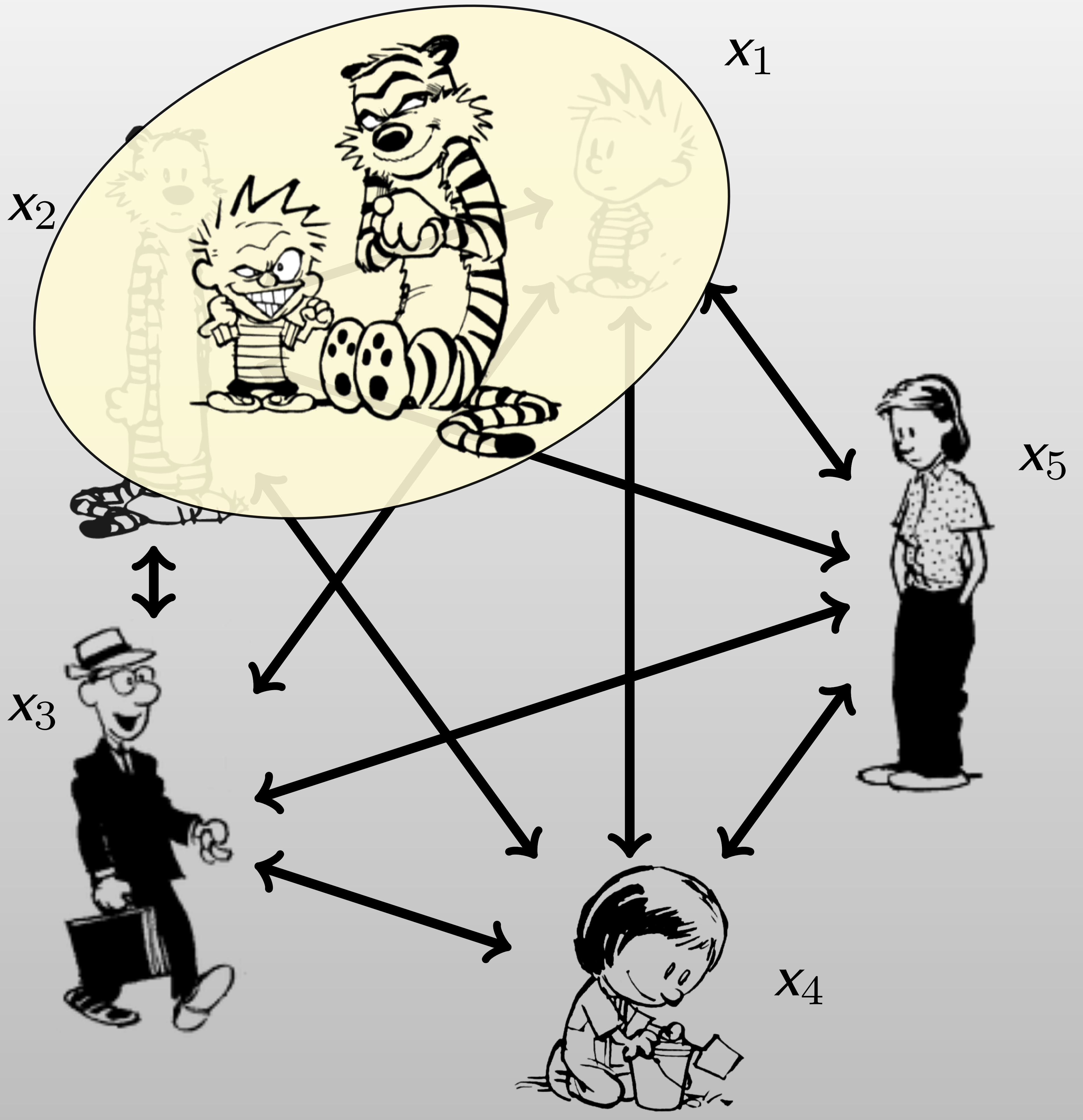
Lecture 22: Protected databases, randomness

- Final exam is Saturday, May 11 at 3-5pm in PHO 211 (usual classroom)
- Review session for the final is Sat 5/4 at 3-5pm in MCS 180
  - No office hours next week
- Please complete course evaluation at bu.campuslabs.com/courseeval by Monday 5/6 (apparently this class has a low response rate so far)
- Have a good summer!

Secure computation



$$\therefore f(x_1, x_2, x_3, x_4, x_5)$$

Premise:

- Mutually distrusting parties, each with a private input
- Learn the result of agreed-upon computation
- *Ex:* election, auction, etc.

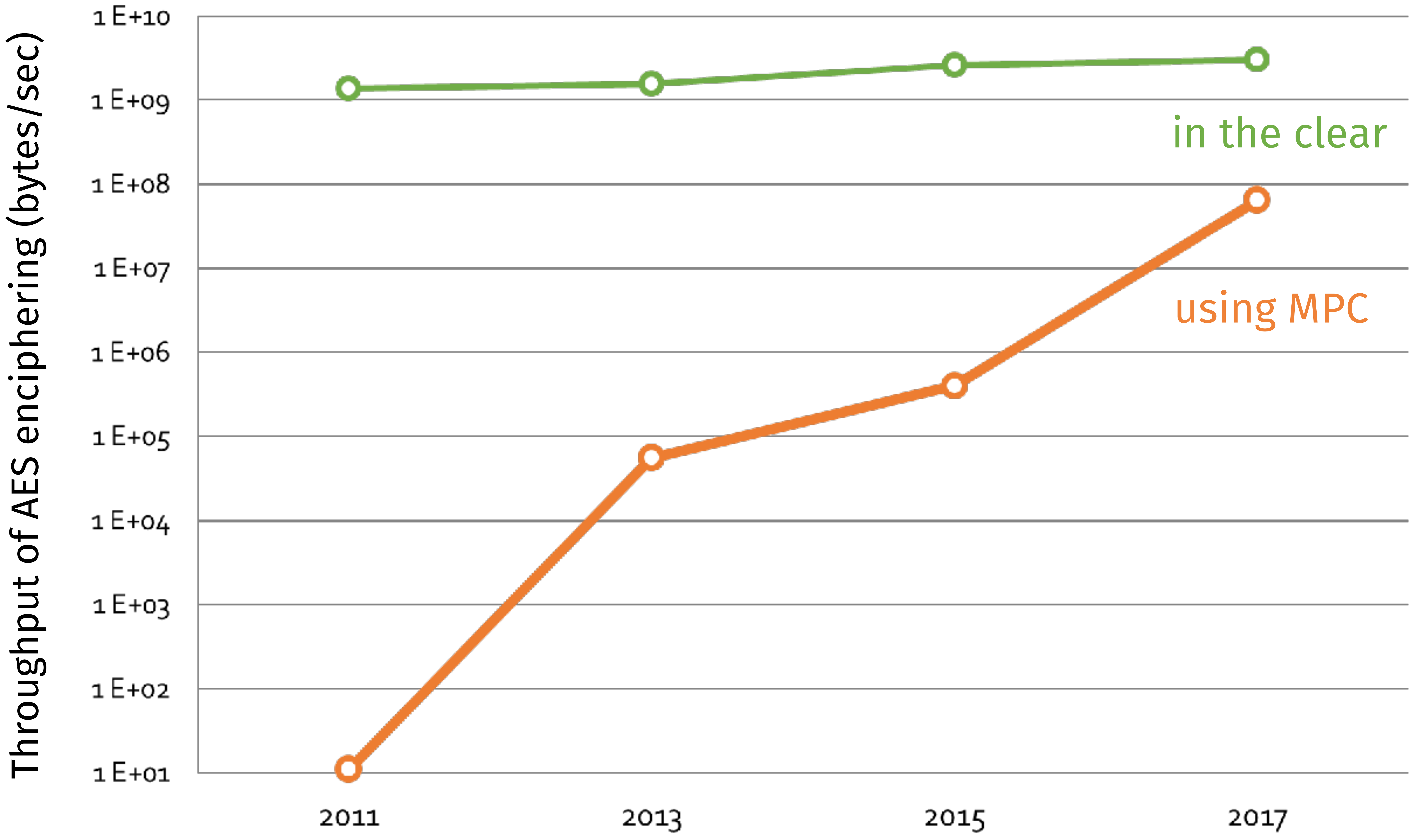
Security guarantees:

