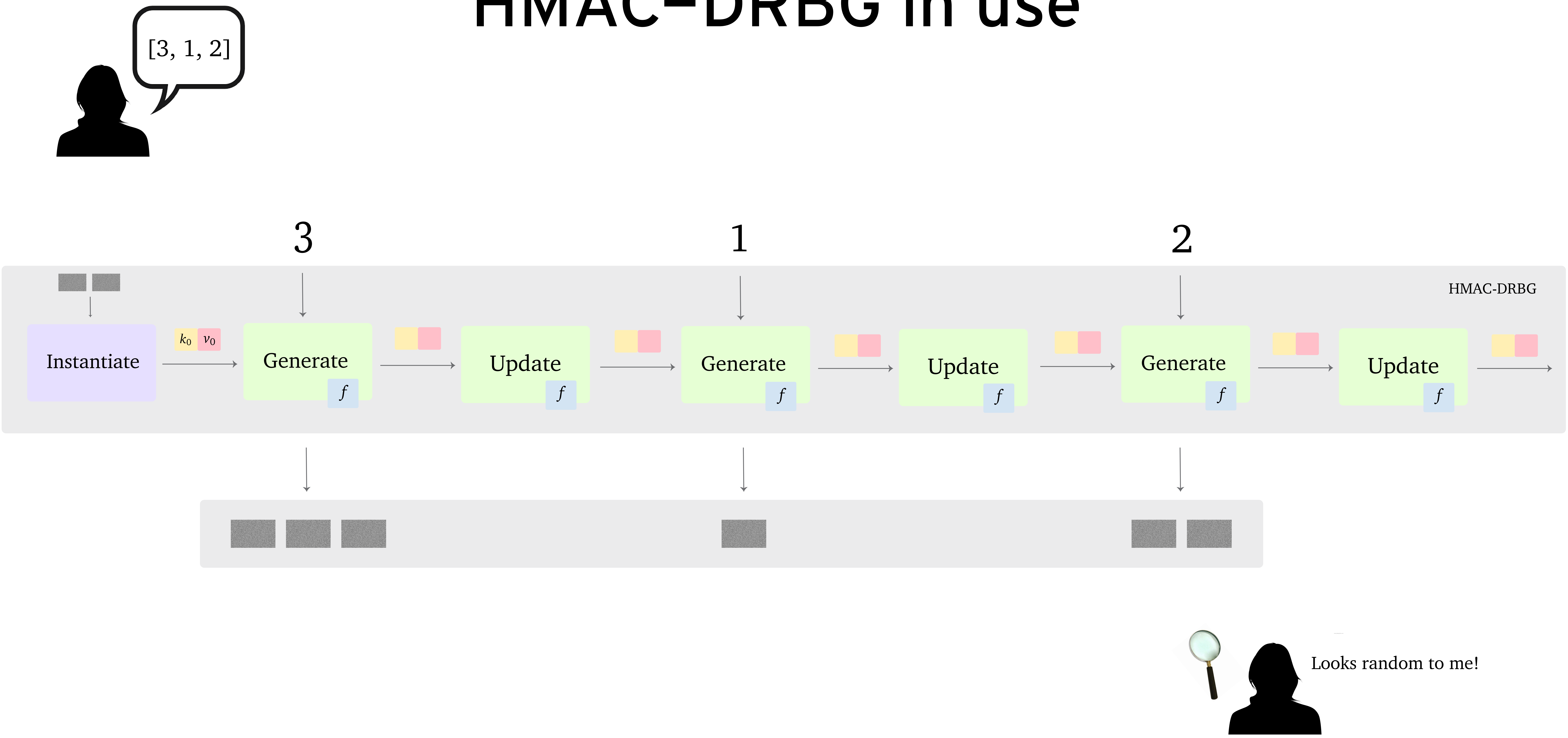


HMAC-DRBG in use

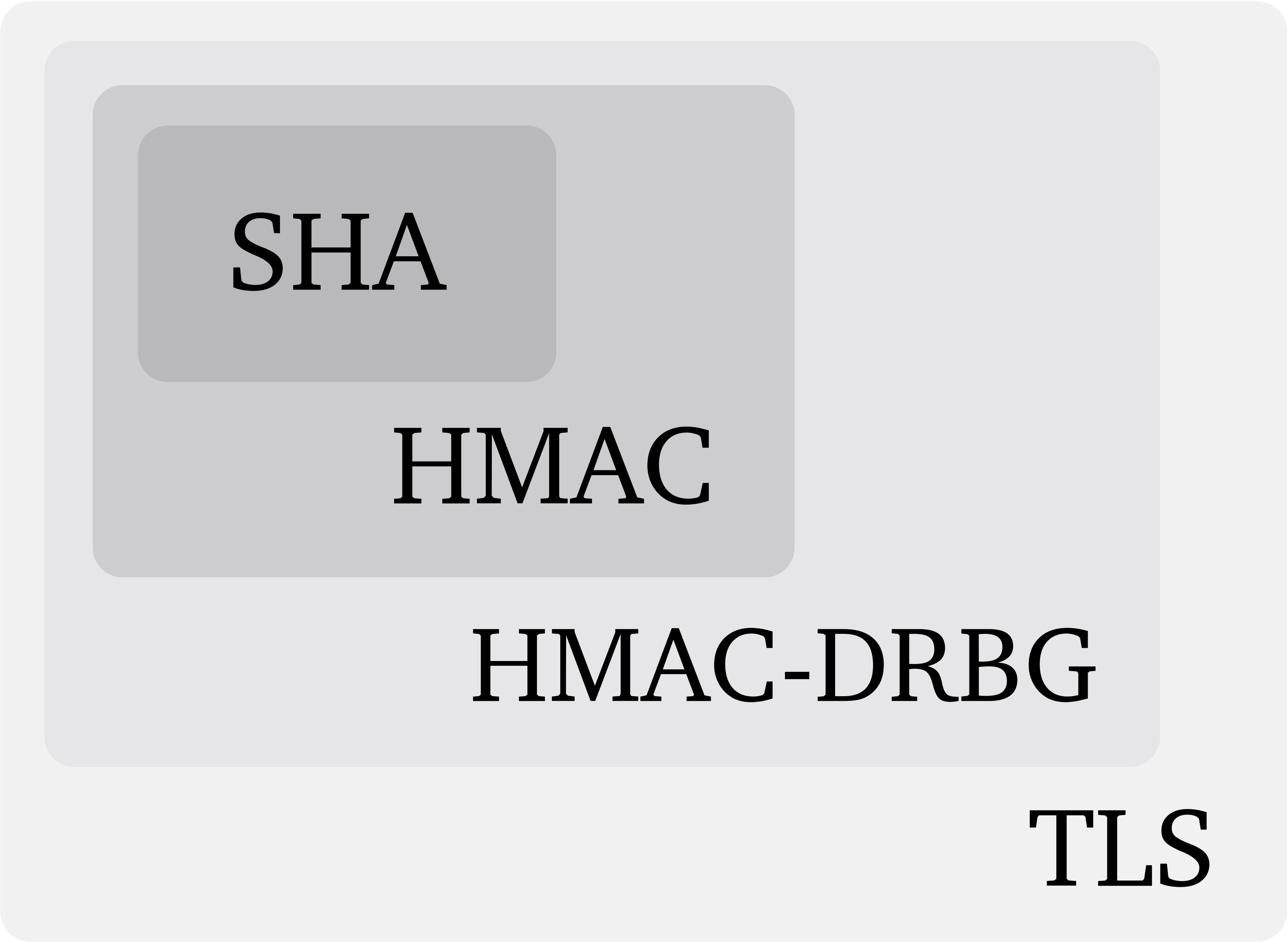
HMAC-DRBG in use



HMAC–DRBG in use



Our proof of indistinguishability from random (HMAC-DRBG security)



Previously, we built a machine–checked proof that HMAC is a PRF, subject to the usual assumptions

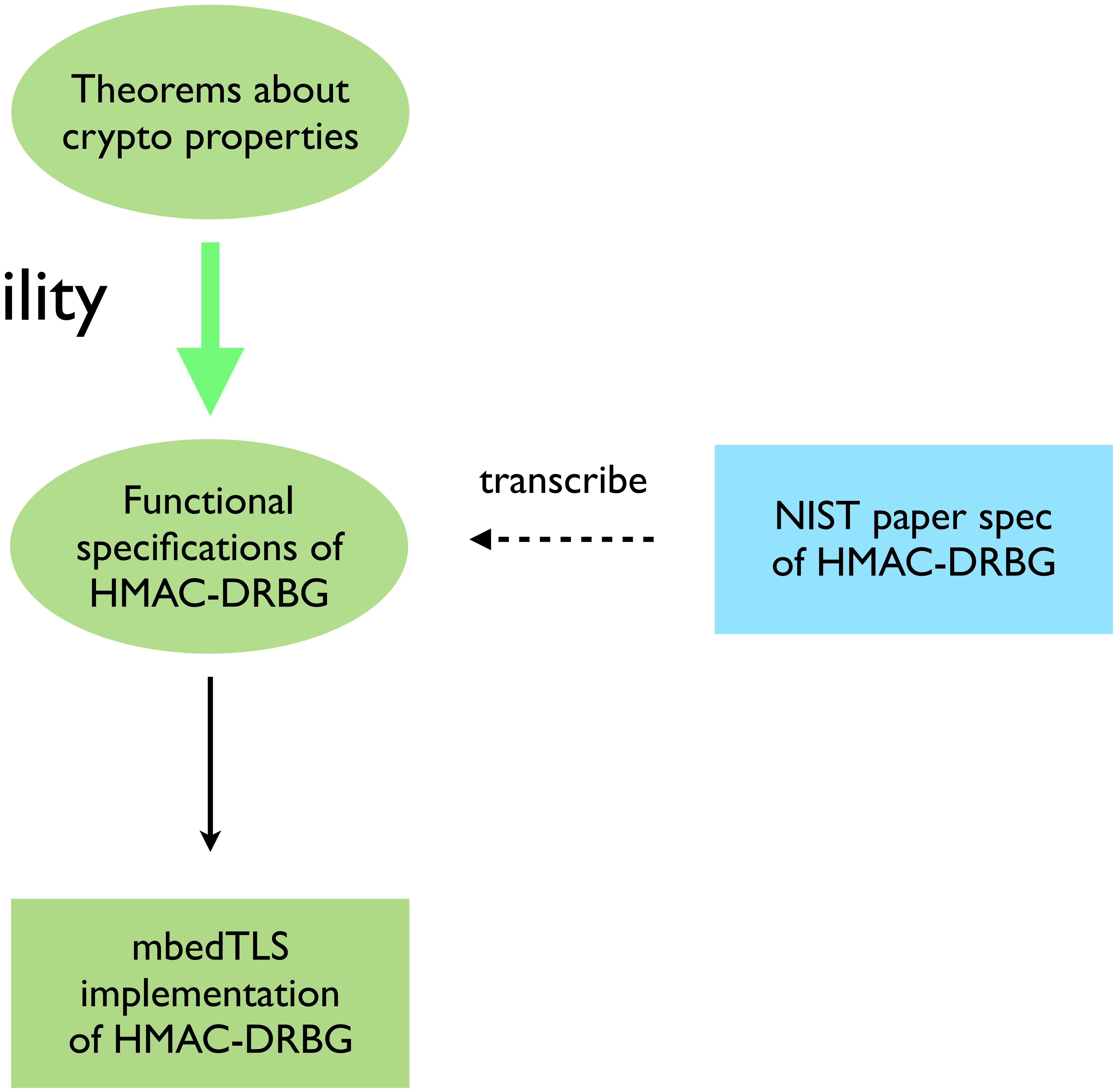
Verified correctness and security of OpenSSL HMAC, Beringer et al, USENIX Security '15

Our work

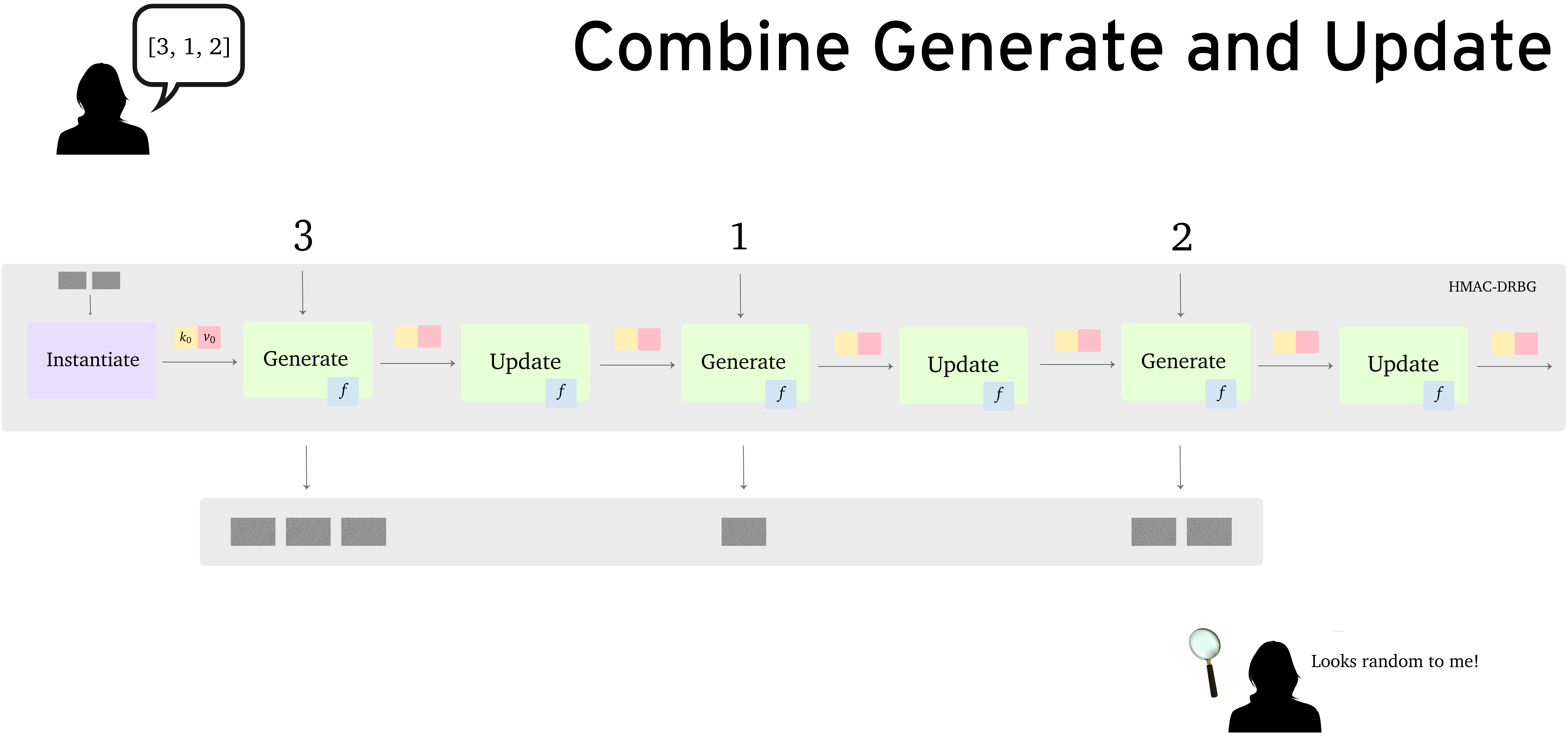
x → y:
x implements y

proof of indistinguishability

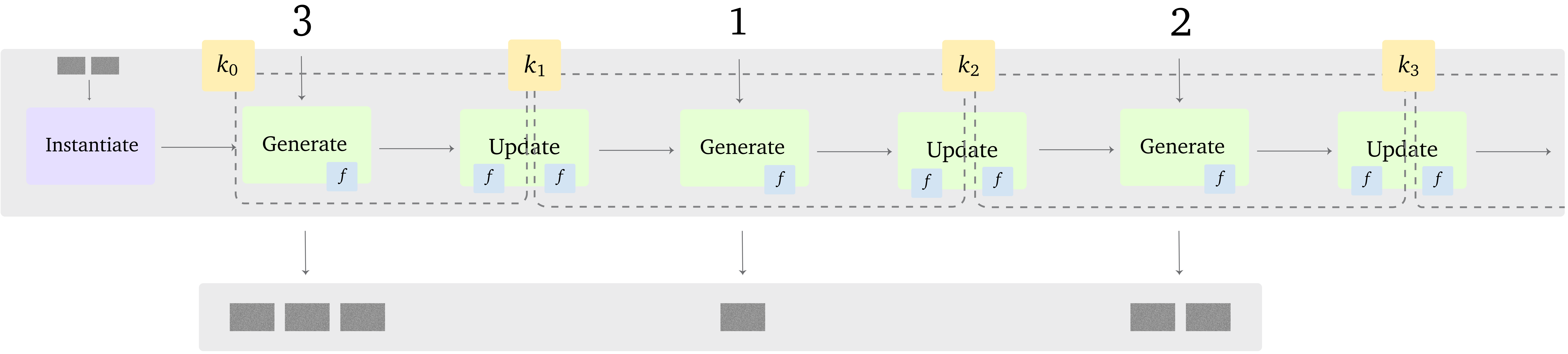
Structurally similar to
Security Analysis of DRBG Using HMAC in NIST SP 800-90,
Hirose (2008)
(though done independently)



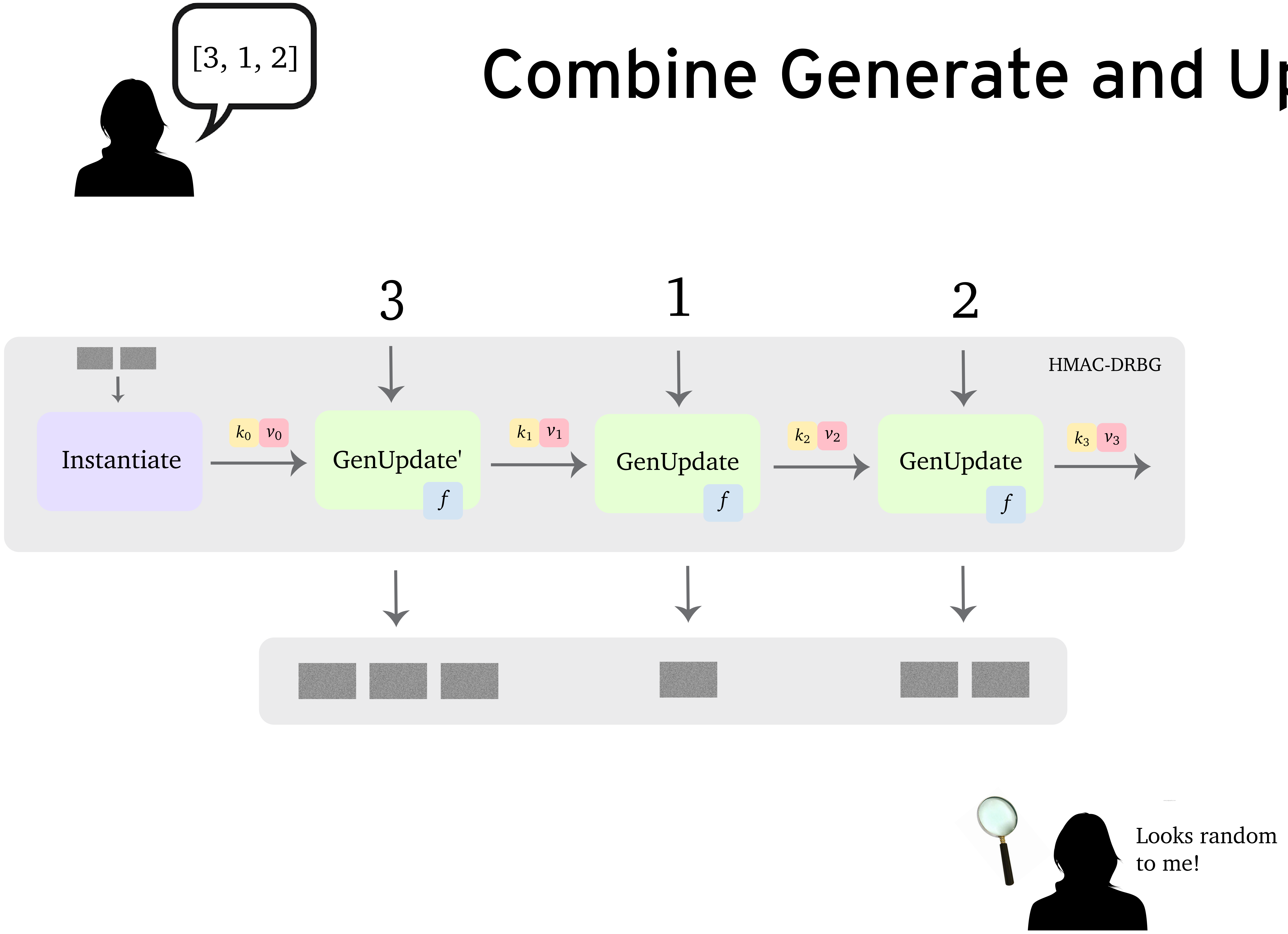
Combine Generate and Update



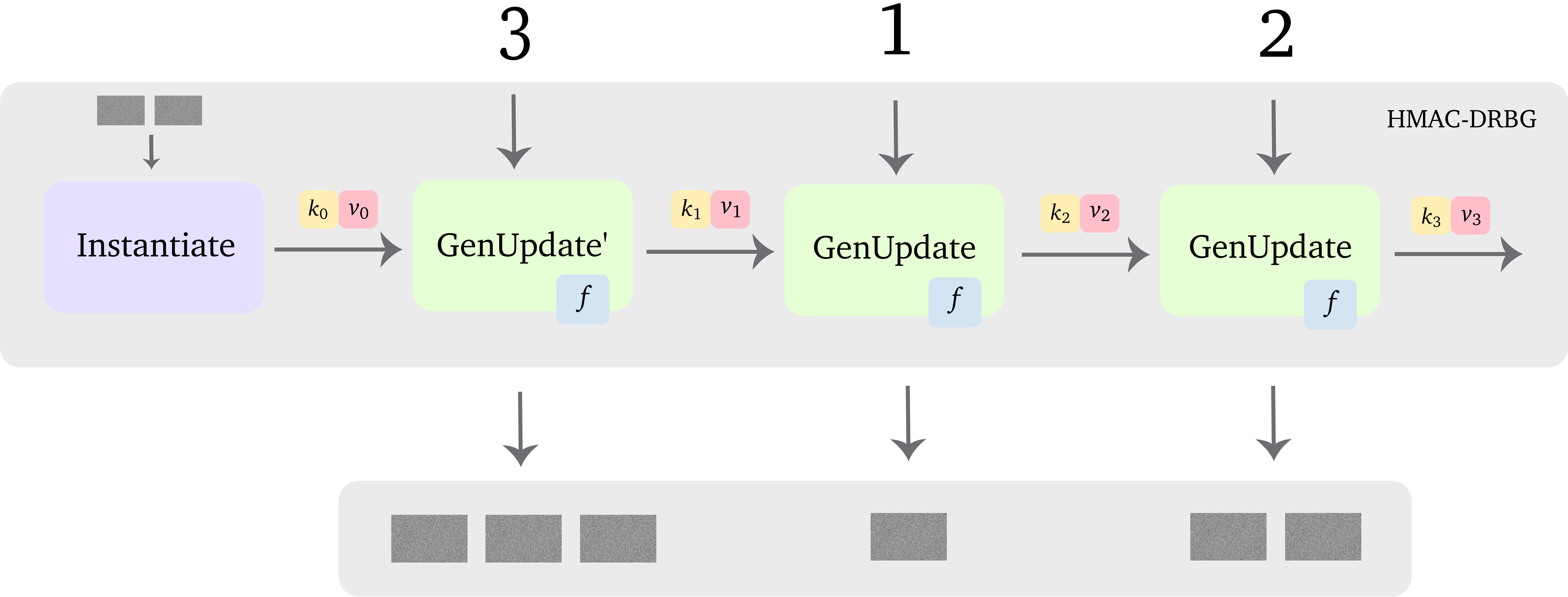
Combine Generate and Update



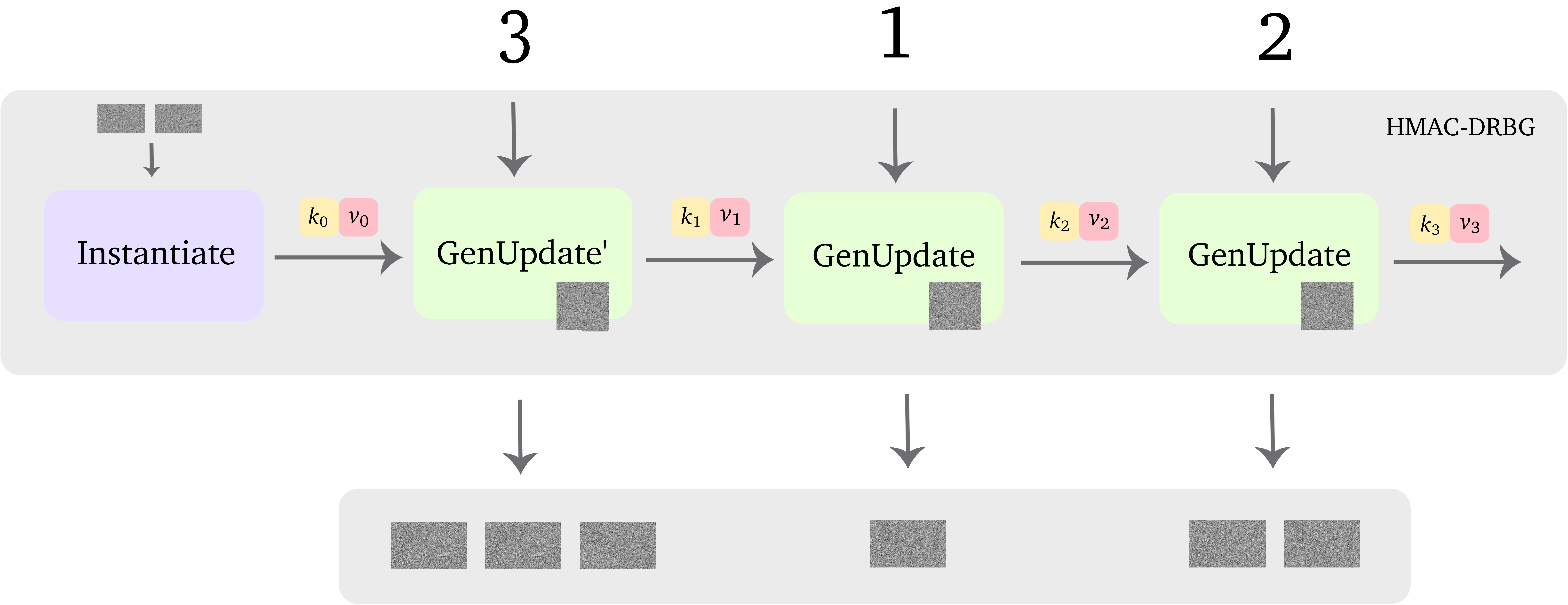
Combine Generate and Update



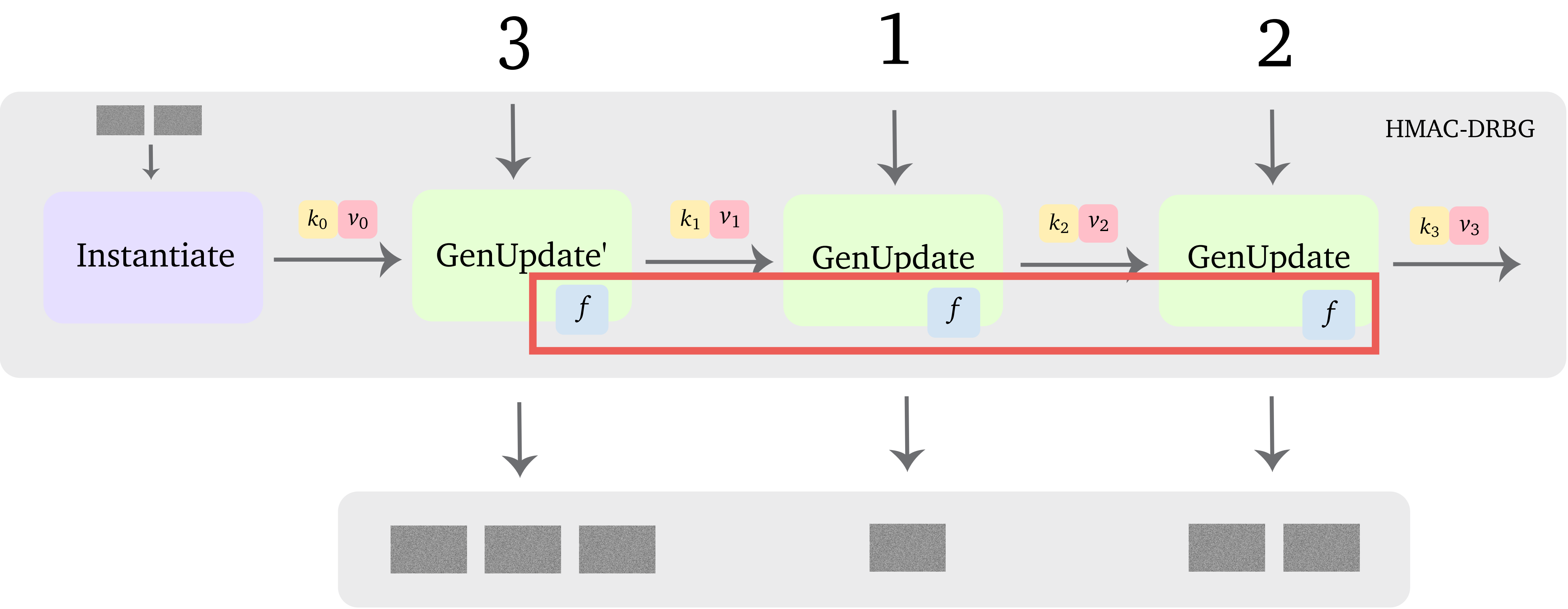
Real–world and ideal–world hybrids

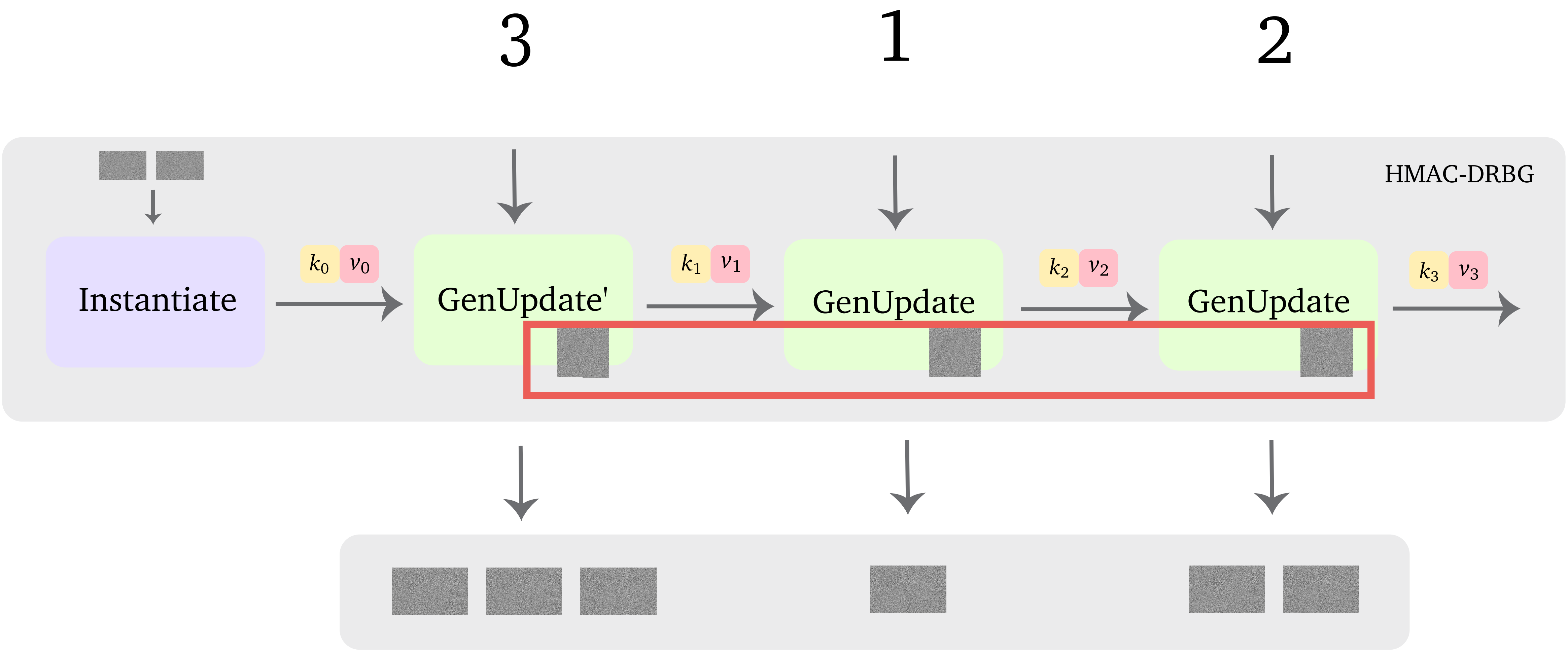


Real–world and <u>ideal–world</u> hybrids



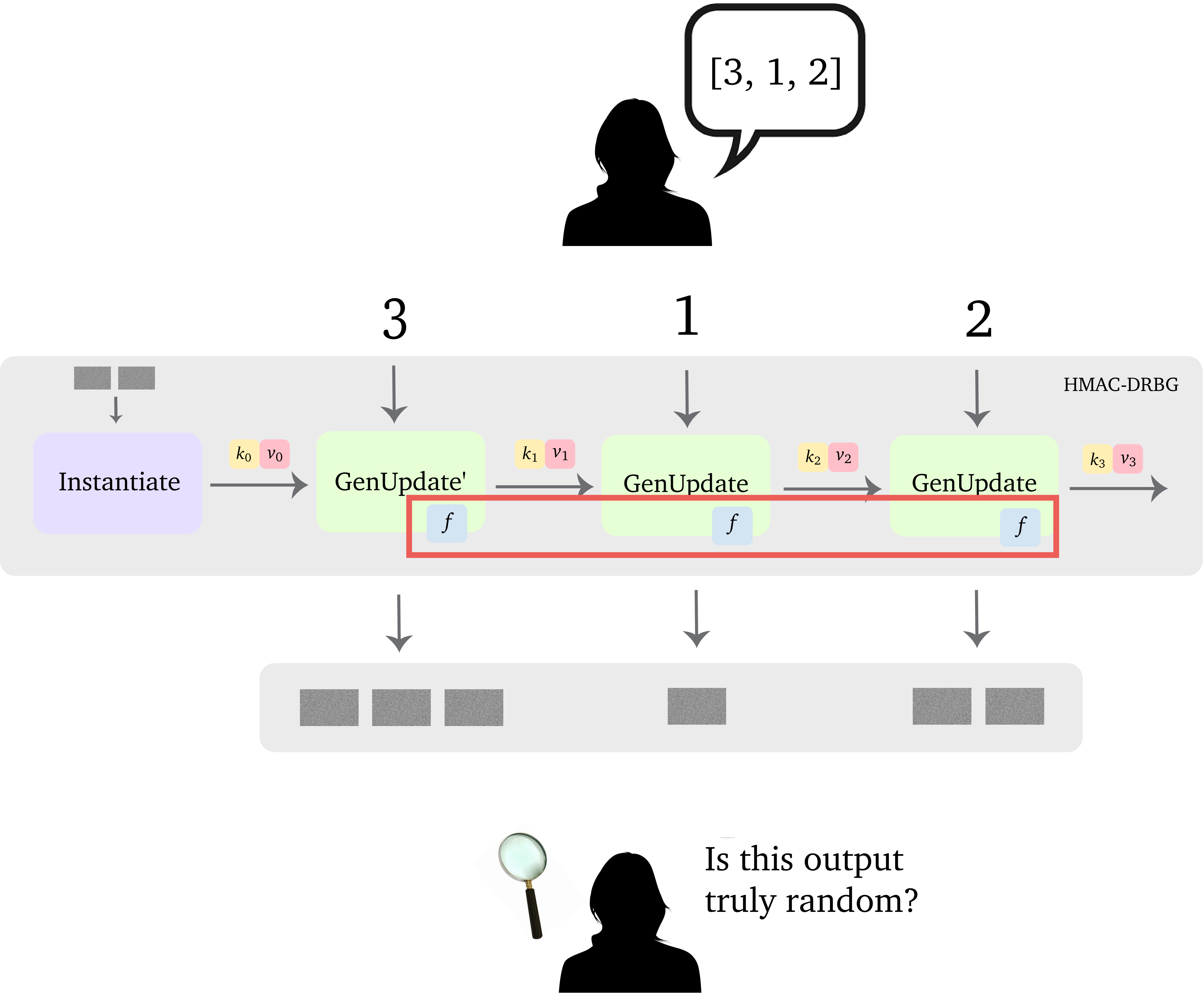
Real–world and ideal–world games

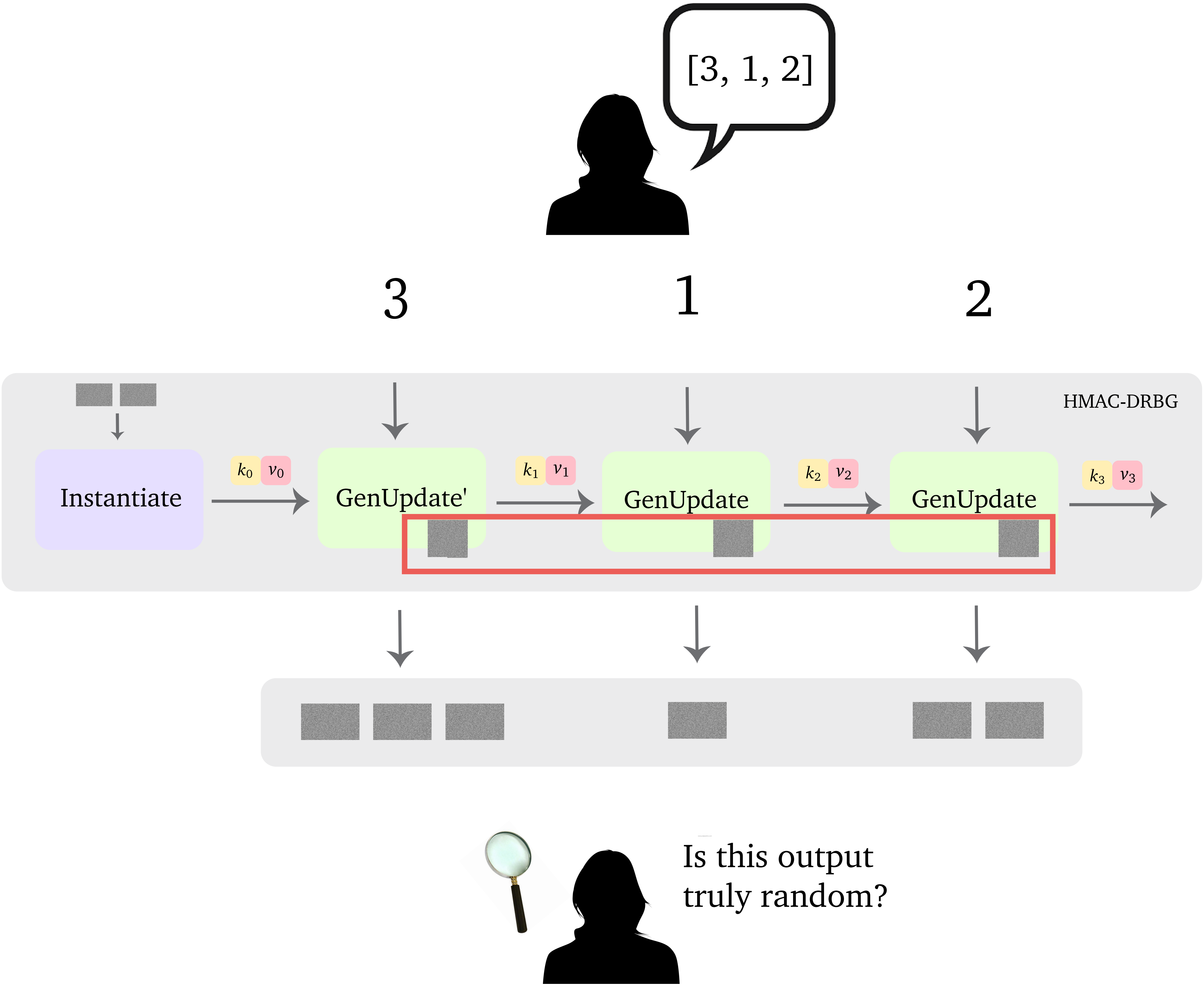




Real–world and ideal–world games

Prove that it's hard to tell the difference!