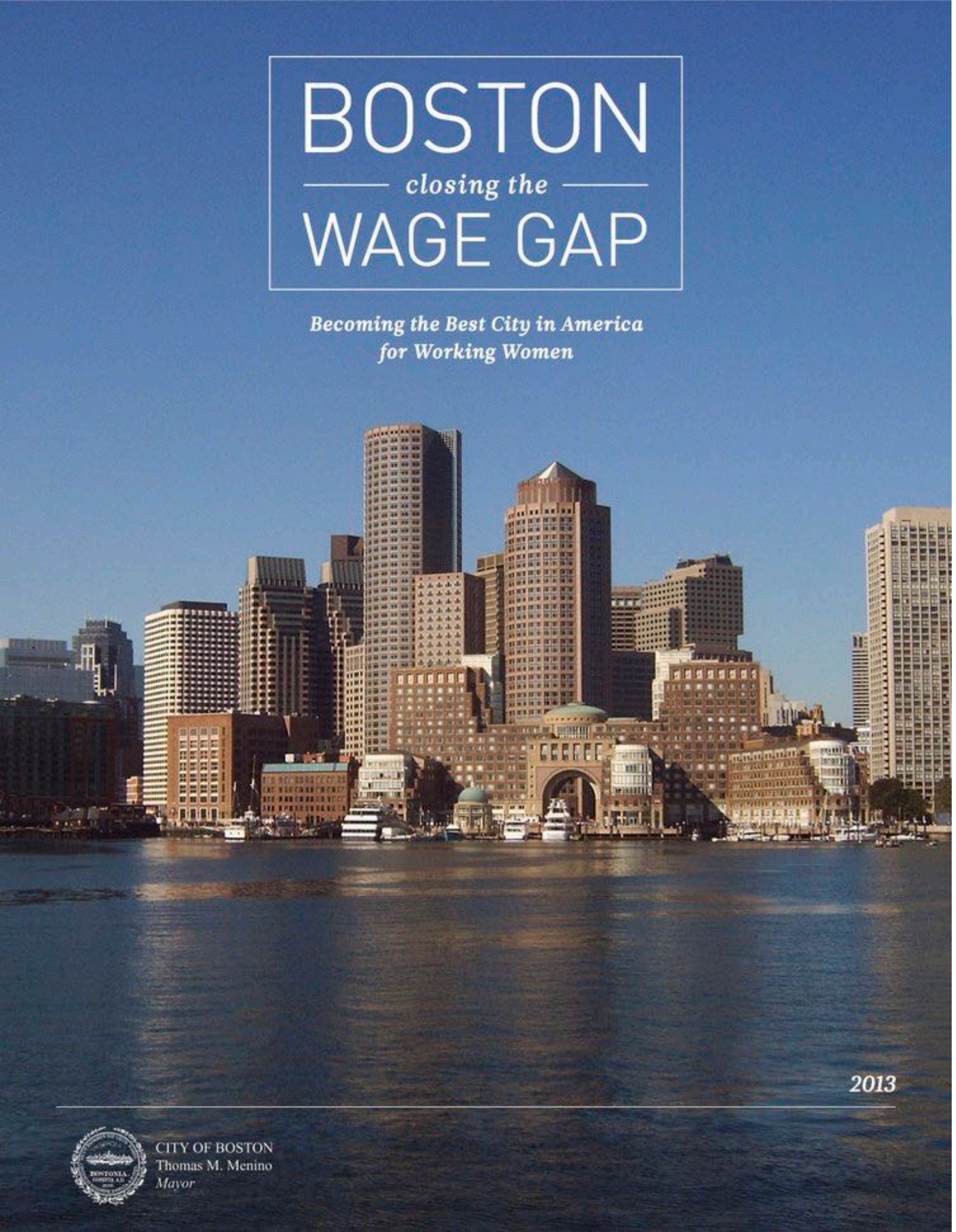
- Privacy ("learn no more than" prescribed output)
- Input independence
- Output consistency, etc..

..even if some parties cheat, collude!

Techniques for cryptographically secure computing

- Garbled circuits
- Secret sharing





100% TALENT

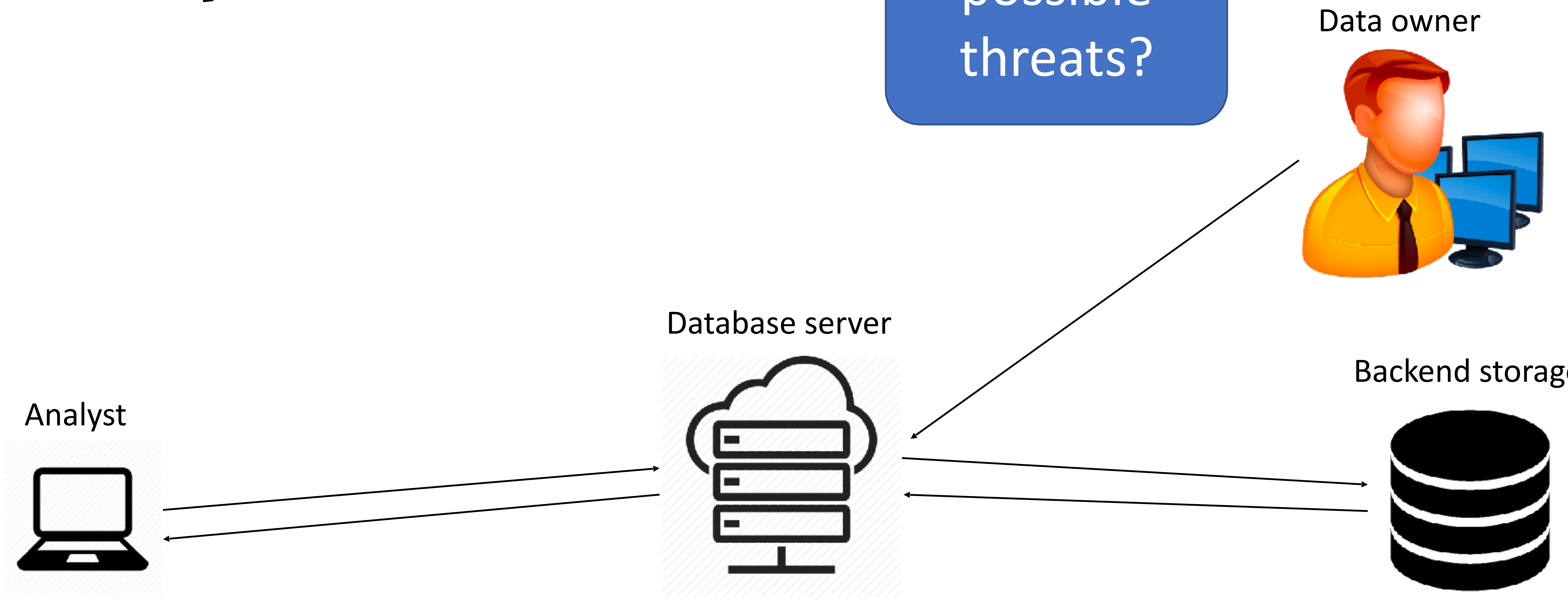
The Boston Women's Compact

SIMMONS COLLEGE BOSTON · MASSACHUSETTS
CITY OF BOSTON Office of the Mayor Martin J. Walsh
STATE STREET
EMC²
Raytheon
build smart
VERTEX
STAPLES MAKE more HAPPEN
MassMutual FINANCIAL GROUP
SUFFOLK
aim Associated Industries of Massachusetts
Putnam INVESTMENTS
Care.com
Eastern Bank
FINAGLE A BAGEL
nationalgrid
MASSACHUSETTS TECHNOLOGY COLLABORATIVE
EVERSOURCE ENERGY
Abt Associates BOLD THINKERS DRIVING REAL-WORLD IMPACT
BBK WORLDWIDE
HILL HOLLIDAY
BJ's
MASSART MASSACHUSETTS COLLEGE OF ART AND DESIGN
WENTWORTH Institute of Technology
PARTNERS. HEALTHCARE FOUNDED BY BRIGHAM AND WOMEN'S HOSPITAL AND MASSACHUSETTS GENERAL HOSPITAL
Charlestown nursery school
Tech Networks of Boston We're better together.
TUFTS Health Plan
WGA WILLIAM GALLAGHER ASSOCIATES
WHEELOCK COLLEGE
tBf The Boston Foundation
Top it off