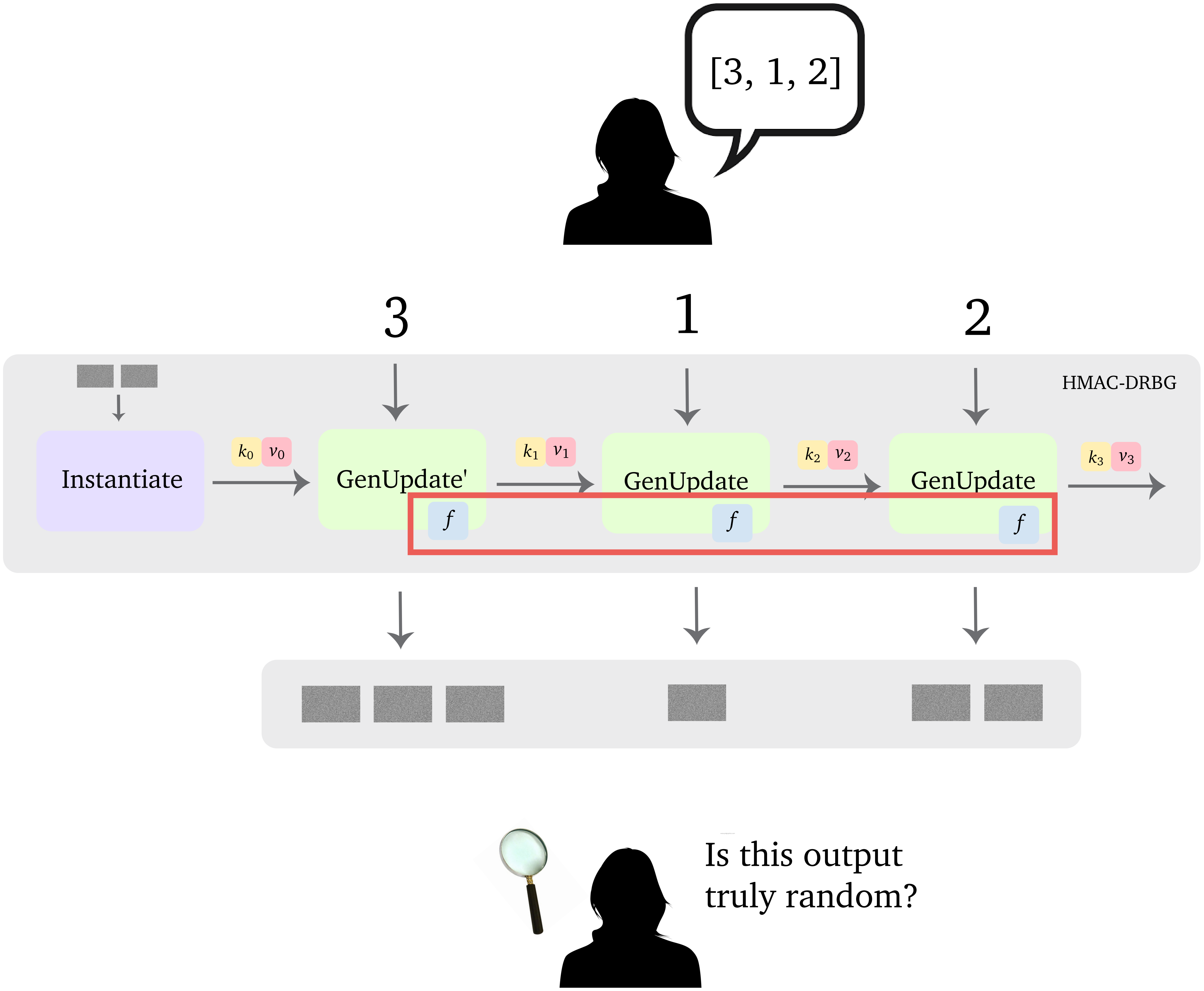


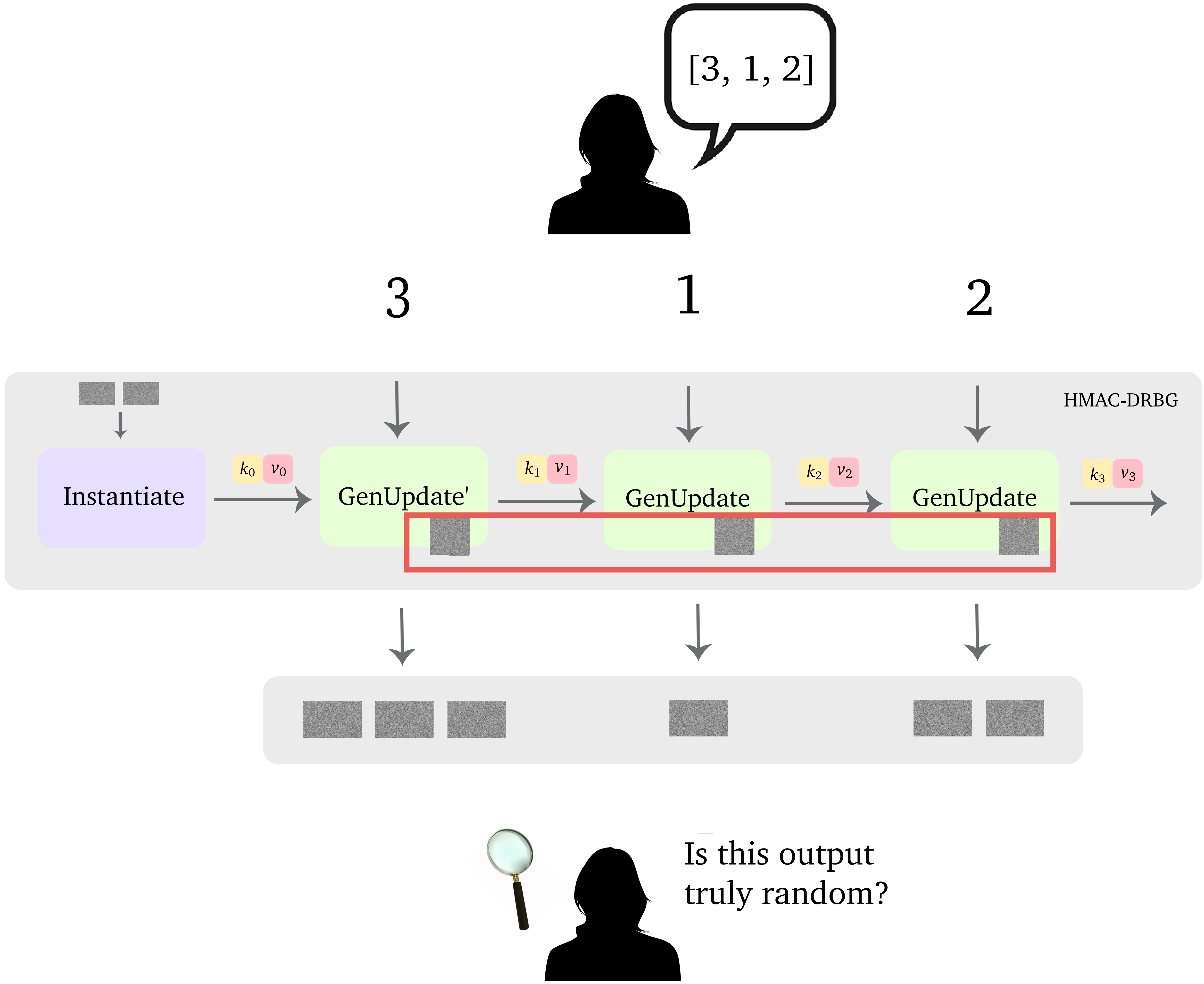


Bridging the real–world and ideal–world definitions with hybrids

Real–world and ideal–world games

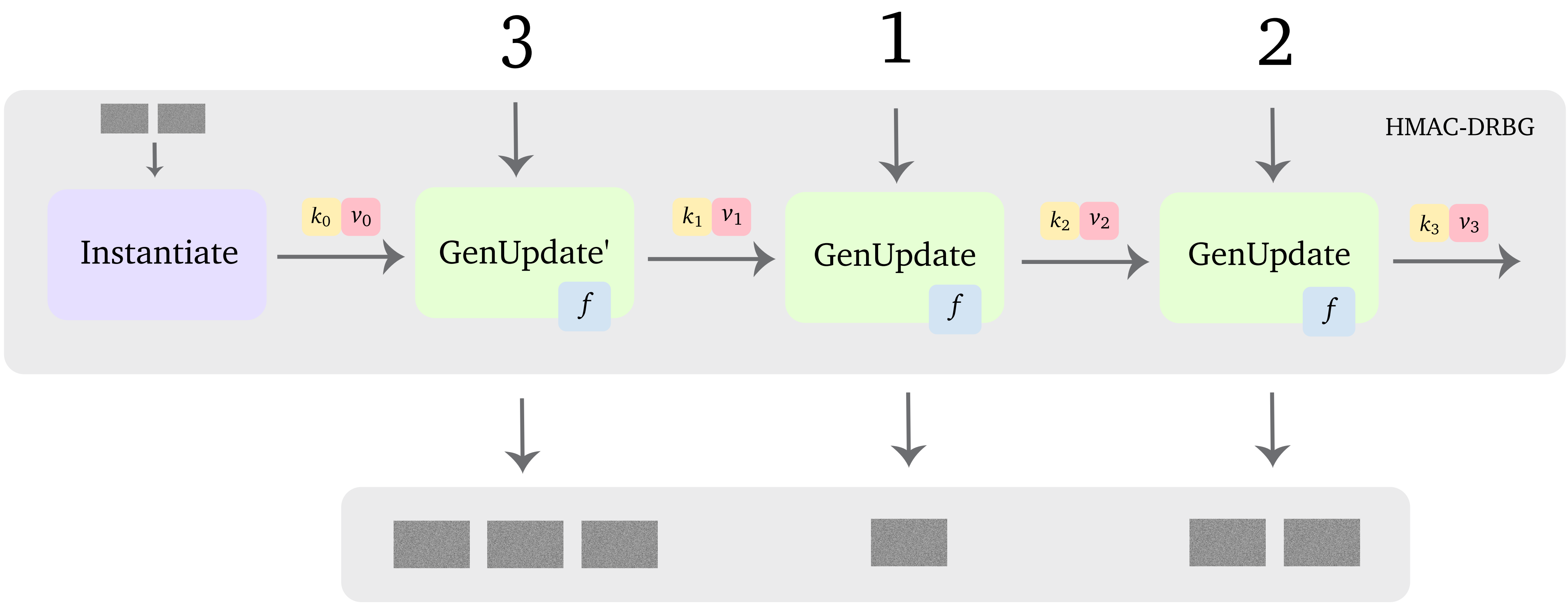
Prove that it's hard to tell the difference!



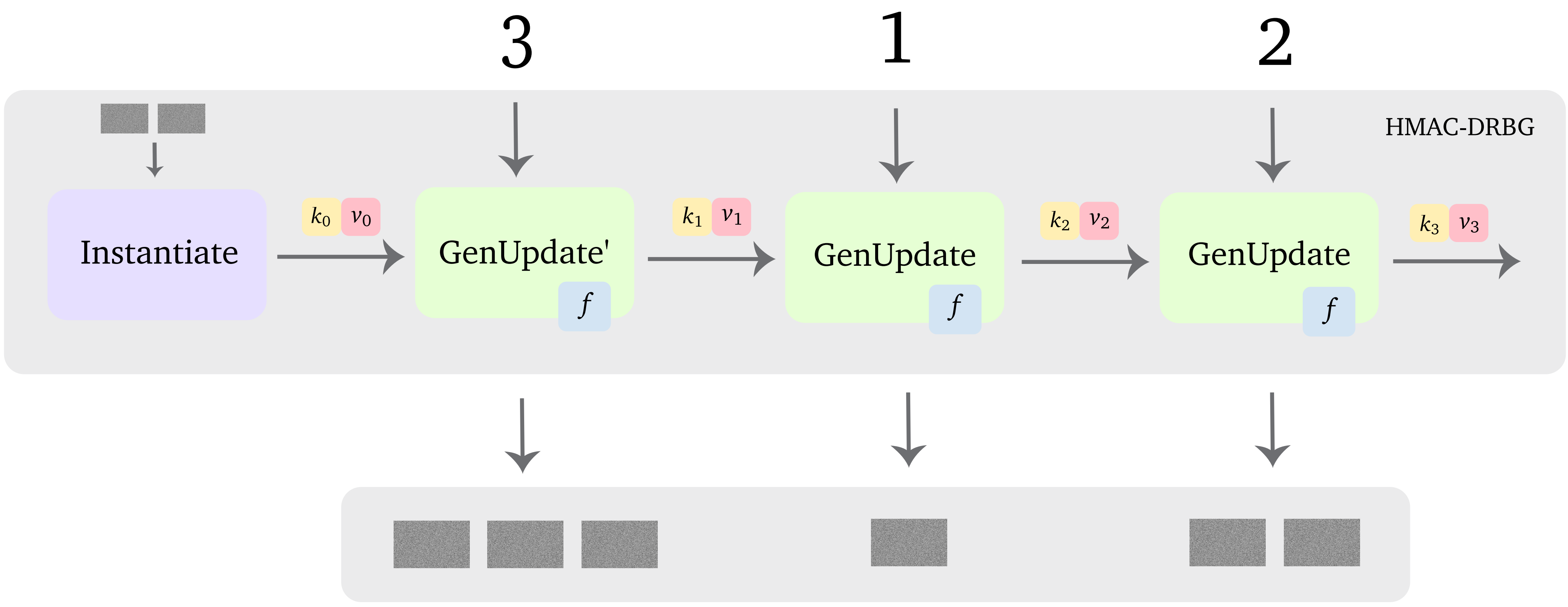


Hybrids

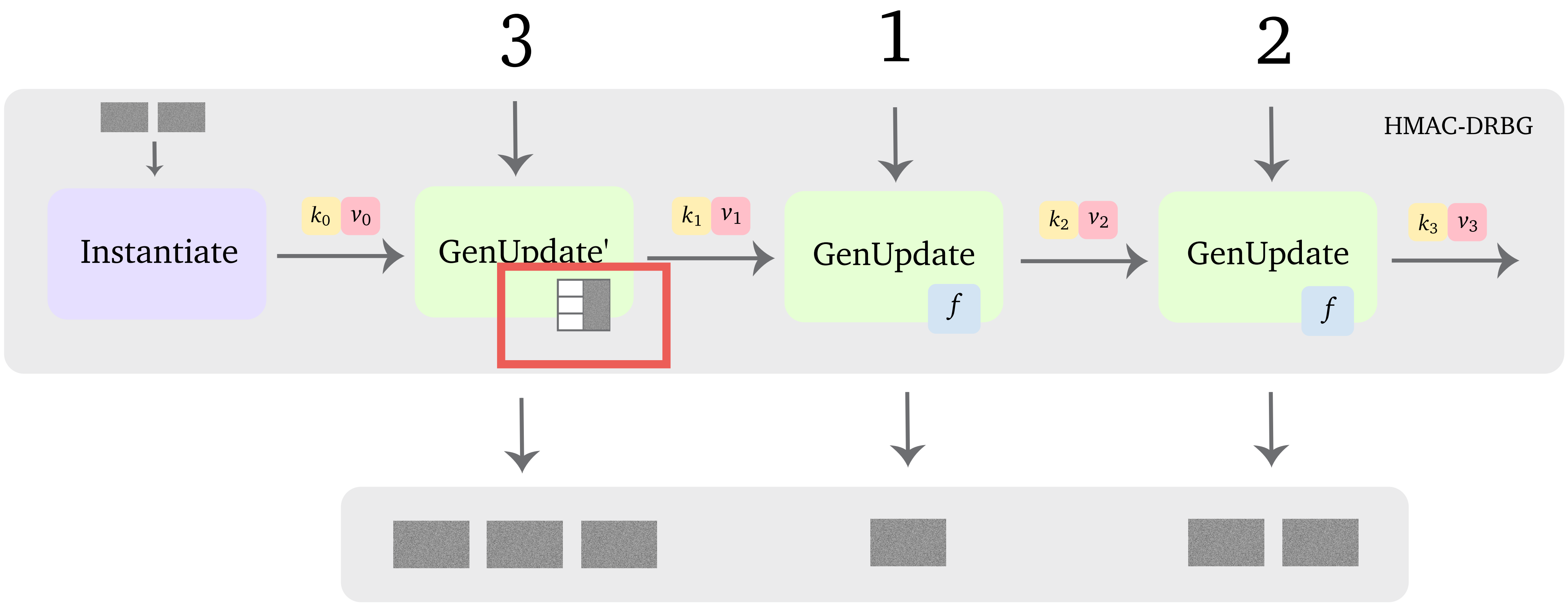
real–world hybrid



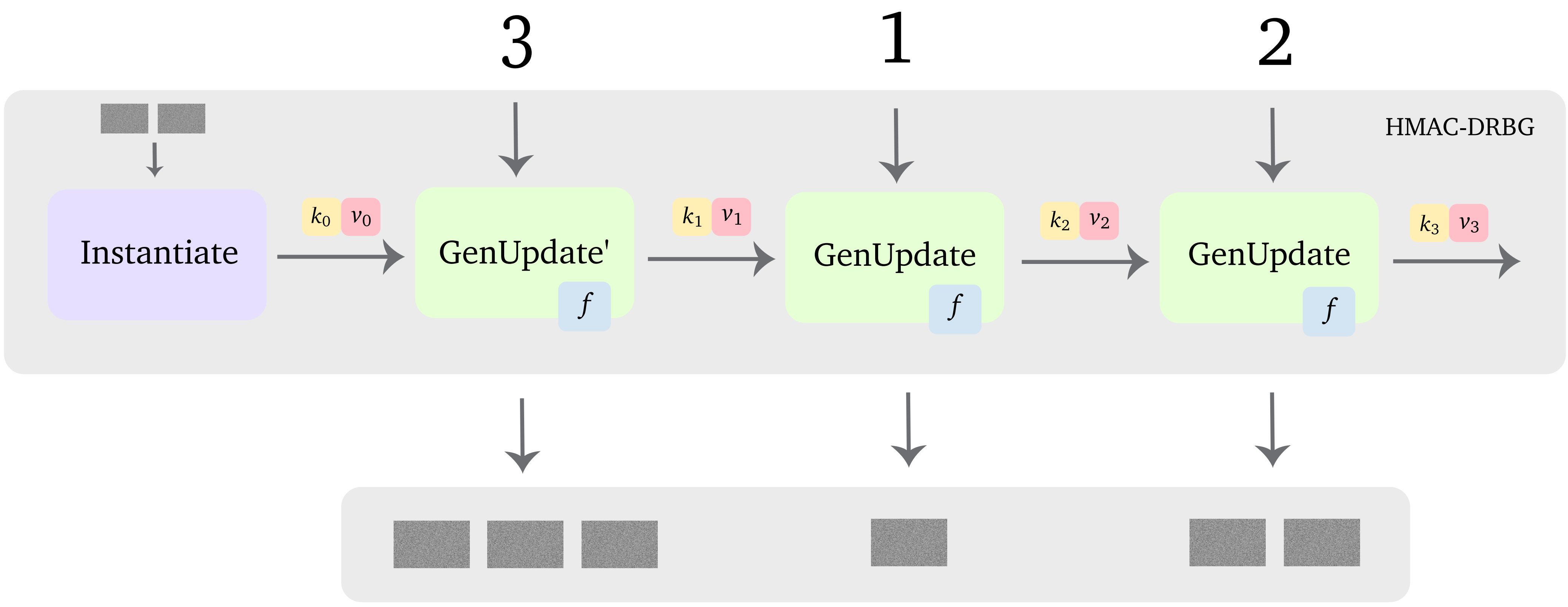
Hybrids

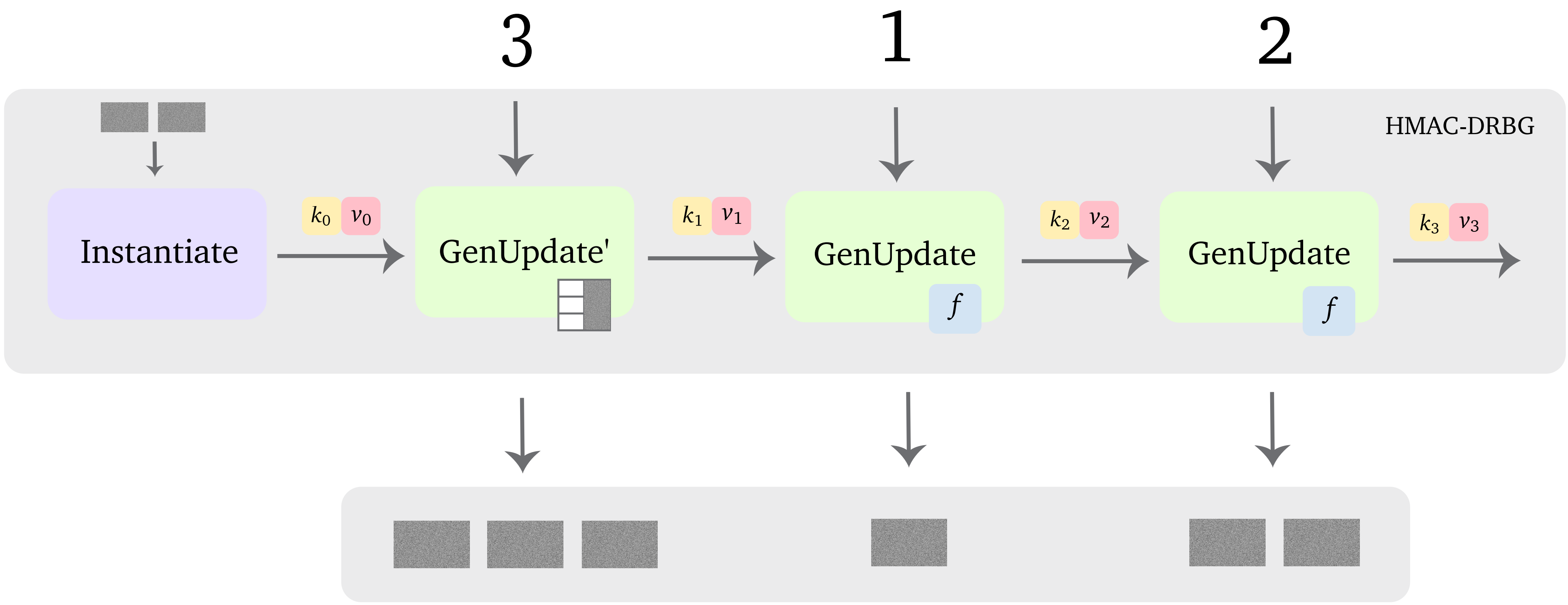


replace PRF with random function

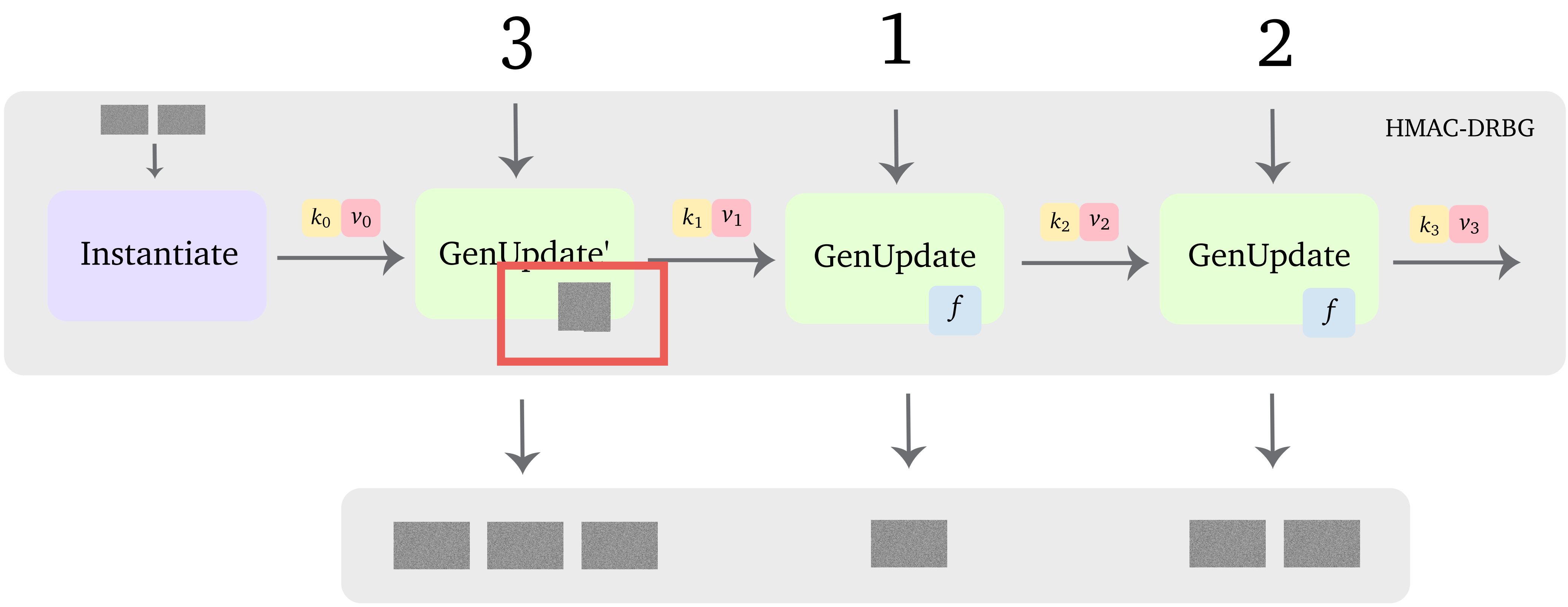


Hybrids



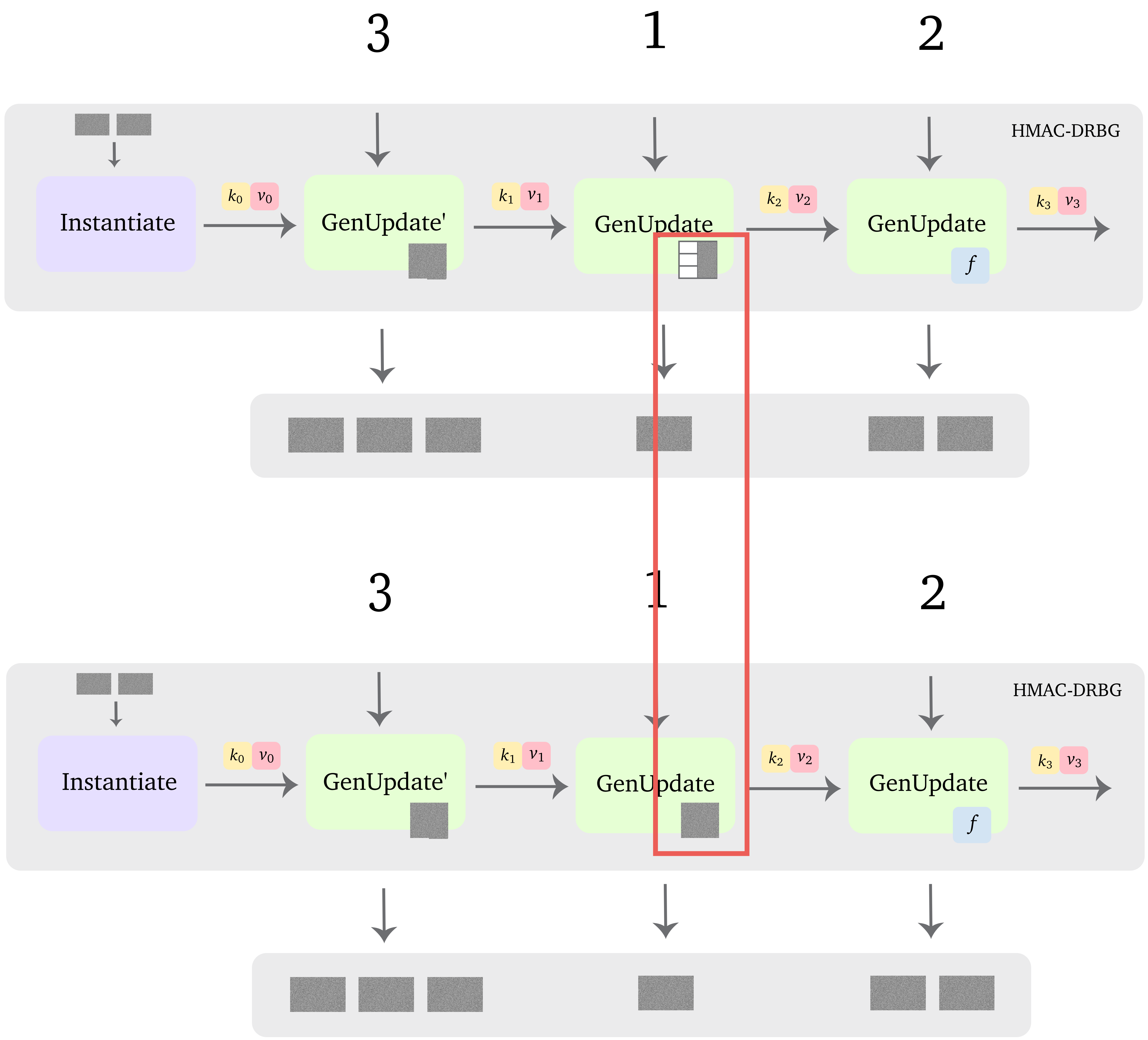


sample all outputs truly randomly



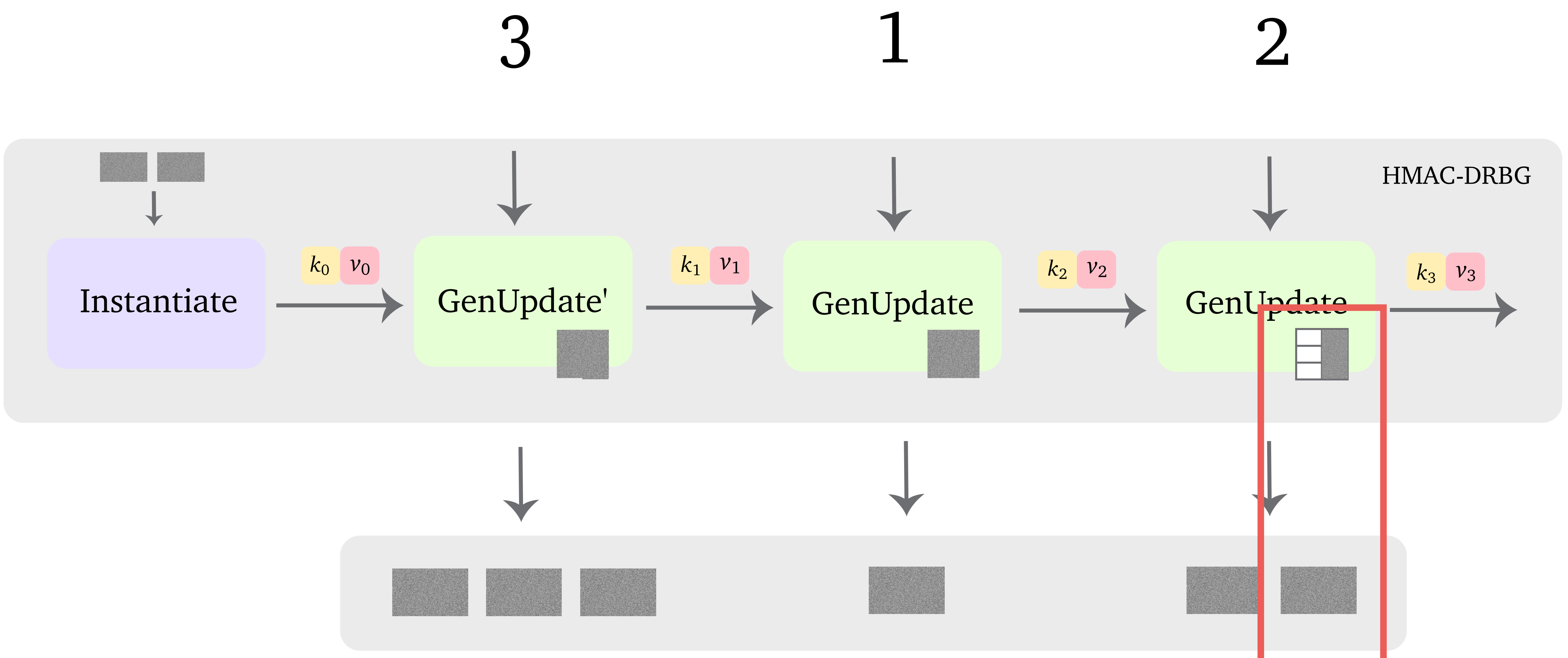
Hybrids

repeat for each call to the PRG

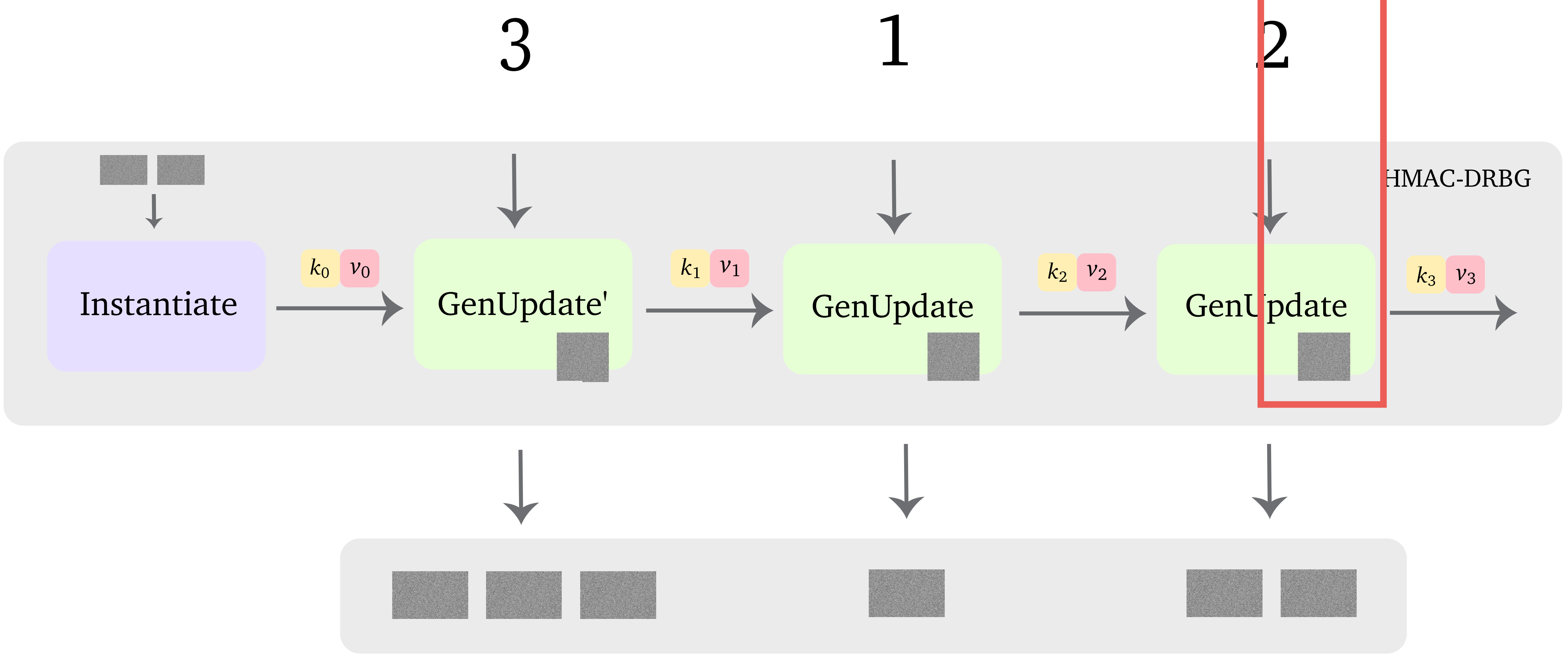


Hybrids

repeat for each call to the PRG



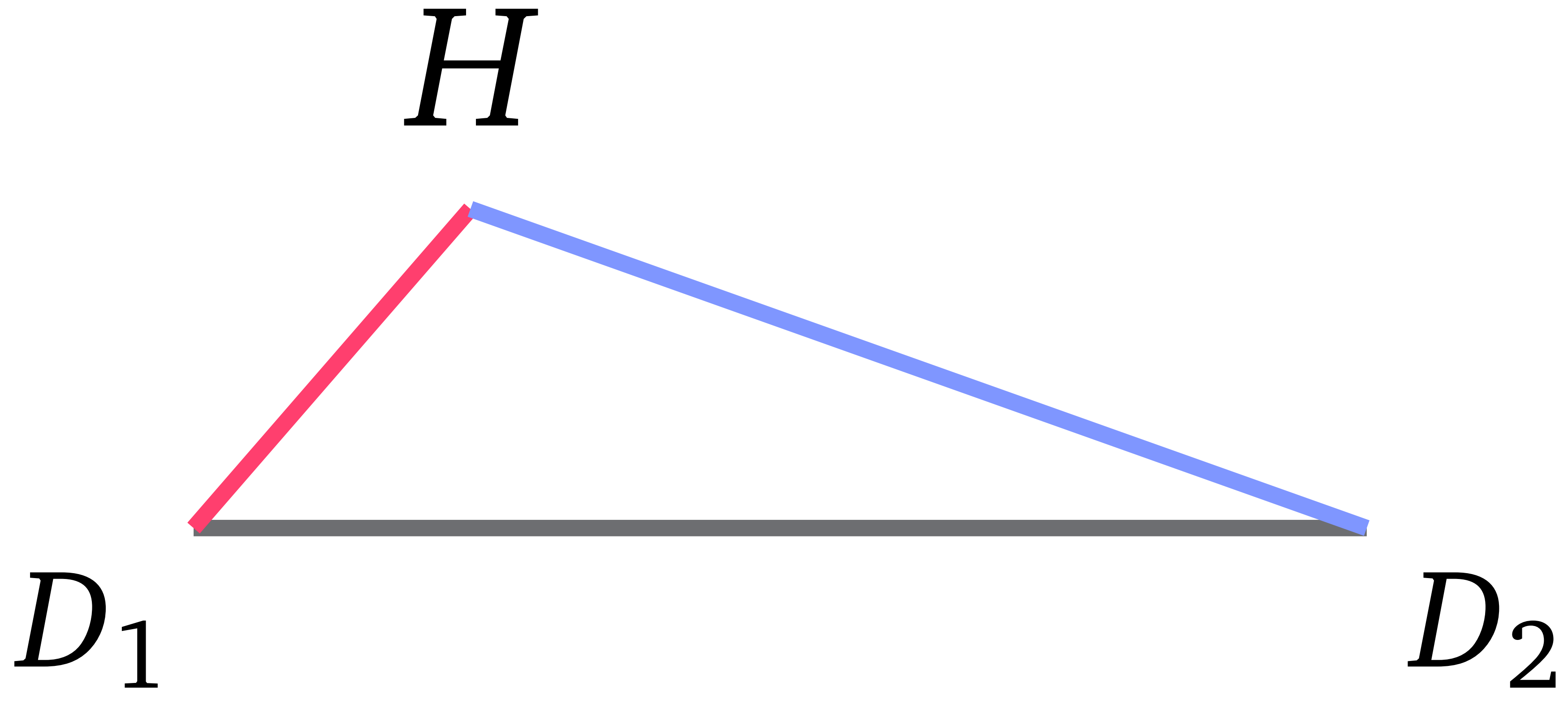
ideal-world hybrid

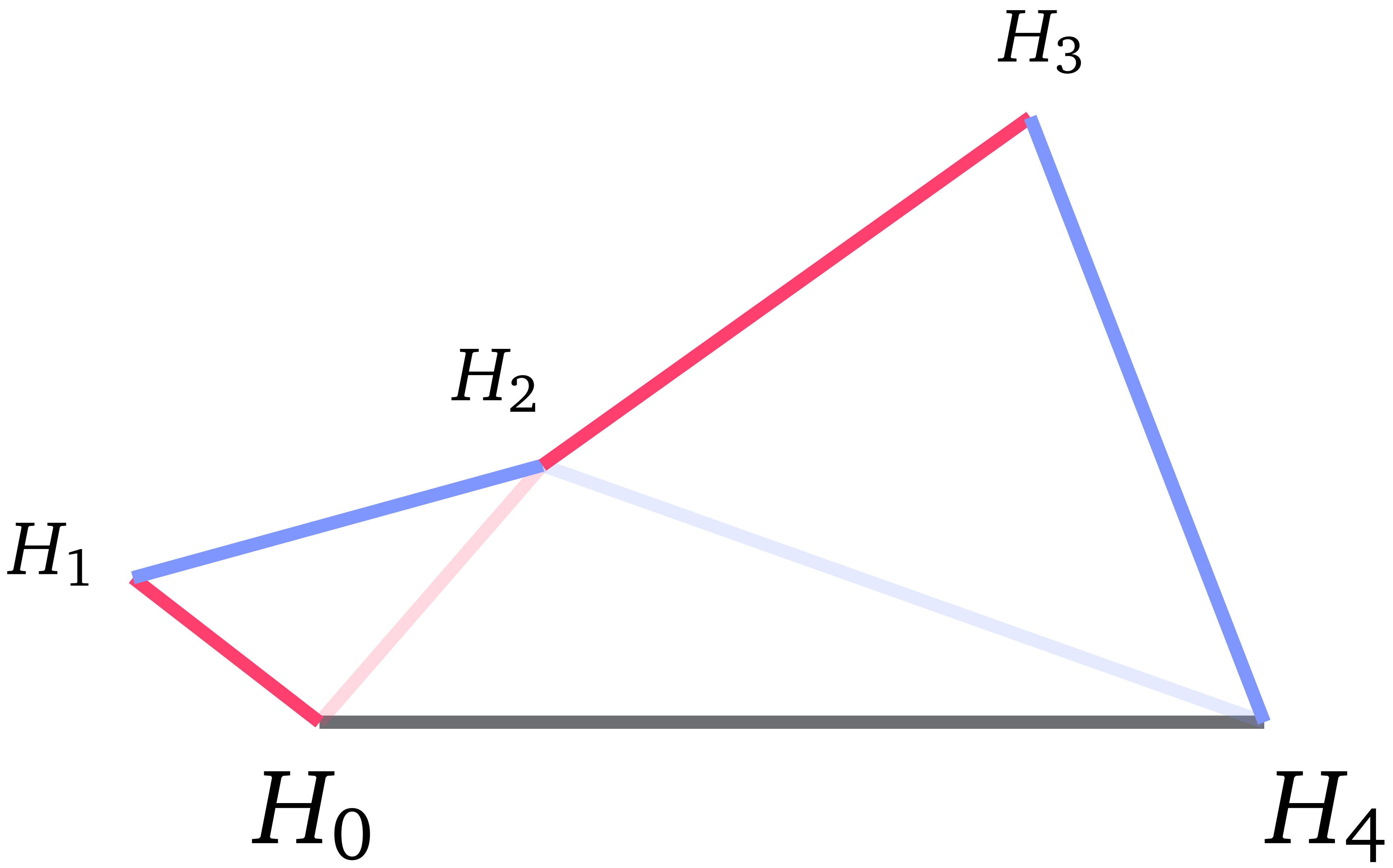


Why are the games close?
The hybrid argument