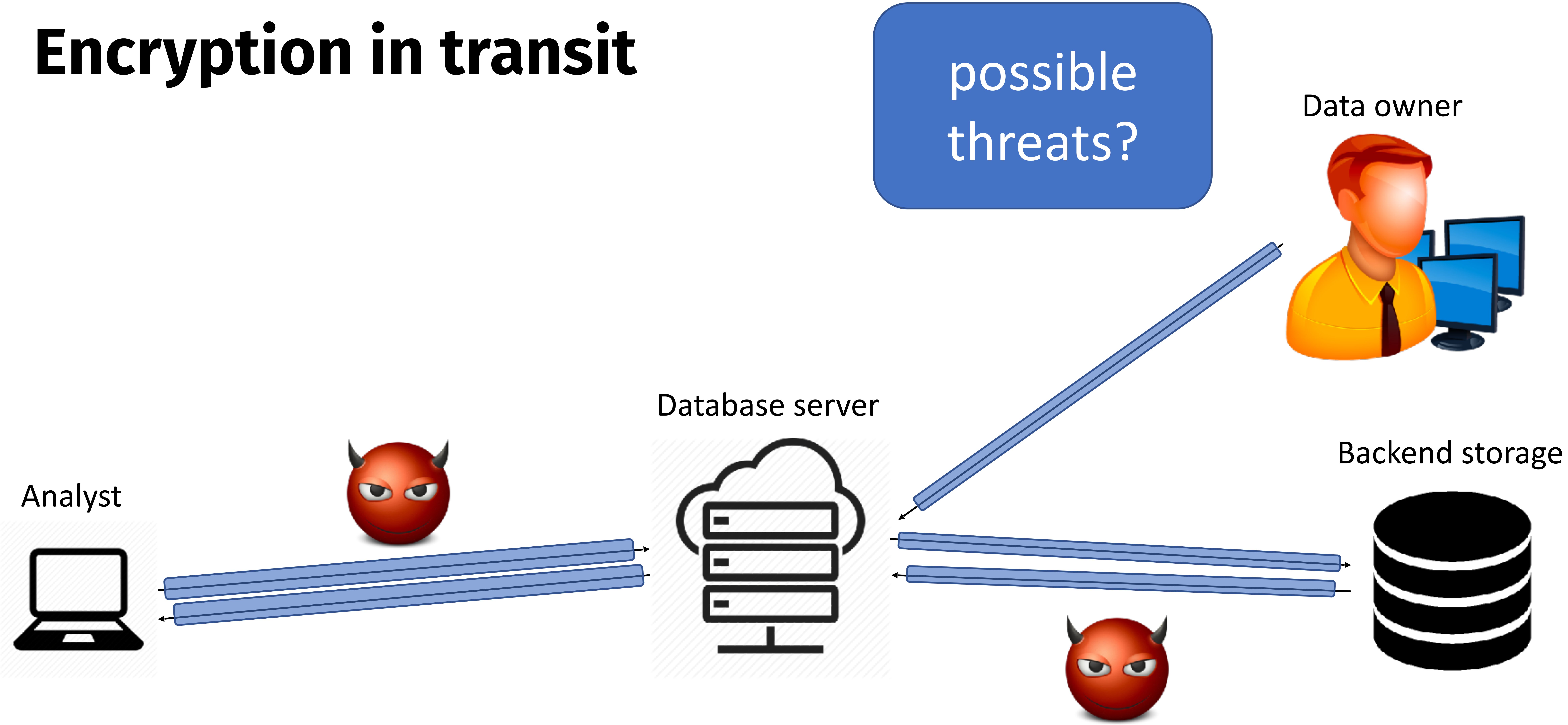
Cryptographically protected data structures

Let’s protect a database

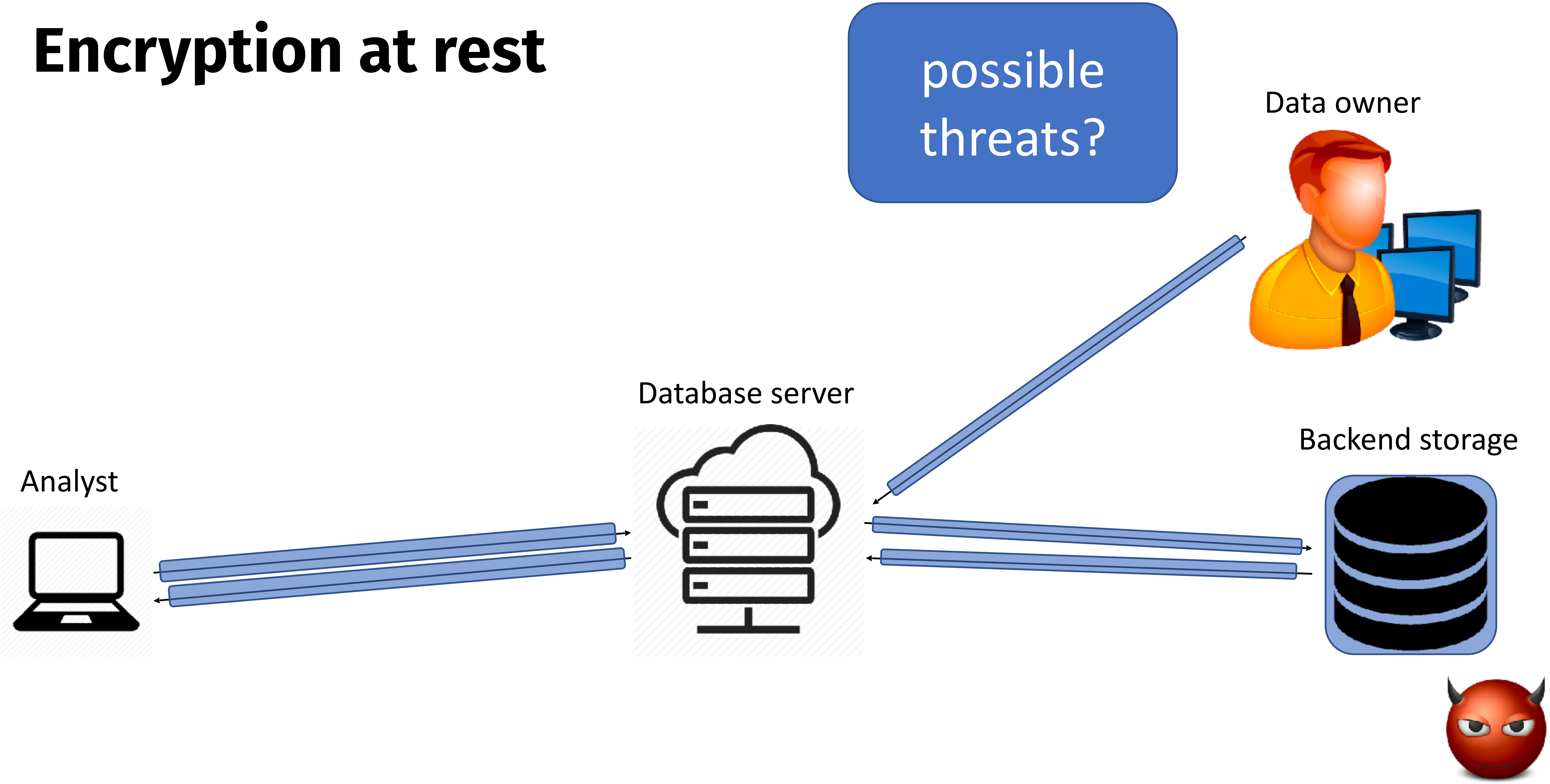
possible threats?



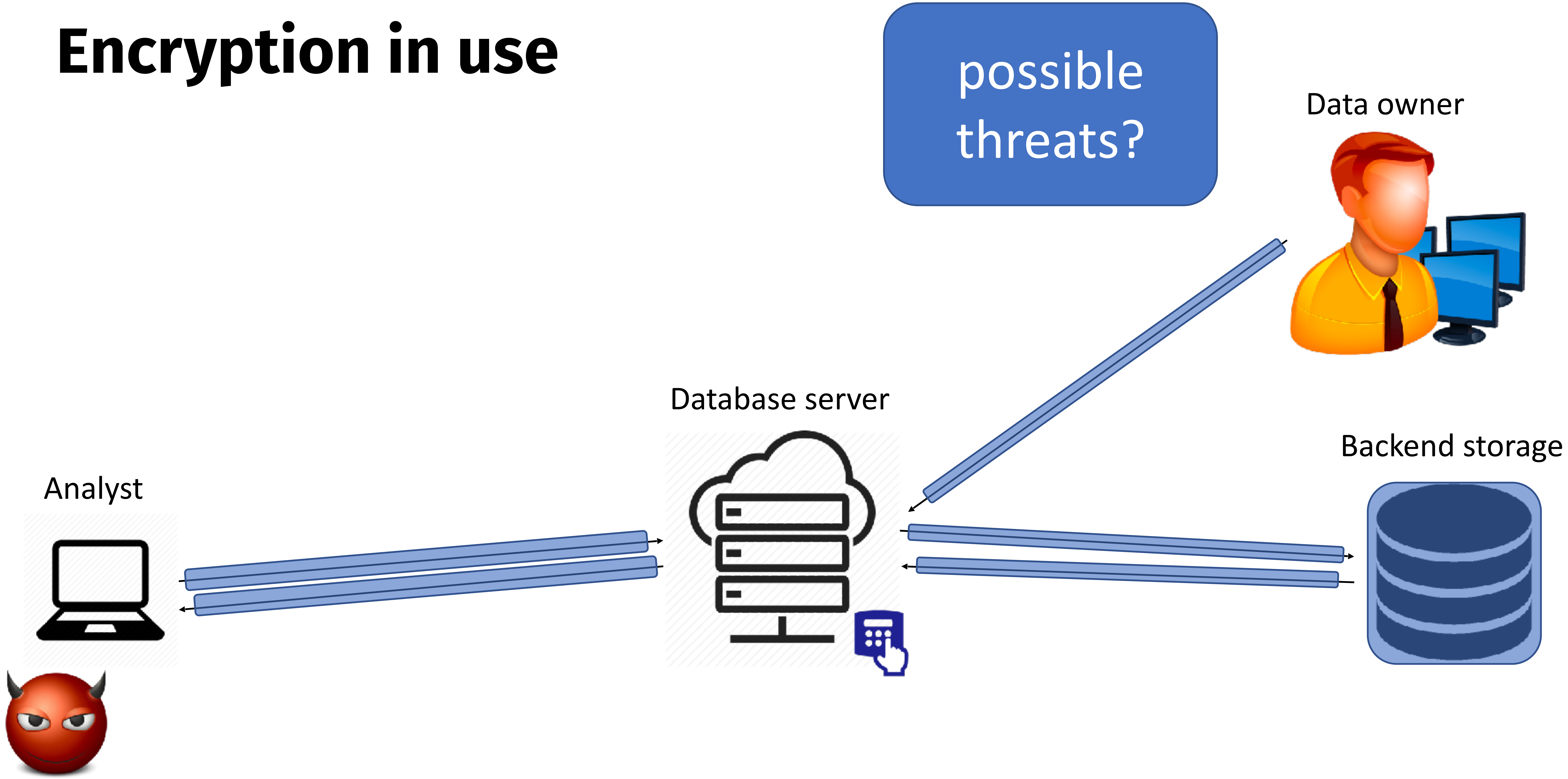
Encryption in transit



Encryption at rest

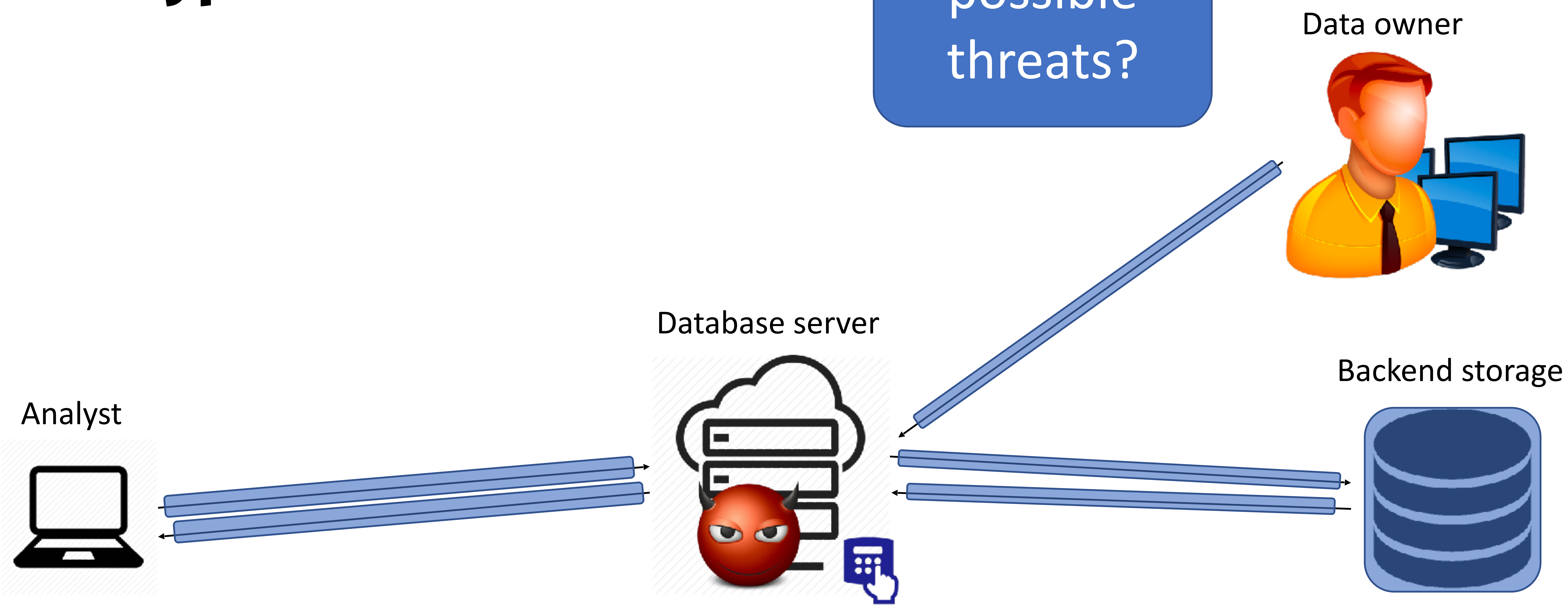


Encryption in use



Encryption in use

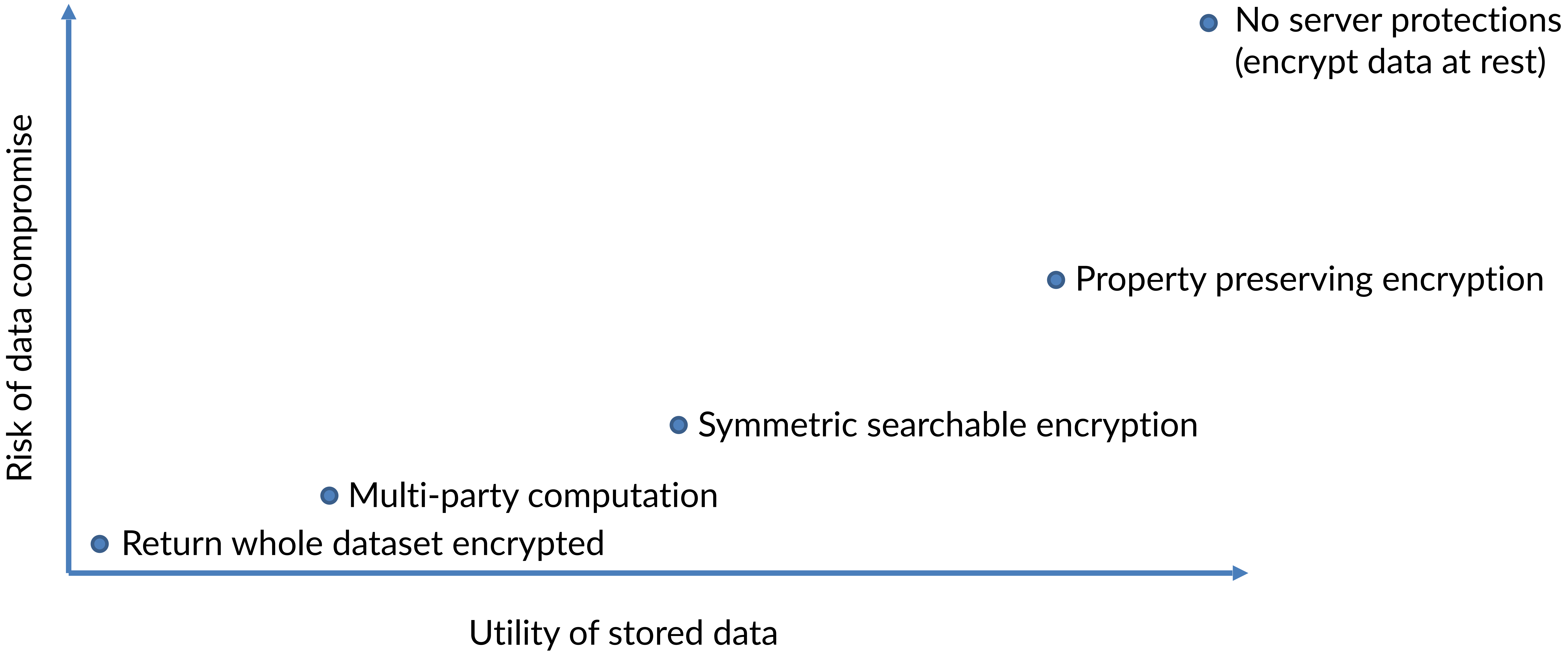
possible threats?