Repeatedly applying the triangle inequality

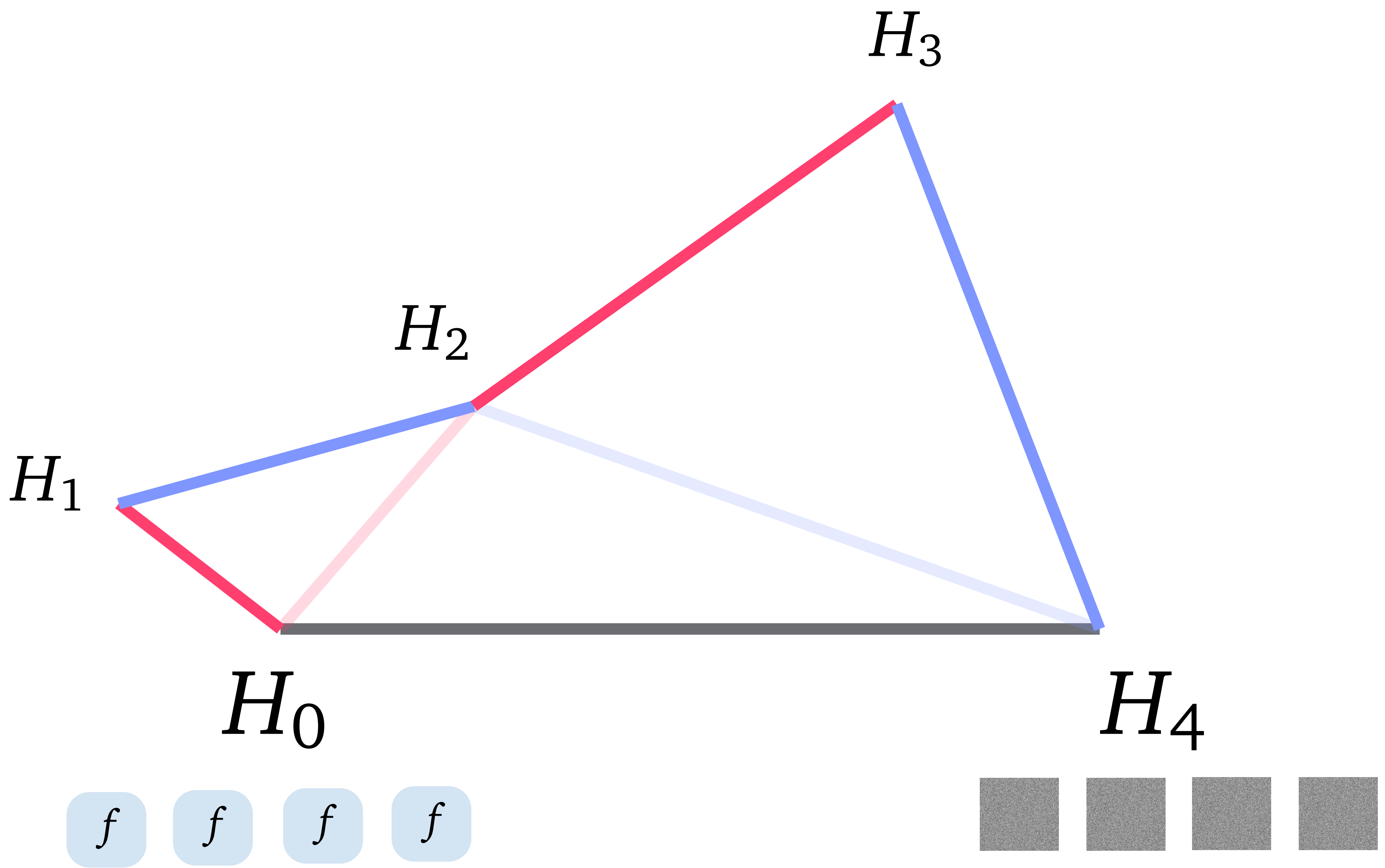
To distinguishing distributions:  A

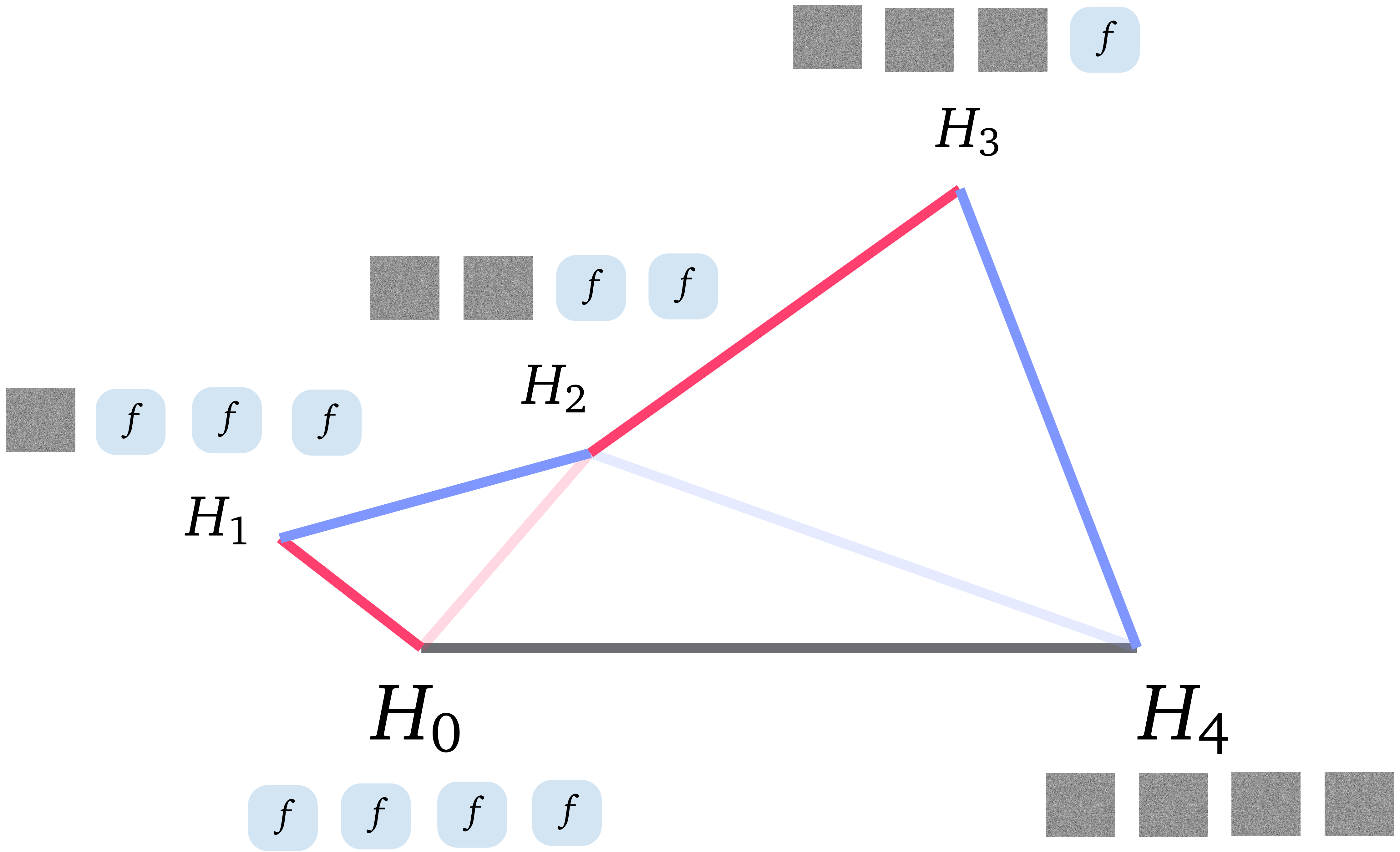




$$\mathrm{Adv}_{D_1,D_2}(\mathbf{A}) \leq \mathrm{Adv}_{D_1,H}(\mathbf{A}) + \mathrm{Adv}_{H,D_2}(\mathbf{A})$$

For the HMAC-DRBG construction, the main hybrids are:



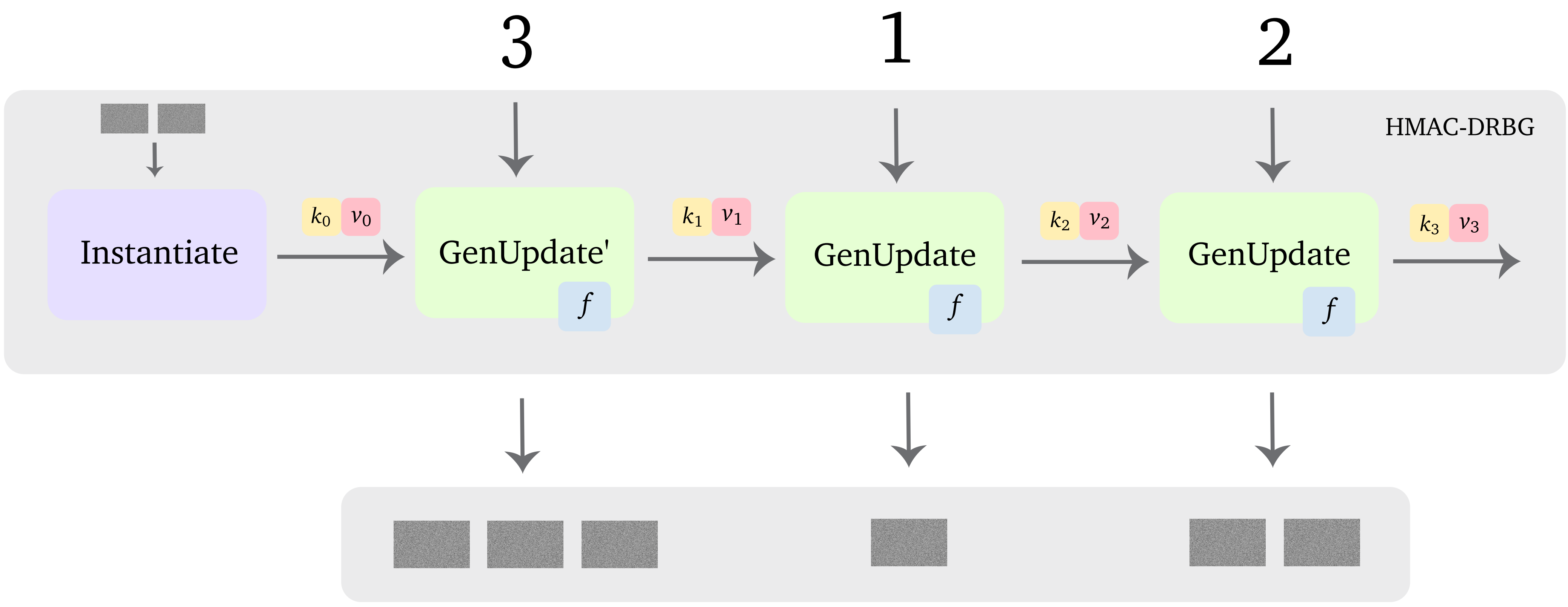


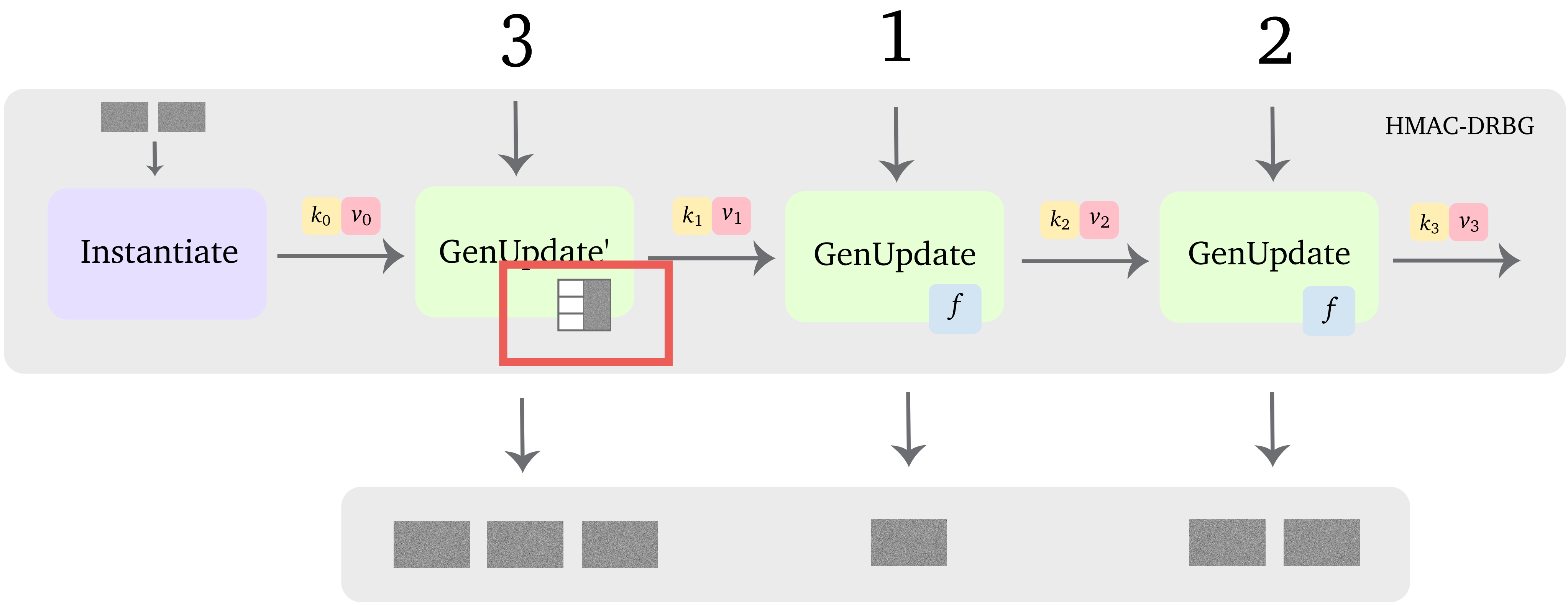
(omitting random functions)

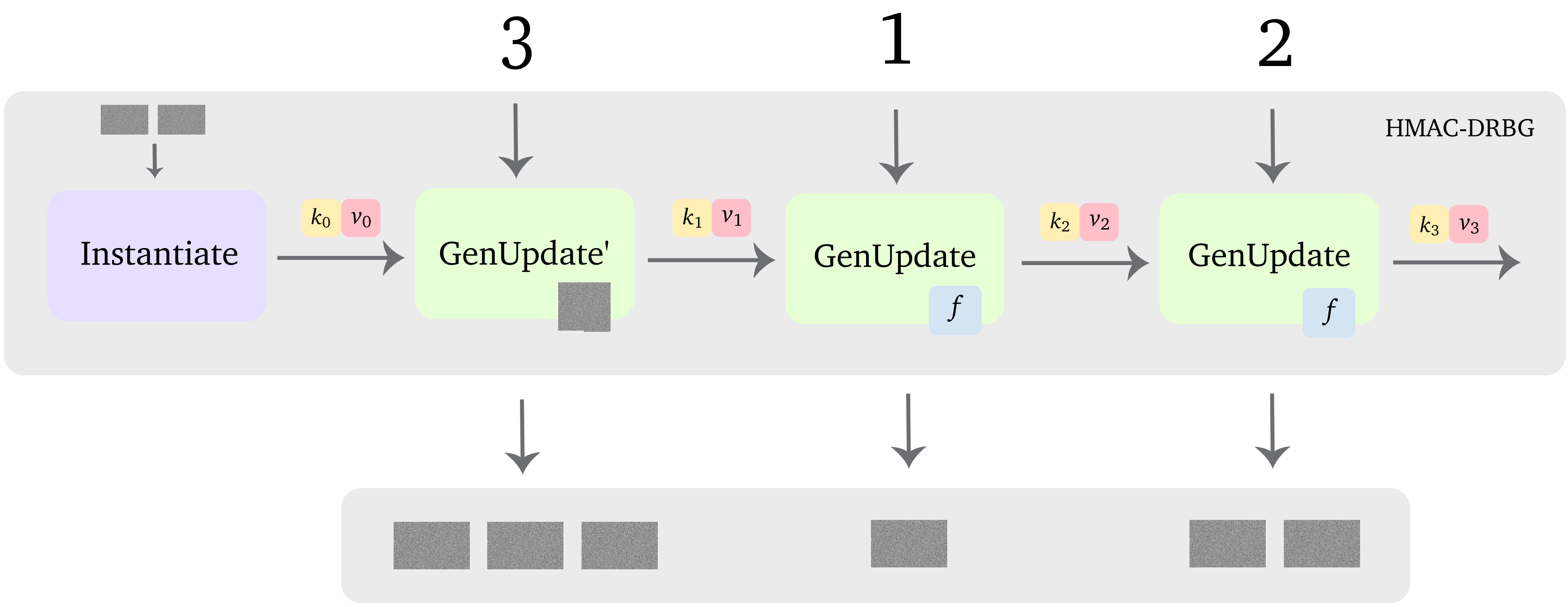
Distinguishing a pseudorandom function from a random function

Hybrids

the chance of
distinguishing between
these two hybrids







Recall: HMAC is a pseudorandom function.

What's a random function again?