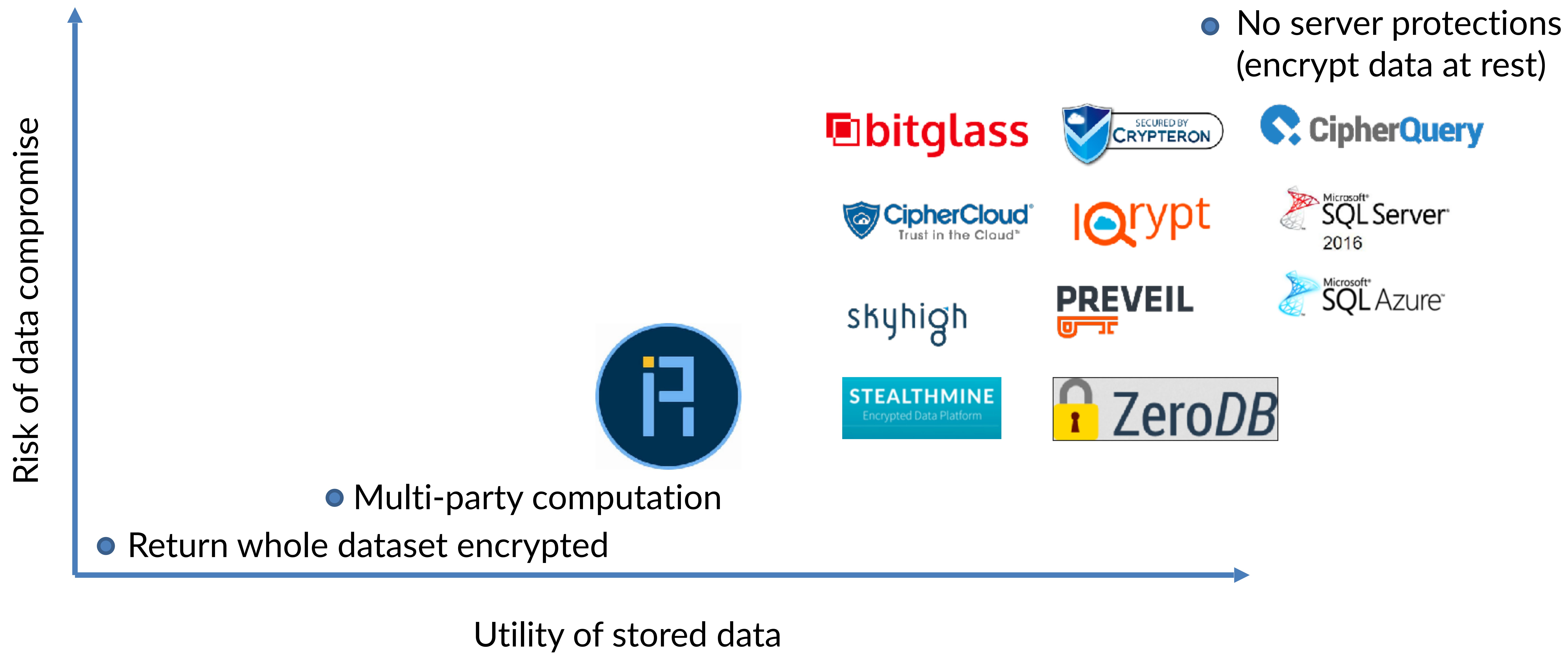
Desired goal: “garbled indexes” that permit the server to search directly over encrypted records

- Server shouldn’t see either data or queries
- Server might observe access patterns though

Cryptographically protected database search

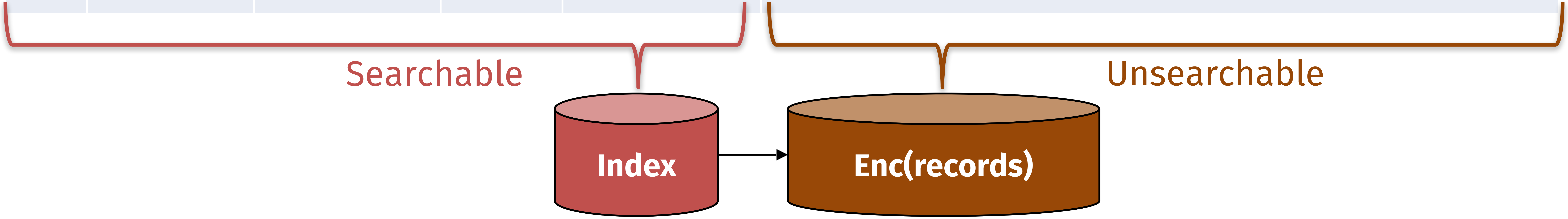


State of the art



Abstract view of a single-table database

id	fname	lname	Age	Income	Photo
1	Alice	Jones	20	71,000	<alice.jpg>
2	Bob	Jones	25	58,000	<bob.jpg>
3	Charlie	Smith	50	62,000	<charlie.jpg>
4	David	Williams	55	75,000	<david.jpg>



Small data structure: map searchable terms to associated record ids

Large file store: standard authenticated encryption applied to each record

1. Property Preserving Encryption (PPE)

- Apply transformation that preserves relevant features
- Insert into a legacy database for indexing & searching

id	fname	lname	Age	Income
1	Alice	Jones	20	71,000
2	Bob	Jones	25	58,000
3	Charlie	Smith	50	62,000
4	David	Williams	55	75,000

id	fname	lname	Age	Income
1	qlap1	Lf4Pz	cnr	g^{71} r^{90}
2	7fBwo	Lf4Pz	duo	g^{58} r^{84}
3	AKx0k	sw2AD	syv	g^{62} r^{22}
4	CK6ZD	6lVTH	tng	g^{75} r^{38}

Operation:	**DET (=)**	**OPE (<)**	**HOM (+, x)**
Method:	Choose Enc function at random	Choose random *monotonic* function	Public-key crypto
Drawback:	Cloud sees equality patterns	Cloud sees < and ~distances	Slow

1. Property Preserving Encryption (PPE)

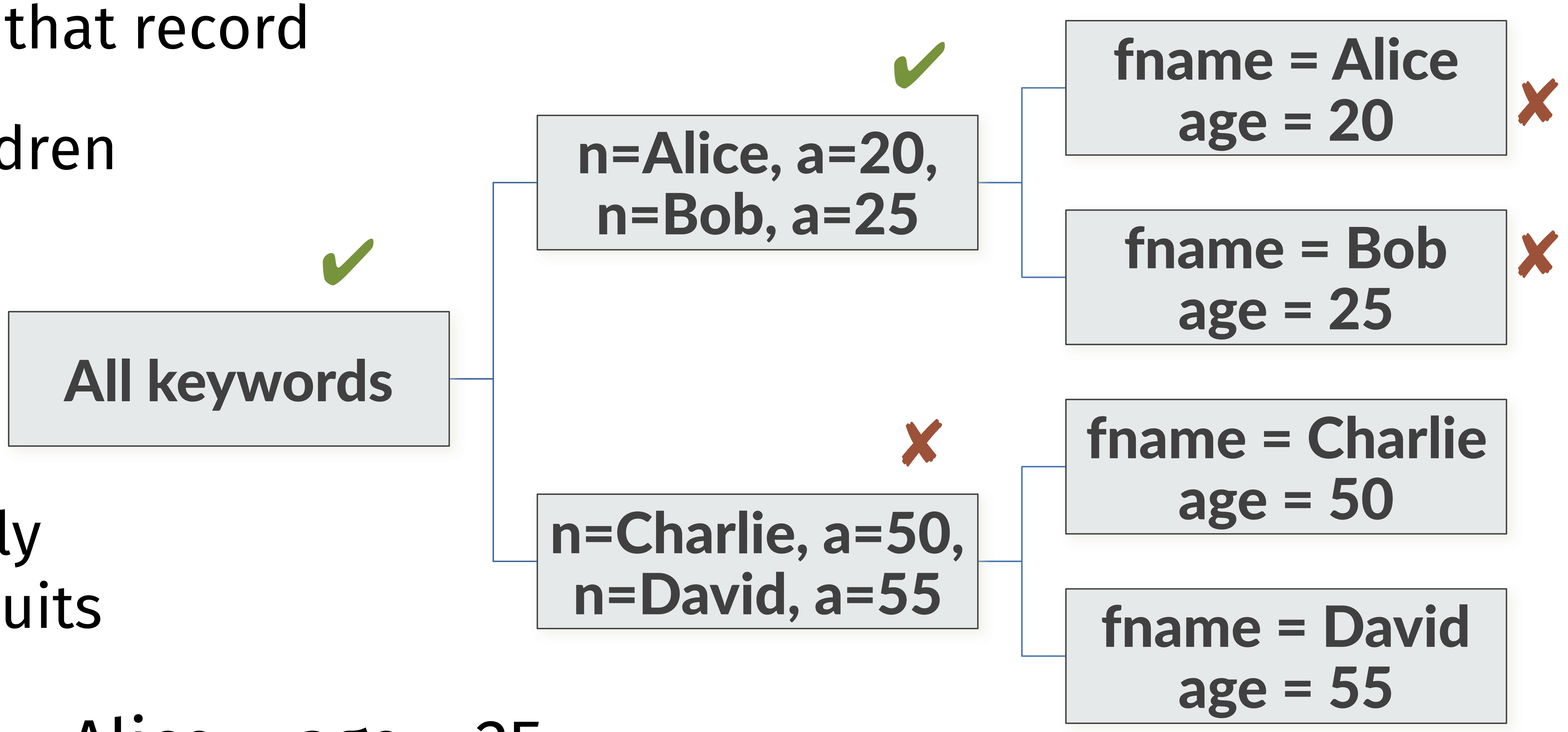
- Fast & legacy compliant
- Supported by a database near you!
  - Google: Encrypted BigQuery
  - Microsoft: SQL Server 2016, Azure SQL Database
  - Startups: Bitglass, Ciphercloud, CipherQuery, Crypteron, IQrypt, Kryptonostic, PreVeil, Skyhigh, ZeroDB
- Weakness: leakage provided to cloud is strong enough to permit data & query reconstruction attacks

2. Searchable Symmetric Encryption (SSE)

- Privacy: reveals or “leaks” less information to the database server
- Query expressivity: large subset of SQL
- Scale: tested on databases with 100m records
- Performance: within 5x of MariaDB

SSE example (Blind Seer)

- Consider a tree in which each node stores a set
  - Leaves: set of keywords in that record
  - Other nodes: union of children
- Roles
  - Data owner makes tree
  - Cloud server & client jointly traverse using garbled circuits



- Consider the query name = Alice $\wedge$ age = 25
- Imperfect security: tree search pattern reveals info about data

All database types can be protected with crypto

Structure	Query basis	Examples	Strengths
Relational	Set mathematics	MySQL, Oracle, Postgres	Transactional support Standardized SQL interface
Key-value	Associative arrays	BigTable, Hbase, Accumulo	High insert rates Flexible data models
Graph	Linear algebra	IBM System G, GraphBLAS, Neo4j	Natural data representation Amenable to graph algs
Array	Linear algebra	SciDB, TileDB	Transactional support High performance Specialized to scientific computing
NewSQL	Set mathematics	Google Spanner, MemSQL, Spark	Transactional support High insert rate

Most SQL query types are supported by SSE

Type	Description	SQL Example
Short string	Equality	`fname = ‘Homer’`
	Wildcard	`notes LIKE ‘%oo %oo!’`
	Substring	`notes LIKE ‘%mmm%’`
Numeric	Inequality	`age ⩾ 30`
	Ranges	`age BETWEEN 38 and 42`
Long text	Free-text keyword	`CONTAINED_IN(notes, ‘donut’)`
	Stemming	`CONTAINS_STEM(notes, ‘work’)`
Boolean	Conjunction/disjunction	`lname = ‘Simpson’ AND city = ‘Springfield’`
	Threshold	`M_OF_N(2, 3, income > 40000, citizenship = ‘Yes_Born_In_US’, marital_status = ‘Married’)`
	Ranking	`M_OF_N(2, 3, income > 40000, citizenship = ‘Yes_Born_In_US’, marital_status = ‘Married’) ORDER BY RANK`

Information revealed by SSE

- Protected search schemes reveal or leak some information about the query, data set, and result set to each party.
  1. Structure: size of an object, e.g. length of a string or cardinality of a set
  2. Identifiers: pointers to objects that persist across multiple accesses
  3. Equality or Order of values
- Some schemes leak:
  1. At Initialization on entire DB
  2. At Query on relevant records

Weekly reading: the Pareto Frontier of protected databases

			Threats			S leakage		Scale			Crypto			Network		
Query type	Scheme (References)	Approach	# of parties	Adversarial Q	Adversarial S	Init	Query	Updatable?	Implemented?	Scale tested	Crypto type	Insert: # ops	Query: # ops	# round trips	Data sent	Unique feature
Equality	Arx-EQ [14]	Legacy	2	—	◐	○	◔	●	✔	◐	●	●	●	●	●	legacy compliant
	Kamara-Papamanthou [106]	Custom	2	—	◐	○	◔	●	—	—	●	○	○	●	●	parallelizable
	Blind Storage [100]	Custom	2	—	◐	○	◔	●	✔	◐	●	◐	●	◐	●	low S work
	Sophos (Σοφος) [101]	Custom	2	—	◐	○	◔	●	✔	◐	○	◐	●	●	●	**Refresh w/ Insert**
	Stefanov et al [107]	Custom	2	—	◐	○	◔	●	✔	◐	●	○	○	●	●	**Refresh w/ Insert**
	vORAM+HIRB [120]	Obliv	2	—	◐	○	○	●	✔	●	●	○	○	○	◔	history independ.
	TWORAM [121]	Obliv	2	—	◐	○	○	●	—	—	◐	○	○	◐	◔	const round
	3PC-ORAM [124]	Obliv	3	◐	◐	○	○	●	✔	◔	●	○	○	○	◔	dual S
Boolean	DET [15], [92]	Legacy	2	—	◐	◕	◕	●	✔	●	●	◐	◐	●	●	supports JOINs
	BLIND SEER [16], [17]	Custom	3	●	●	○	◐	◐	✔	●	◐	◐	○	○	◔	hide field, r_i's
	OSPIR-OXT [18]–[21], [104]	Custom	3	●	◐	○	◐	●	✔	●	◐	◐	◐	◕	●	excels w/ small r_1
	Kamara-Moataz [102]	Custom	2	—	◐	○	◐	○	—	—	◐	◐	○	●	◔	relational SPC
Range	OPE [93]–[95]	Legacy	2	—	◐	●	●	●	✔	●	●	●	●	●	●	leak some content
	Mutable OPE [97]	Legacy	2	—	◐	●	●	●	✔	◐	●	○	○	○	◐	interactive
	Partial OPE [111]	Custom	2	—	◐	○	●	●	✔	◐	●	●	●	◐	●	fast insertions
	Arx-RANGE [110]	Custom	2	—	◐	○	◐	●	✔	◐	◐	○	○	●	○	non-interactive
	SisoSPIR [22]	Obliv	3	◐	◐	○	○	○	✔	●	●	●	●	○	◔	split, non-colluding S
Other	GraphEnc_1 [116]	Custom	2	—	◐	◐	◔	○	✔	◐	●	●	●	●	◔	approx. graph dist.
	GraphEnc_3 [116]	Custom	2	—	◐	◐	◐	○	✔	◐	○	●	●	●	●	approx. graph dist.
	Chase-Shen [109], [126]	Custom	2	—	●	○	◐	○	✔	◔	●	●	●	◐	●	substring search
	Moataz-Blass [123]	Obliv	2	—	◐	○	○	●	✔	●	●	○	○	○	◔	substring search

Weekly reading: inference attacks from leaked information

Attacker goal	Required S leakage: Init	Required S leakage: Query	Required attack conditions: Ability to inject data	Required attack conditions: Prior knowledge	Attack efficacy: Runtime	Attack efficacy: Sensitivity to prior knowledge	Attack efficacy: Keyword universe tested	Attack name
Query Recovery	○	○	—	◔	●	?	○	Communication Volume Attack [125]
	○	◔	✔	○	○	○	○	Binary Search Attack [127]
	○	◔	—	◔	●	?	○	Access Pattern Attack [125]
	○	◔	—	◕	◐	●	●	Partially Known Documents [128]
	○	◔	✔	◕	◐	○	●	Hierarchical-Search Attack [127]
	○	◔	—	●	◐	●	●	Count Attack [128]
Data Recovery	○	◔	—	◐	●	●	◐	Graph Matching Attack [129]
	◕	—	—	◐	○	?	○	Frequency Analysis [130]
	◕	—	✔	◐	○	?	●	Active Attacks [128]
	◕	—	—	◕	○	?	●	Known Document Attacks [128]
	●	—	—	◐	○	○	●	Non-Crossing Attack [131]

TABLE III

Summary of current leakage inference attacks against protected search base queries. S is the server and the assumed attacker for all attacks listed. S leakage symbols have the same meaning as in Table II. Each attack is relevant to schemes in Table II with at least the S leakage specified in this table. Some attacks require the attacker to be able to inject data by having the provider insert it into the database. Legends for the rest of the columns follow. In all columns except "Keyword universe tested," bubbles that are more filled in represent properties that are better for the scheme and worse for the attacker.