Random function as lookup table

Input	Output

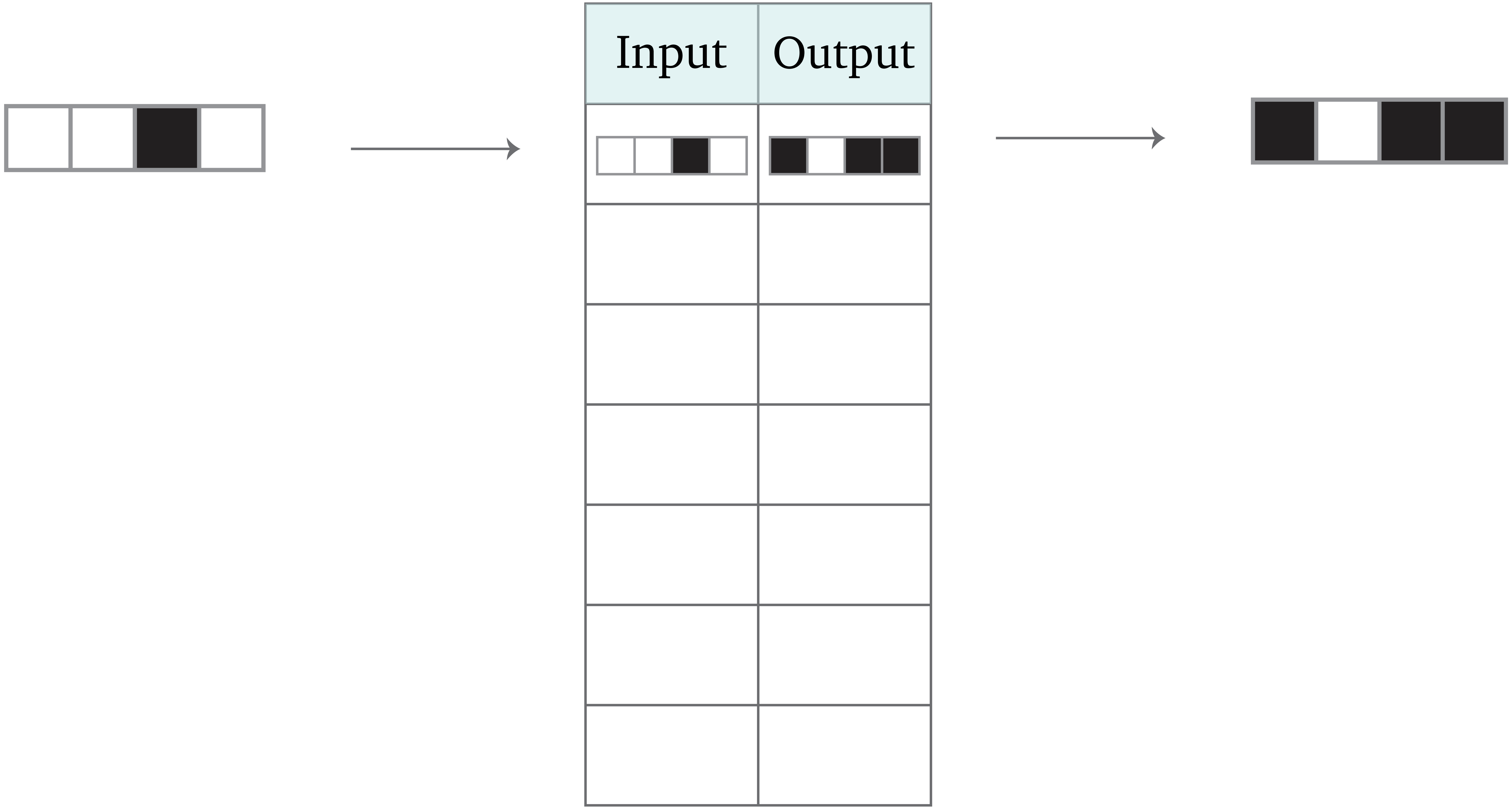
Random function as lookup table

Input	Output
	

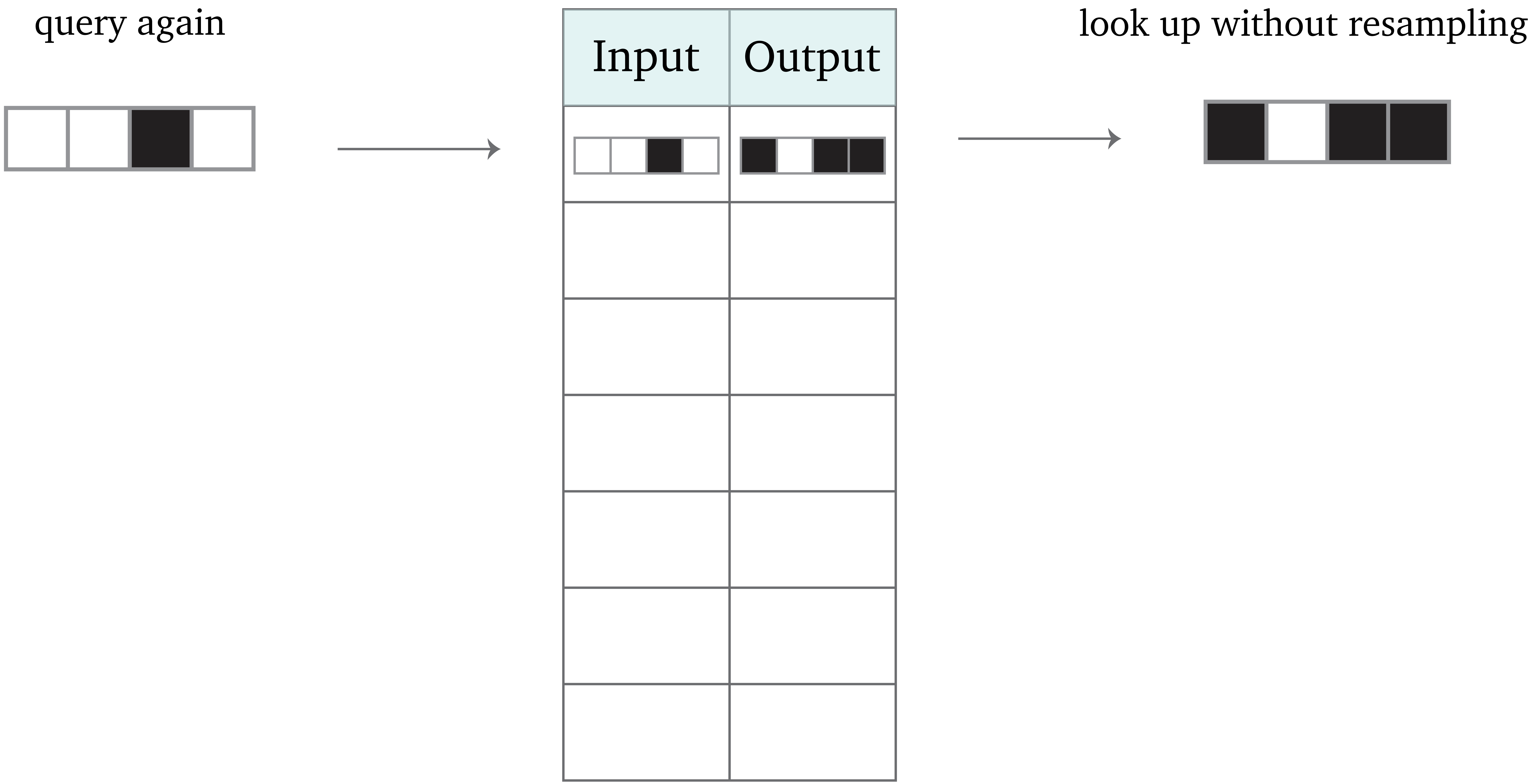
sample
random
output

Input	Output
	

Random function as lookup table

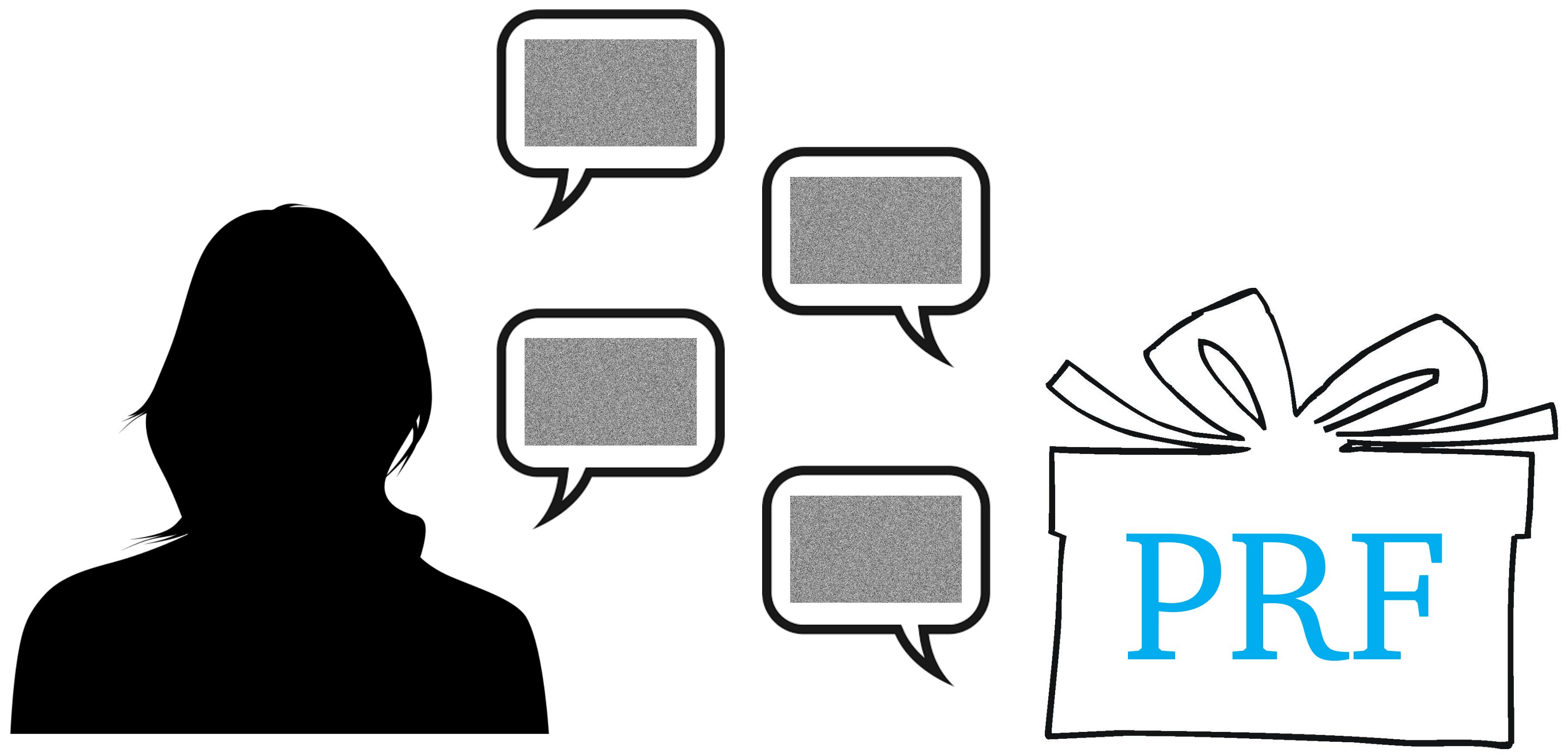


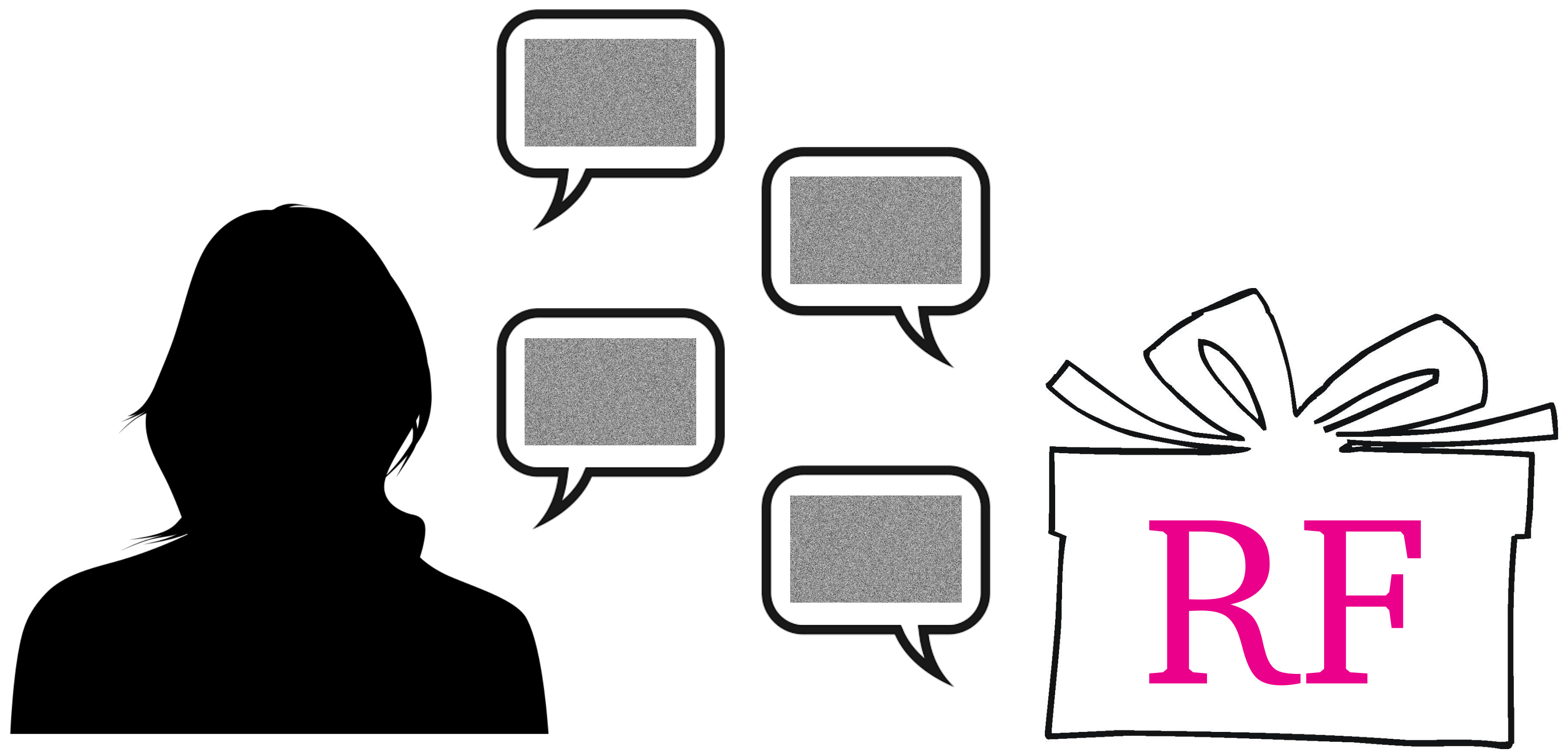
Random function as lookup table

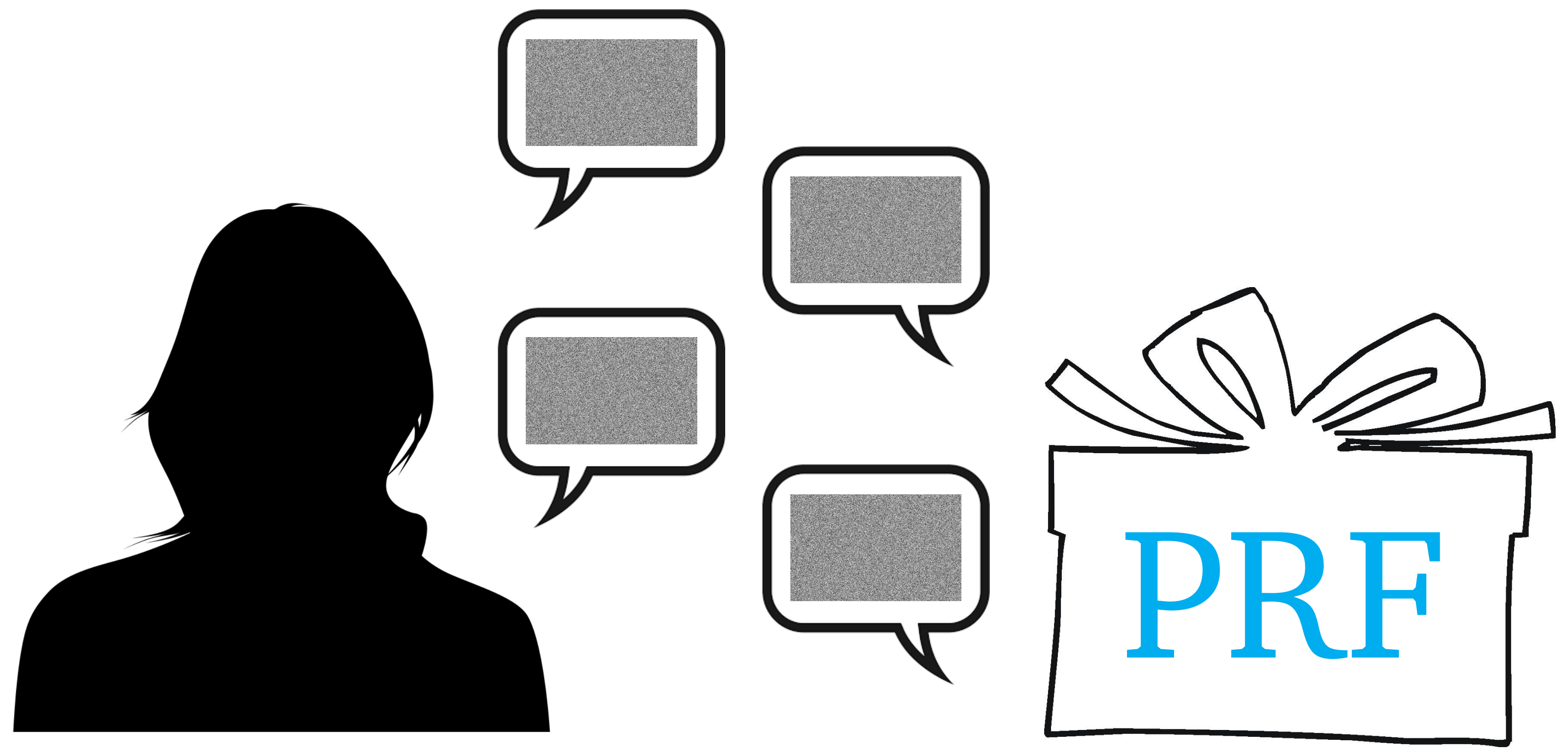


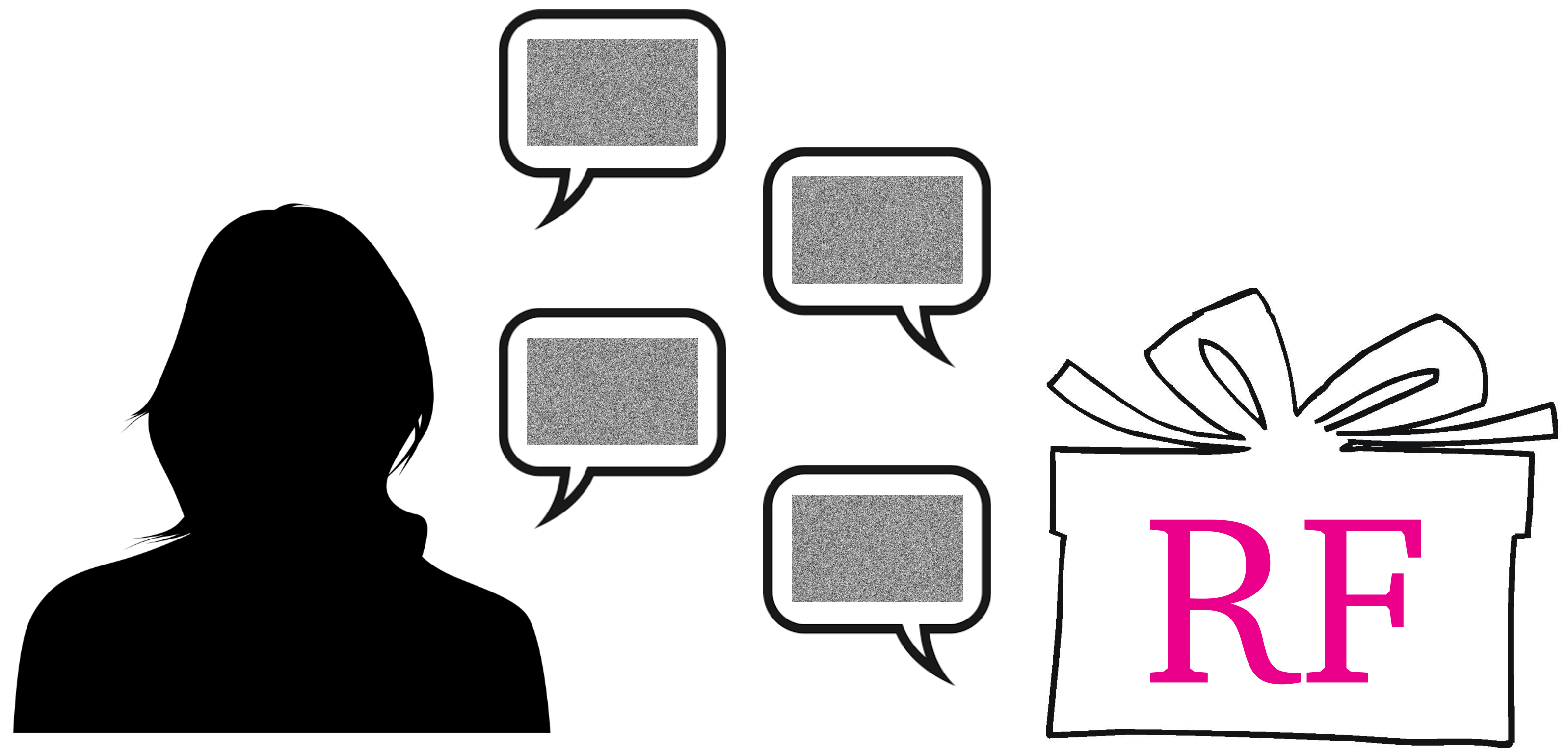
Pseudorandom function
(e.g. HMAC)