Prior knowledge
- ●– Contents of full dataset
- ◕– Contents of a subset of dataset
- ◐– Distributional knowledge of dataset
- ◔– Distributional knowledge of queries
- ○– Keyword universe

Runtime (in # of keywords)
- ●– More than quadratic
- ◐– Quadratic
- ○– Linear

Sensitivity to prior knowledge
- ●– High
- ○– Low
- ? – Untested

Keyword universe tested
- ●– > 1000
- ◐– 500 to 1000
- ○– < 500

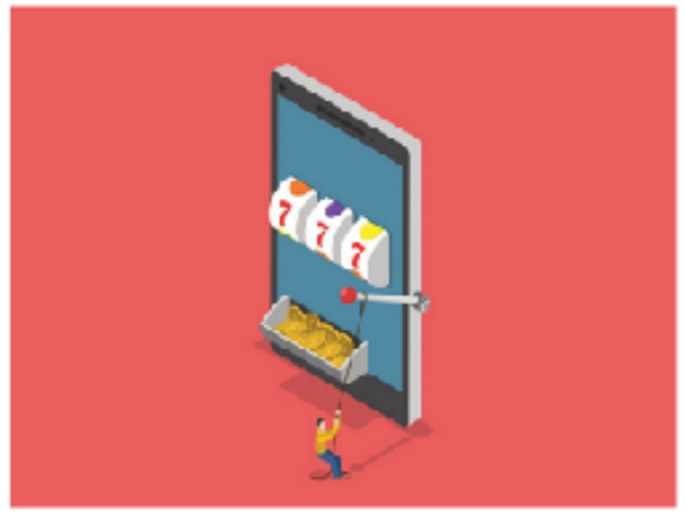
Random number generation

Randomness ⇒ Unpredictability ⇒ Secrecy

Effects of bad randomness

- Lottery fraud

RUSSIANS ENGINEER A BRILLIANT SLOT MACHINE CHEAT—AND CASINOS HAVE NO FIX

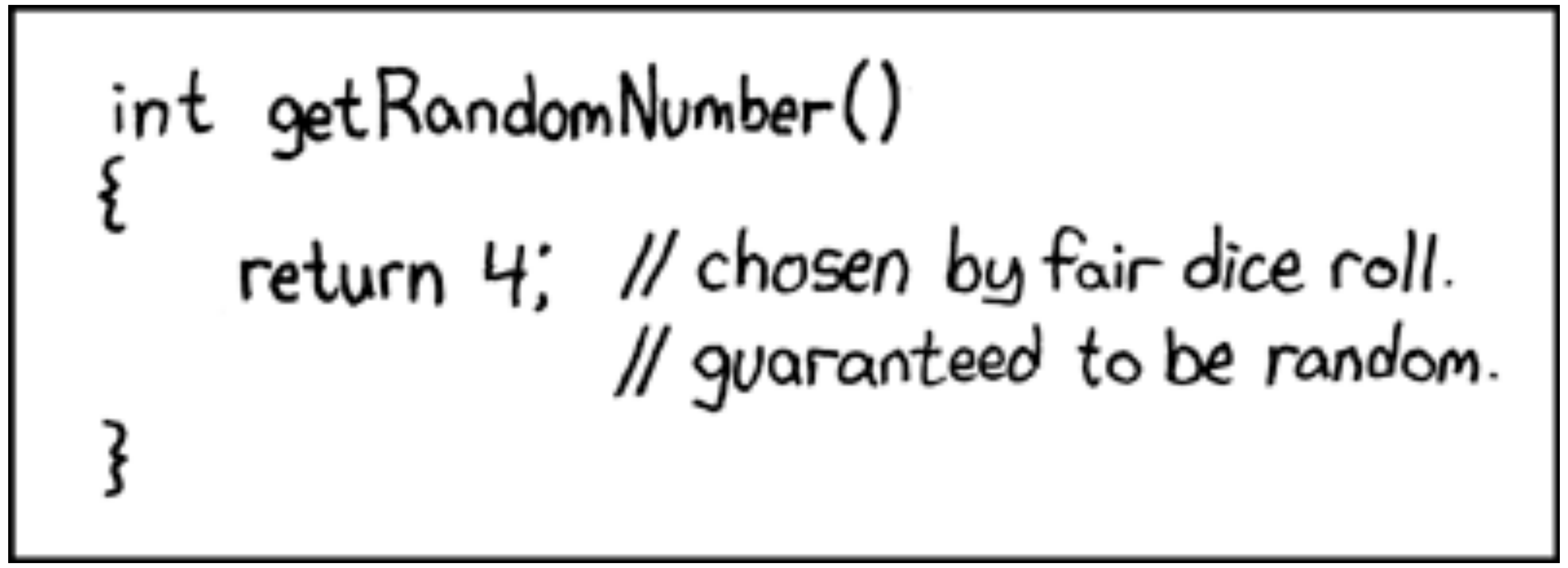


A forensic examination found that the generator had code that was installed after the machine had been audited by a security firm that directed the generator not to produce random numbers on three particular days of the year if two other conditions were met. Numbers on those days would be drawn by an algorithm that Tipton could predict, Iowa Division of Criminal Investigation agent Don Smith wrote in an affidavit.

All six prizes linked to Tipton were drawn on either Nov. 23 or Dec. 29 between 2005 and 2011.

Investigators were able to recreate the draws and produce "the very same 'winning numbers' from the program that was supposed to produce random numbers," Smith wrote.

- Weak TLS keys on Debian computers in 2006-2008
- Weak RSA keys
- ...and more



Source: xkcd.com/221

Bad randomness in Debian

Bug forum discussion, 2003

I'm using Valgrind to debug a program that uses the OpenSSL libraries, and got warnings about uninitialized data in the function RSA_padding_add_PKCS1_type_2(), on the line with "} while (*p == '\0');" (line 171 in version 0.9.7a). The following patch ensures that the data is always modified, something that the bytes() method obviously fails to do.

```
--- rand_lib.c Thu Jan 30 2003
+++ rand_lib.c Wed Feb 26 2003
@@ -154,6 +154,7 @@
int RAND_bytes(unsigned char *buf, int num)
{
[new code here]
```

Debian security advisory, 2008

Luciano Bello discovered that the random number generator in Debian's openssl package is predictable. This is caused by an incorrect Debian-specific change to the openssl package. As a result, cryptographic key material may be guessable. ...

It is strongly recommended that all cryptographic key material which has been generated by OpenSSL versions starting with 0.9.8c-1 on Debian systems is recreated from scratch. Furthermore, all DSA keys ever used on affected Debian systems for signing or authentication purposes should be considered compromised.

Bad randomness in RSA key generation

Ron was wrong, Whit is right

Arjen K. Lenstra[1], James P. Hughes[2],
Maxime Augier[1], Joppe W. Bos[1], Thorsten Kleinjung[1], and Christophe Wachter[1]

[1] EPFL IC LACAL, Station 14, CH-1015 Lausanne, Switzerland
[2] Self, Palo Alto, CA, USA

Abstract. We performed a sanity check of public keys collected on the web. Our main goal was to test the validity of the assumption that different random choices are made each time keys are generated. We found that the vast majority of public keys work as intended. A more disconcerting finding is that two out of every one thousand RSA moduli that we collected offer no security. Our conclusion is that the validity of the assumption is questionable and that generating keys in the real world for "multiple-secrets" cryptosystems such as RSA is significantly riskier than for "single-secret" ones such as ElGamal or (EC)DSA which are based on Diffie-Hellman.
Keywords: Sanity check, RSA, 99.8% security, ElGamal, DSA, ECDSA, (batch) factoring, discrete logarithm, Euclidean algorithm, seeding random number generators, K_9.