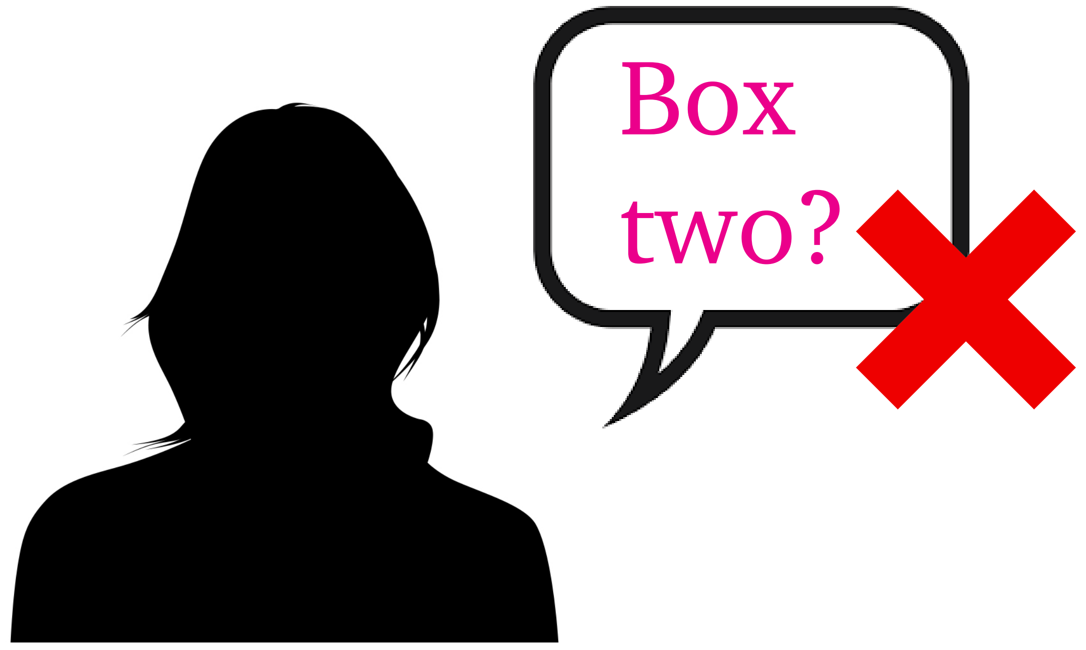
PRF advantage
Find the fake!


PRF


RF


PRF

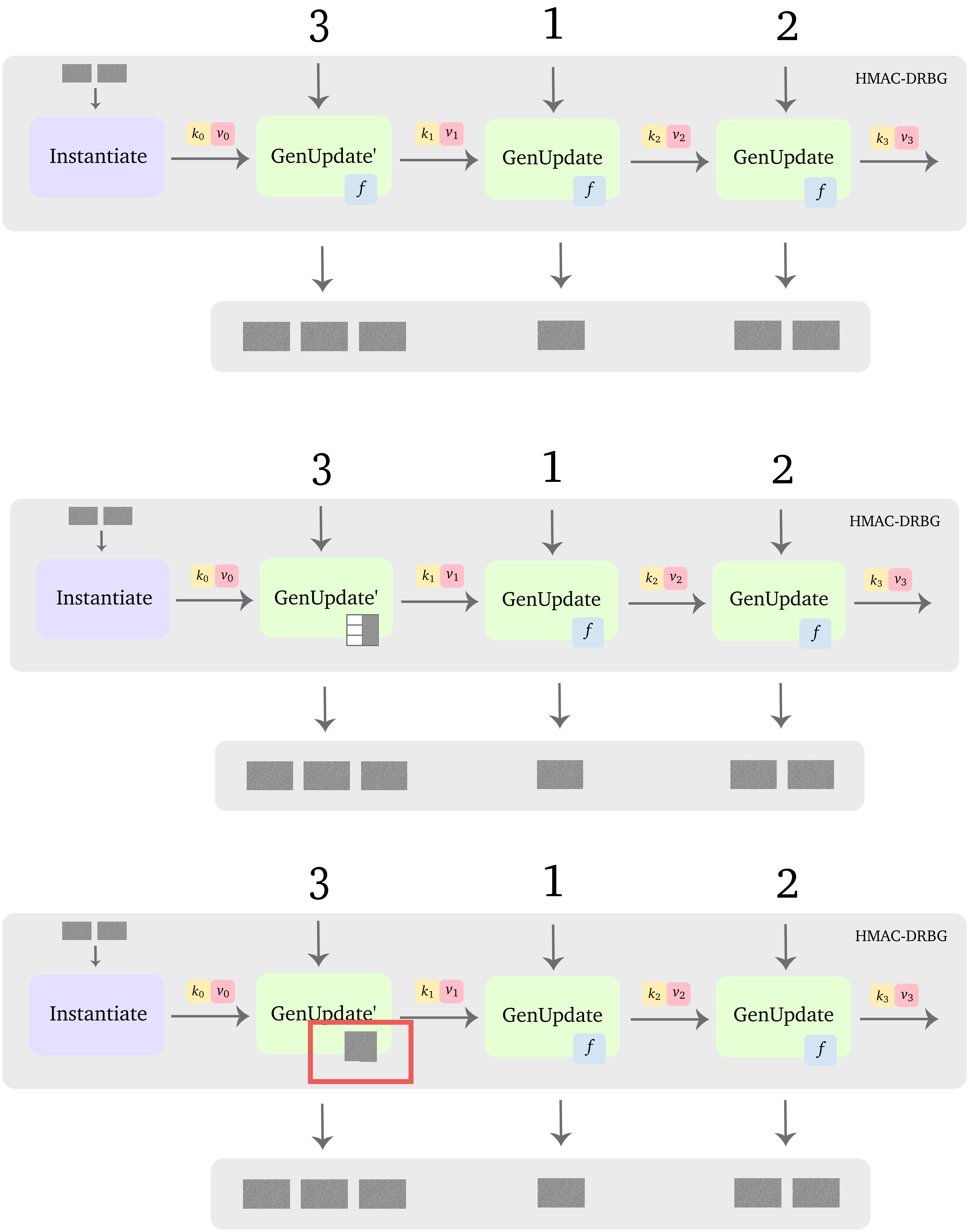

RF


Box two?

Distinguishing a random function from true randomness

(as used in HMAC-DRBG)

Hybrids

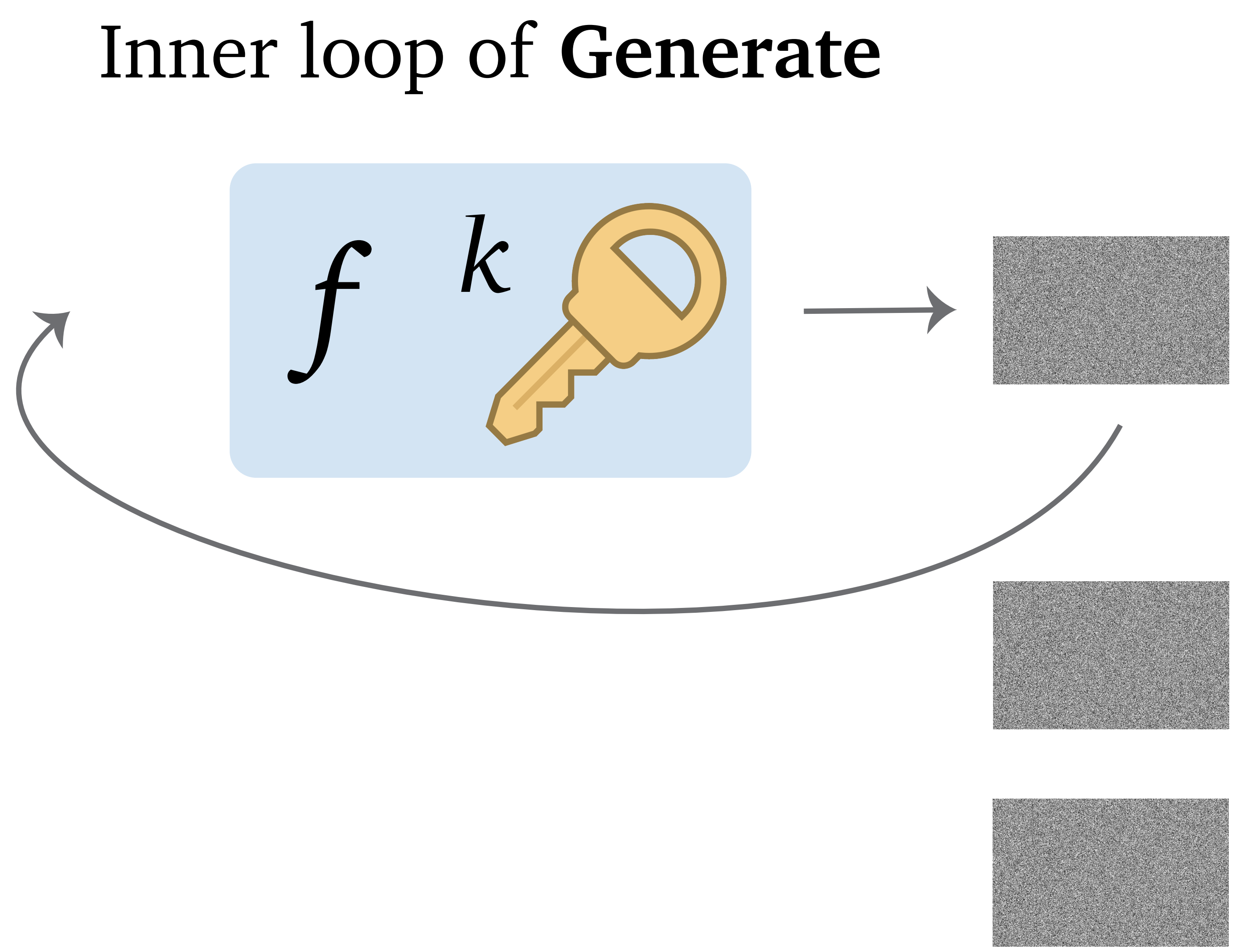


the chance of distinguishing between these two hybrids

Random function vs. true randomness: only noticeable if you call the function on the same input twice

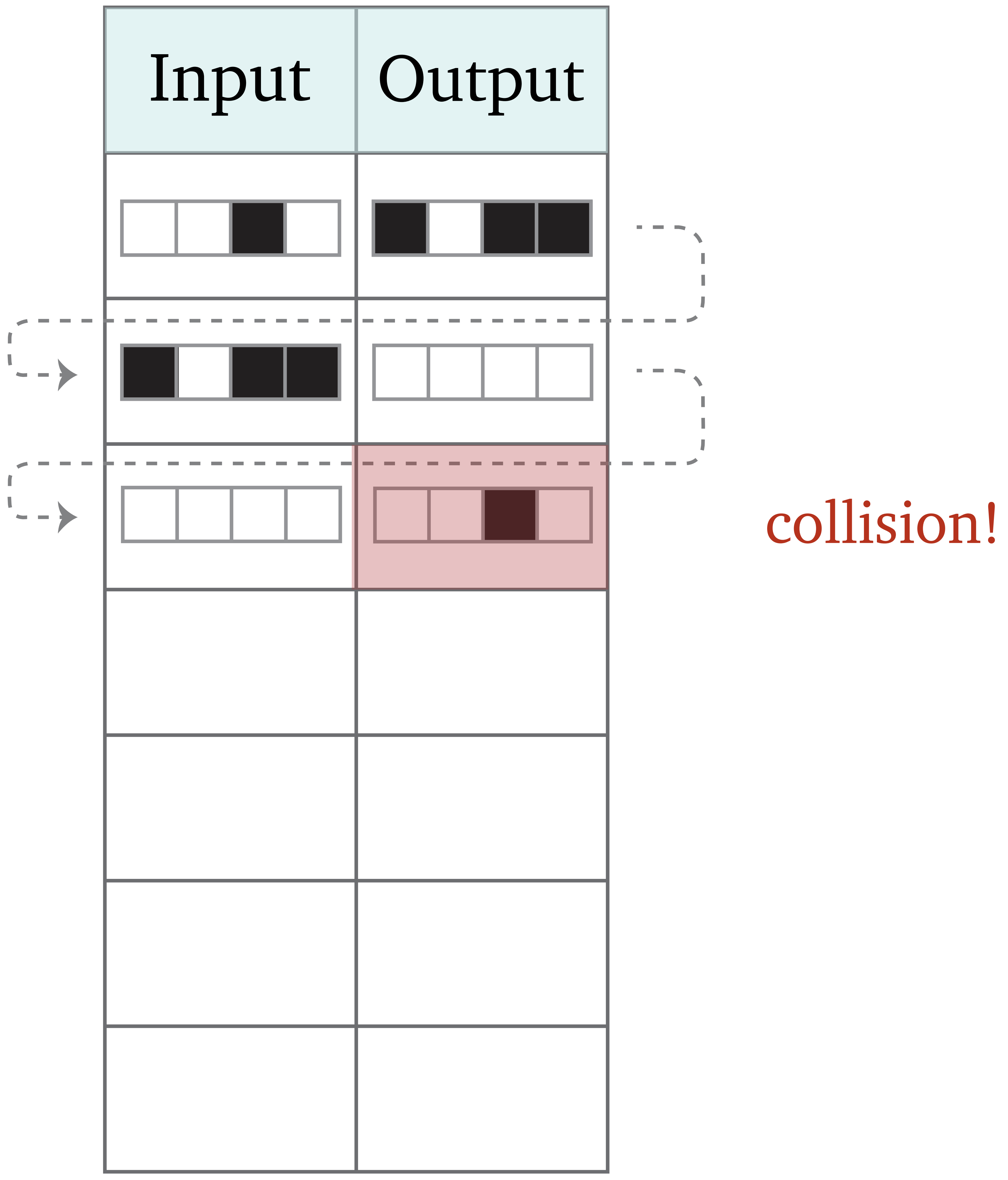
(which we prove is unlikely, as used in HMAC–DRBG)

The outputs are used as inputs



Random function vs. true randomness: only noticeable if you call the function on the same input twice

(which we prove is unlikely, as used in HMAC–DRBG)



Formalizing the proof of indistinguishability

Method

- Code-based game-playing proofs (Bellare and Rogaway, 2008)
- Programs written in a probabilistic programming language; use Hoare-style logic for relating pairs of programs
- Use “identical until bad” lemma
- Machine-checked proof in FCF and Coq

Gray denotes program equivalence proofs.

Proof tree

- The real-world and ideal-world games are close (distinguishable by a PPT A with a particular small probability)
  - The real-world game is the first hybrid
  - Move the v-update in **Update** to the beginning of the next **Generate**
  - Adjacent hybrids G_i and G_{i+1} are close
  - Apply the hybrid argument (induction on i)
  - The ideal-world game is the last hybrid

Full proof tree (to give a sense of the structure)

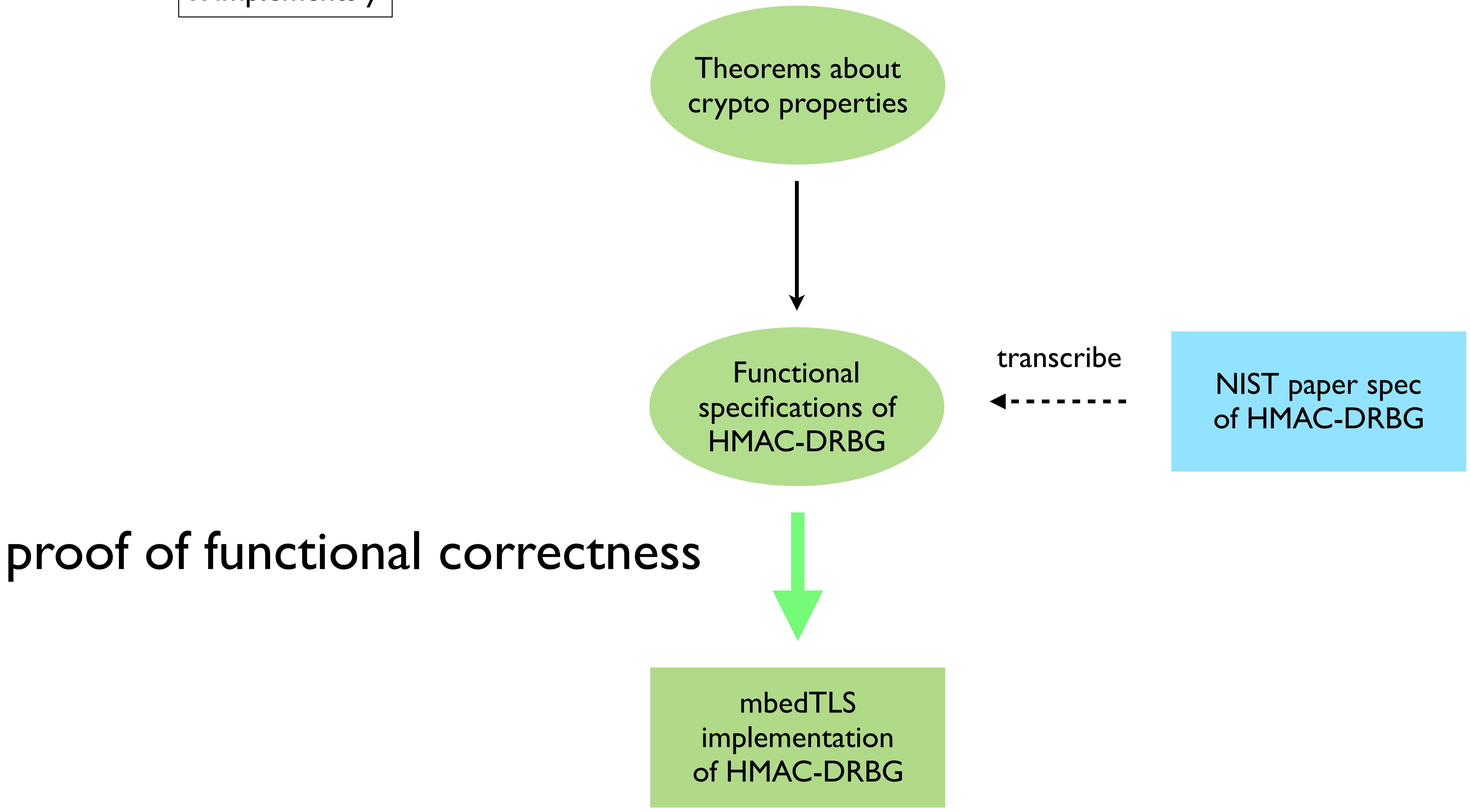
- The real-world and ideal-world games are close (distinguishable by a PPT A with a particular small probability)
  - The real-world game is the first hybrid
  - Move the v-update in **Update** to the beginning of the next **Generate**
  - Adjacent hybrids G_i and G_{i+1} are close
    - Rewrite G_i to use a PRF oracle on the ith call
    - G_i uses a RF oracle on the ith call (denoted G_i'). G_i and G_i' are close.
    - G_i uses a RB oracle on the ith call (denoted G_i''). G_i' and G_i'' are close.
      - Using the RB oracle (G_i'') is equivalent to the next hybrid G_{i+1}.
      - G_i' and G_i'' are identical until bad.
        - Use the "fundamental lemma" to bound the difference between G_i' and G_i''.
        - G_i' and G_i'' are the same if the bad event doesn't happen.
        - The bad event has the same chance of happening in G_i' and G_i''
      - The bad event is the probability of collisions in the inputs to the oracle, which is small
  - Apply the hybrid argument (induction on i)
  - The ideal-world game is the last hybrid

Verifying correctness of the mbedTLS C program

For brevity, we discuss HMAC here.
See *Verified Correctness and Security of OpenSSL HMAC* (Beringer et al, USENIX Security '15) for details.

Our work

x → y:
x implements y



Proofs about functions in Coq

```
Fixpoint map {A B} (f: A->B) (al: list A) :=
match al with
| a::r => f a :: map f r
| nil => nil
end
```

```
Fixpoint cat {A} (al bl: list A) :=
match al with
| a::r => a :: cat r bl
| nil => nil
end.
```

```
Theorem distr_map_cat:   forall {A B} (f: A->B) (al bl: list A),
  map f (cat al bl) = cat (map f al) (map f bl).

Proof. intros A B f al bl. induction al. reflexivity. simpl. rewrite IHal. reflexivity. Qed.
```

This is a rather trivial theorem. A more interesting one is, "The HMAC-DRBG algorithm, expressed as a function in Gallina, produces cryptographically strong pseudorandom output."

Proofs about C programs

```
struct list catenate (struct list *p, struct list *q) {
 if (p==NULL) return q;
 while (p->tail != NULL) p=p->tail;
 p->tail=q;
 return p;
}
```

DECLARE _catenate
WITH p: val, q: val, σ_1: list val, σ_2: list val
PRE [_p OF tptr (tstruct _list), _q OF tptr (tstruct _list)]
 PROP() LOCAL (temp _p p; temp _q q) SEP (listrep σ_1 p; listrep σ_2 q)
POST [tptr (tstruct _list)]
 EX v: val, PROP() LOCAL (temp ret_temp v) SEP (listrep (cat σ_1 σ_2) v).

This is a rather trivial theorem. A more interesting one is,
“The mbedTLS implementation of HMAC-DRBG correctly implements the HMAC-DRBG algorithm expressed as a function in Gallina.”

Proofs about C programs

```
struct list catenate (struct list *p, struct list *q) {
 if (p==NULL) return q;
 while (p->tail != NULL) p=p->tail;
 p->tail=q;
 return p;
}
```

DECLARE _catenate
WITH p: val, q: val, σ_1: list val, σ_2: list val
PRE [_p OF tptr (tstruct _list), _q OF tptr (tstruct _list)]
 PROP() LOCAL (temp _p p; temp _q q) SEP (listrep σ_1 p; listrep σ_2 q)
POST [tptr (tstruct _list)]
 EX v: val, PROP() LOCAL (temp ret_temp v) SEP (listrep $\boxed{(\text{cat } \sigma_1\ \sigma_2)}$ v).

Functional spec

This is a rather trivial theorem. A more interesting one is,
"The mbedTLS implementation of HMAC-DRBG correctly implements the HMAC-DRBG algorithm expressed as a function in Gallina."

HMAC function

```
Definition HmacCore
    IP OP txt (key: list byte): list Z :=
    OUTER OP key (INNER IP key txt).
```

https://github.com/PrincetonUniversity/VST/blob/8750fdd00c8a3156e5103d2f9924b9de3c6ca7b2/sha/HMAC_functional_prog.v
https://github.com/PrincetonUniversity/VST/blob/8750fdd00c8a3156e5103d2f9924b9de3c6ca7b2/hmacdrbg/HMAC_DRBG_algorithms.v

HMAC-DRBG **Generate** function

```
Function HMAC_DRBG_generate_helper_Z
  (HMAC: list Z -> list Z -> list Z)
  (key v: list Z)(requested_number_of_bytes: Z)
  {measure Z.to_nat requested_number_of_bytes} : (list Z * list Z) :=
  if 0 >=? requested_number_of_bytes then (v, [])
  else
    let len := 32%nat in
    let (v, rest) := HMAC_DRBG_generate_helper_Z HMAC key v
                      (requested_number_of_bytes - (Z.of_nat len)) in
    let v := HMAC v key in
    let temp := v in
    (v, rest ++ temp).
```

https://github.com/PrincetonUniversity/VST/blob/8750fdd00c8a3156e5103d2f9924b9de3c6ca7b2/sha/HMAC_functional_prog.v
https://github.com/PrincetonUniversity/VST/blob/8750fdd00c8a3156e5103d2f9924b9de3c6ca7b2/hmacdrbg/HMAC_DRBG_algorithms.v

HMAC API in C

```
unsigned char *HMAC (
    unsigned char *key,
    int key_len,
    unsigned char *d,
    int n,
    unsigned char *md);
```

Key input

Message input

Message-digest output

API Spec of HMAC

Common logical variables

```
DECLARE _HMAC
WITH kp: val, key:DATA, KV:val, msgVal: val, MSG:DATA, shmd: share, md: val
PRE [ _key OF char*, _key_len OF int, _d OF char*,  _n OF int, _md OF char*]
  PROP(writable share shmd;
          has lengthK (LEN key) (CONT key);
          has lengthD 512 (LEN msg) (CONT msg))
  LOCAL(temp _md md; temp _key kp; temp _d msgVal;
          temp _key_len (Vint (Int.repr (LEN key)));
          temp _n (Vint (Int.repr (LEN msg)));
          var _K256 (tarray tuint 64) KV)
   SEP(data-block Tsh (CONT key) kp;
        data-block Tsh (CONT msg) msgVal;
        K-vector KV;
        memory-block shmd (Int.repr 32) md)
POST [ tvoid ]
   SEP(K-vector KV;
        data-block shmd (HMAC (CONT msg) (CONT key)) md;
        data-block Tsh (CONT key) kp;
        data-block Tsh (CONT msg) msgVal)
```

PRECONDITION

Logical Propositions

Local/global variable bindings

Spatial (memory) predicates

POSTCONDITION

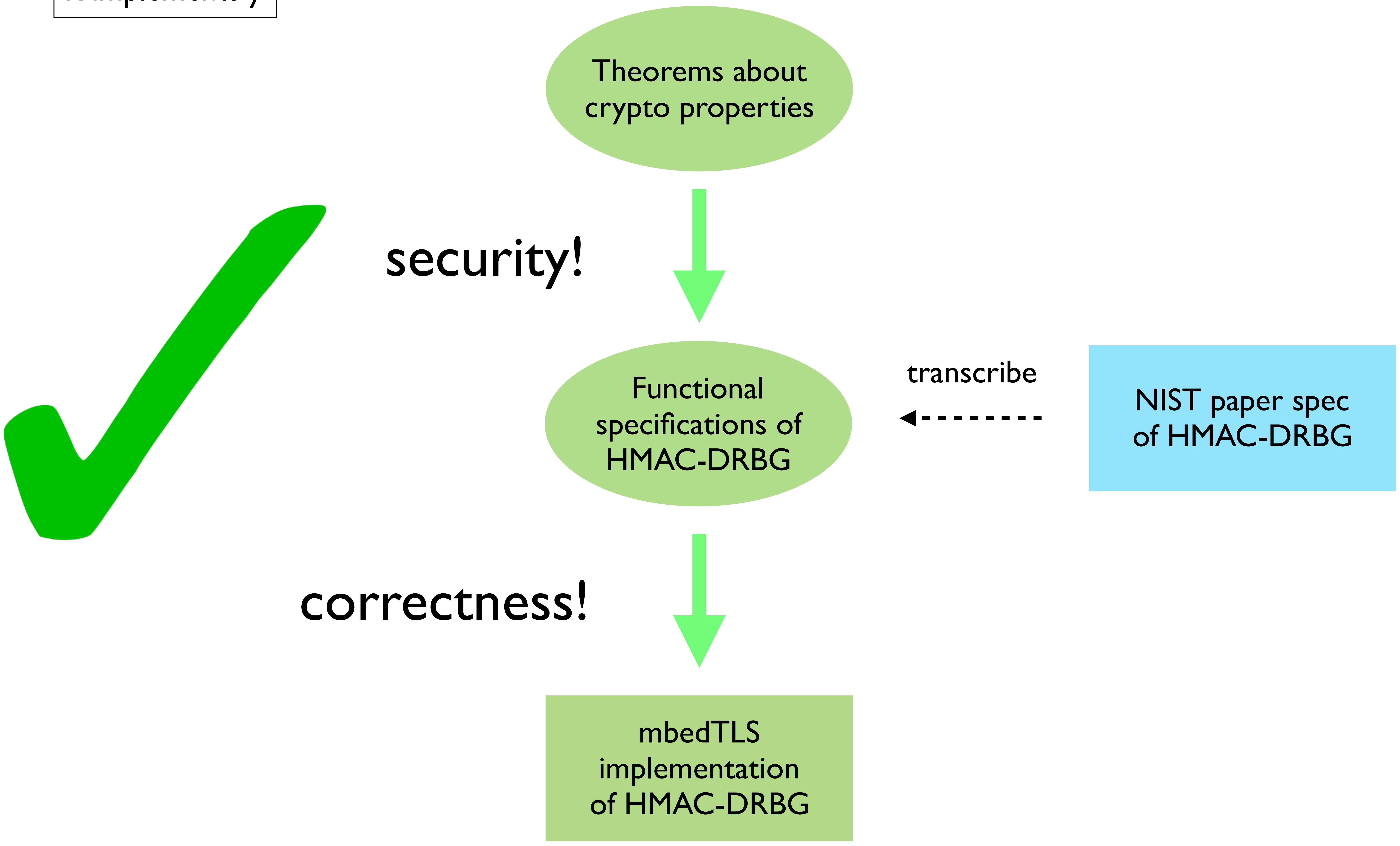
Functional spec

Our work

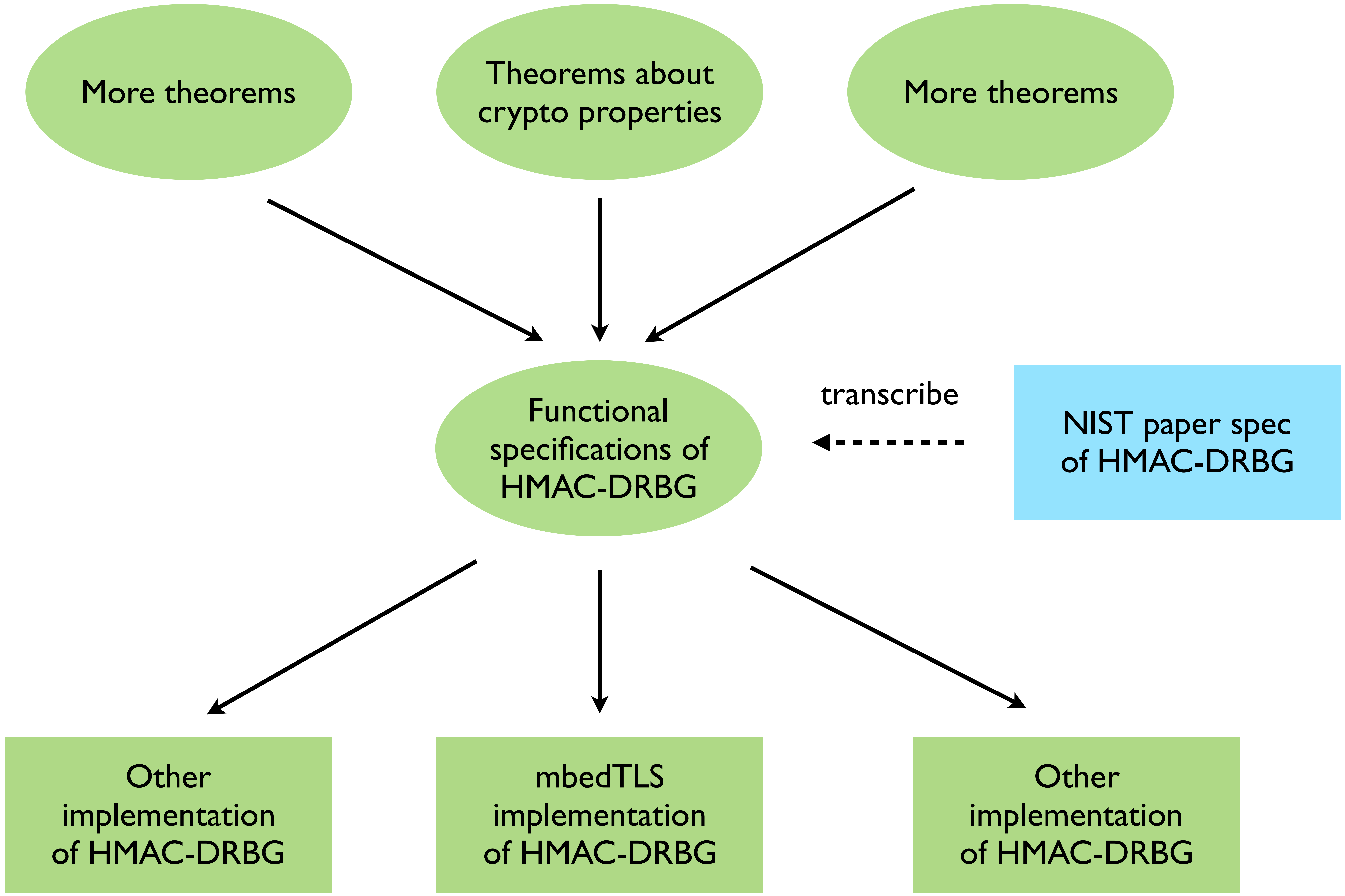
First end-to-end formal security-and-correctness verification of a real-world PRG.

Our work

x → y:
x implements y



Modular proofs



Open problem

- Security of HMAC-DRBG **Instantiate** relies on HMAC being an entropy extractor
- It is not known whether HMAC is an entropy extractor (the way that HMAC-DRBG uses it)!

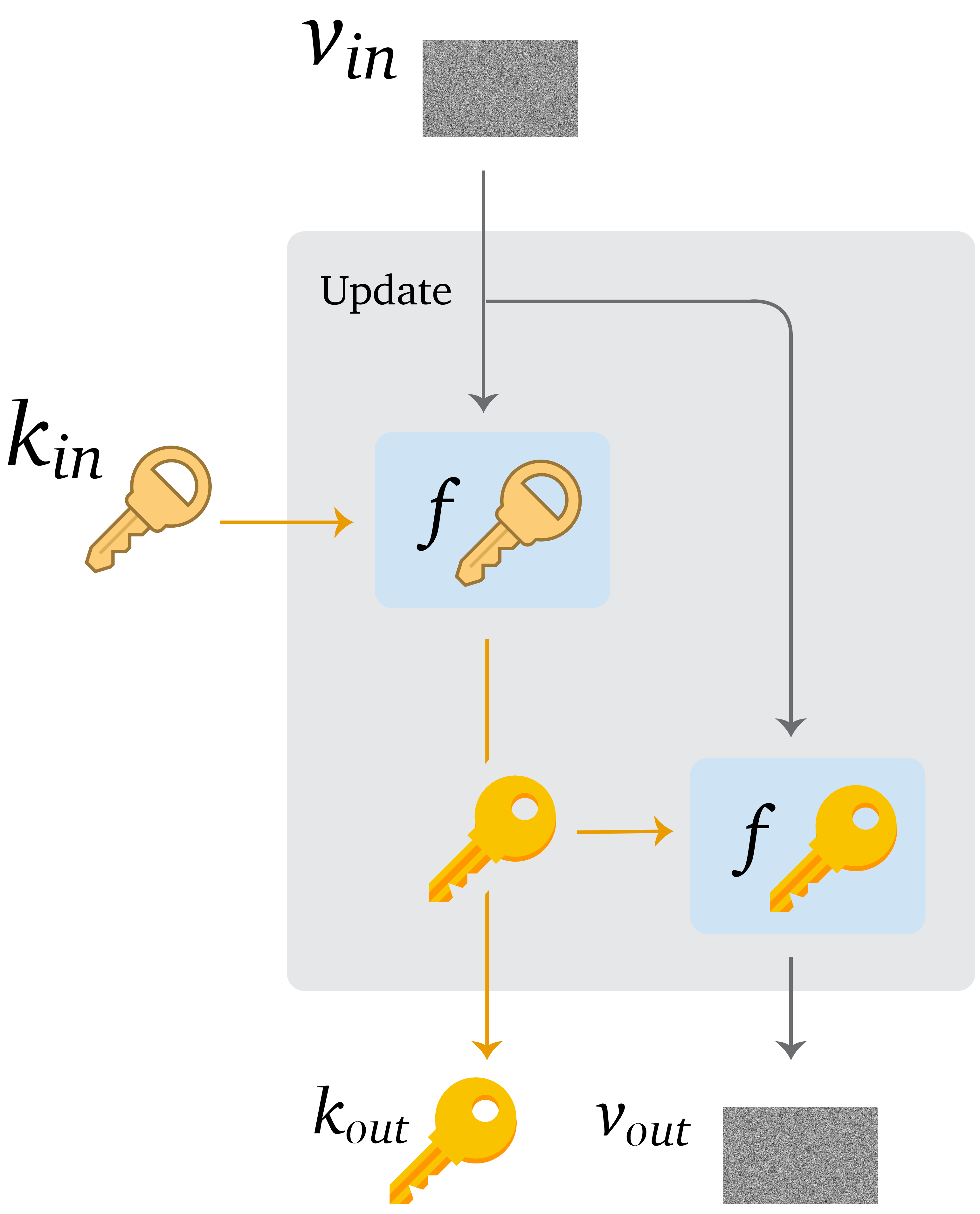
Future work

Prove more security properties of PRGs, *e.g.* backtracking resistance and prediction resistance

Lessons learned

- NIST design decisions: the good (PRF re-key method), the bad (**Instantiate** key with entropy in PRF as input, not key), the ugly (re-key location)

- Verification helps deal with tricky indices and typos in argument



Lessons learned

- Stitch together proofs via machine-checking (see KRACK)

- Formal specifications are useful and necessary!

Key Reinstallation Attacks: Forcing Nonce Reuse in WPA (Vanhoef and Piessens, CCS '17) deep spec crypto

Verified Software Toolchain

Thanks!

Don't believe us?
Check out the artifact:

github.com/PrincetonUniversity/VST/tree/master/hmacdrbg

Appendix

HMAC_DRBG Instantiate Process:

1. *seed_material* = *entropy_input* || *nonce* || *personalization_string*.
2. *Key* = 0x00 00...00. Comment: *outlen* bits.
3. *V* = 0x01 01...01. Comment: *outlen* bits.

   Comment: Update *Key* and *V*.
4. (*Key*, *V*) = **HMAC_DRBG_Update** (*seed_material*, *Key*, *V*).
5. *reseed_counter* = 1.
6. Return *V*, *Key* and *reseed_counter* as the *initial_working_state*.

HMAC_DRBG Update Process:

1. *K* = **HMAC** (*K*, *V* || 0x00 || *provided_data*).
2. *V* = **HMAC** (*K*, *V*).
3. If (*provided_data* = *Null*), then return *K* and *V*.
4. *K* = **HMAC** (*K*, *V* || 0x01 || *provided_data*).
5. *V* = **HMAC** (*K*, *V*).
6. Return *K* and *V*.

Questions

- What's the trusted code base?
- What bugs or attacks does your method *not* prevent? What about side-channels?
- How well does your method scale to larger codebases?
- Is your proof still valid if the underlying mbedTLS code changes?
- Can I apply your method to verify other faster or better DRBGs, like AES-DRBG?
- Would your method have prevented real-world incidents like the Debian OpenSSL fiasco or the Juniper bugs?

Questions

- How would you prove other security properties of DRBGs, like backtracking resistance and prediction resistance?
- How does your proof link with other proofs that might involve HMAC-DRBG, like proving security of TLS?
- What does the bound on your proof mean? Concretely, how long would it take an adversary to break HMAC-DRBG indistinguishability by brute force?
- So your proof means everyone should be using mbedTLS HMAC-DRBG, right?

Questions

- Did your proof involve any new math? How does it differ from Hirose's proof?
- Does it matter that you assume a nonadaptive adversary?
- How does assuming an ideal *Instantiate* (without entropy) weaken your proof? What about the additional input?
- Why use the logics of PRHL and Verifiable C?