How you obtain randomness: /dev/urandom

```
Computer:~ Mayank$ cat /dev/urandom | head -c 96 | xxd
00000000: 687d 6207 2bb2 2341 a26b f6f6 80a7 6a51  h}b.+.#A.k....jQ
00000010: 7445 1cd6 d65b feb4 05ff f917 9c29 9c20  tE...[.......).
00000020: a439 48bd ecc8 d06f 246e 76d1 5c68 3184  .9H....o$nv.\h1.
00000030: a075 7722 0b31 8c02 dcab dfc0 54dc ca0c  .uw".1......T...
00000040: cc3b f811 6d50 73f3 4bf1 f9b0 685c ad8b  .;..mPs.K...h\..
00000050: 7d26 c6c5 3f05 4cbc 1e3c 0c4d abef 5f66  }&..?.L..<.M.._f
```

How a computer generates randomness

Statistical tests to see
if entropy is “sufficient”

Step 1: Harvest
Reliably produce a “pool” of bits with true *entropy*

Step 2: Extract
Produce ~128 nearly uniform bits from the pool

Step 3: Expand
Create a large sequence of pseudorandom bits

Security requirements of randomness generation

1. *Performance:* Be fast enough that people will use it
2. *Hard fail:* Only expand once the system has been adequately seeded with true entropy
3. *Resilience:* Adversary can't predict outputs, even if she can partially influence the source of true randomness

   ⇨ Use multiple sources of entropy, and combine them in a smart way
4. *Forward + backward secrecy:* Adversary cannot predict past or future PRNG outputs even if she knows the current seed and state

   ⇨ Re-seed the PRNG periodically with new truly random numbers

Step 1: Harvest
Reliably produce a "pool" of bits with true *entropy*

→

Step 2: Extract
Produce ~128 nearly uniform bits from the pool

→

Step 3: Expand
Create a large sequence of pseudorandom bits

"Fortunately, it's not hard to harvest truly unpredictable randomness by tapping the chaotic universe that surrounds a computer's orderly, deterministic world of 1s and 0s."

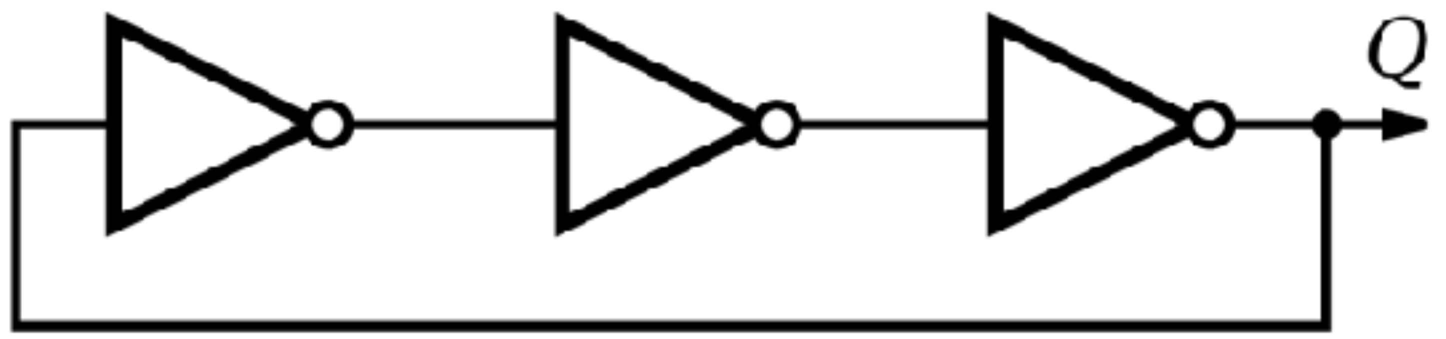
– *IEEE Spectrum*

Step 1: Sources of entropy to harvest

- Physics: EM radiation, temperature (random.org/history)

Step 1: Sources of entropy to harvest

- Physics: EM radiation, temperature (random.org/history)
- Logical gates: Clock drift, thermal noise

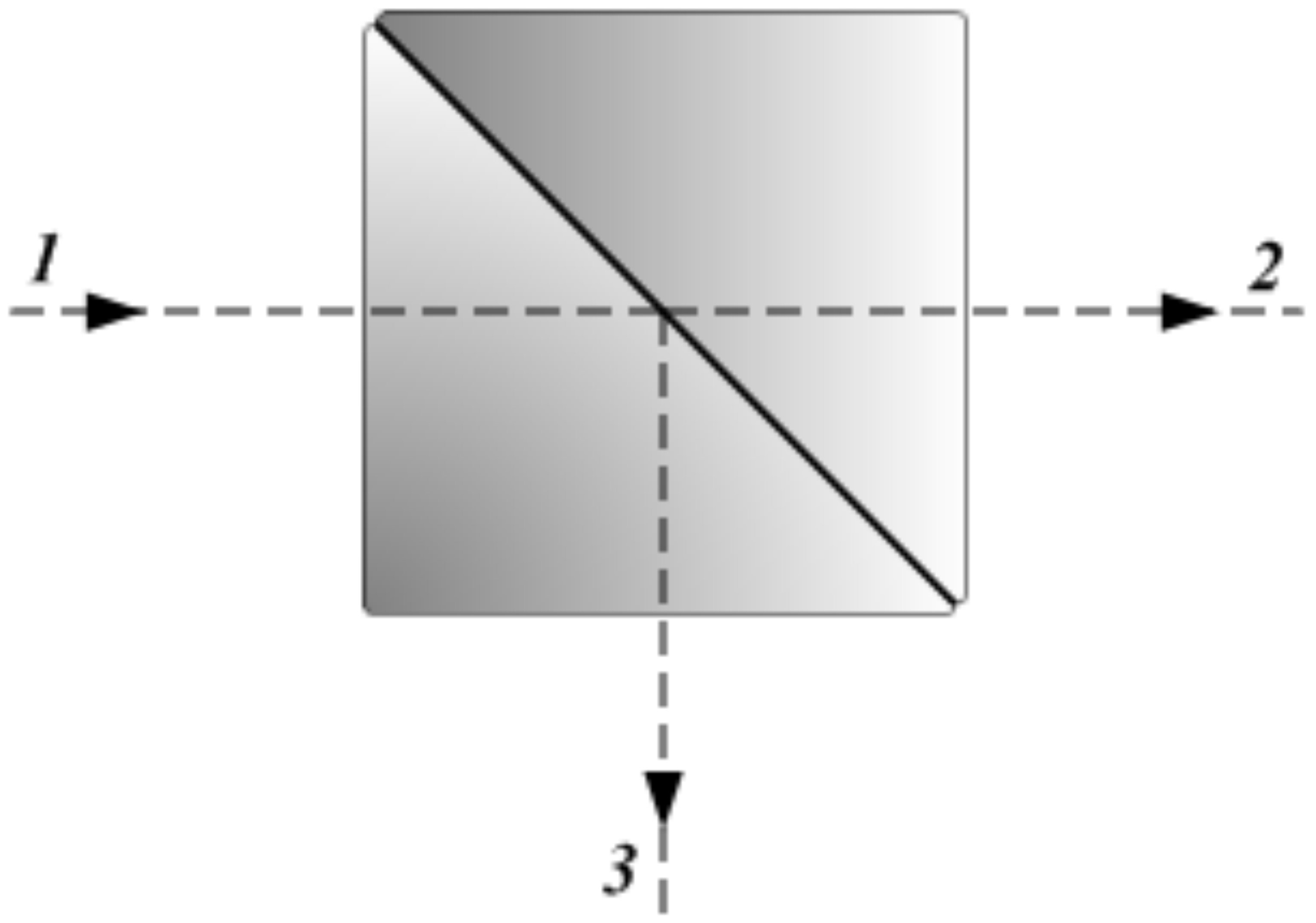


Apple's Secure Enclave

Apart from the UID and GID, all other cryptographic keys are created by the system's random number generator (RNG) using an algorithm based on CTR_DRBG. System entropy is generated from timing variations during boot, and additionally from interrupt timing once the device has booted. Keys generated inside the Secure Enclave use its true hardware random number generator based on multiple ring oscillators post processed with CTR_DRBG.

Step 1: Sources of entropy to harvest

- Physics: EM radiation, temperature (random.org/history)
- Logical gates: Clock drift, thermal noise
- Quantumness: beam splitters & polarization, tunneling, entanglement



Step 1: Sources of entropy to harvest

- Physics: EM radiation, temperature (random.org/history)
- Logical gates: Clock drift, thermal noise
- Quantumness: beam splitters & polarization, tunneling, entanglement
- Human: keystroke timings, mouse movements, hard drive seek times
- Sensors: microphone, camera, gyroscope, Bluetooth/GPS/wifi signal

Step 3: Pseudorandom expansion

Statistical tests to see
if entropy is "sufficient"

Step 1: Harvest
Reliably produce a "pool" of bits with true *entropy*

→

Step 2: Extract
Produce ~128 nearly uniform bits from the pool

→

Step 3: Expand
Create a large sequence of pseudorandom bits

Step 3: NIST standards for DRBGs

- Use counter mode as a stream cipher (CTR_DBRG)

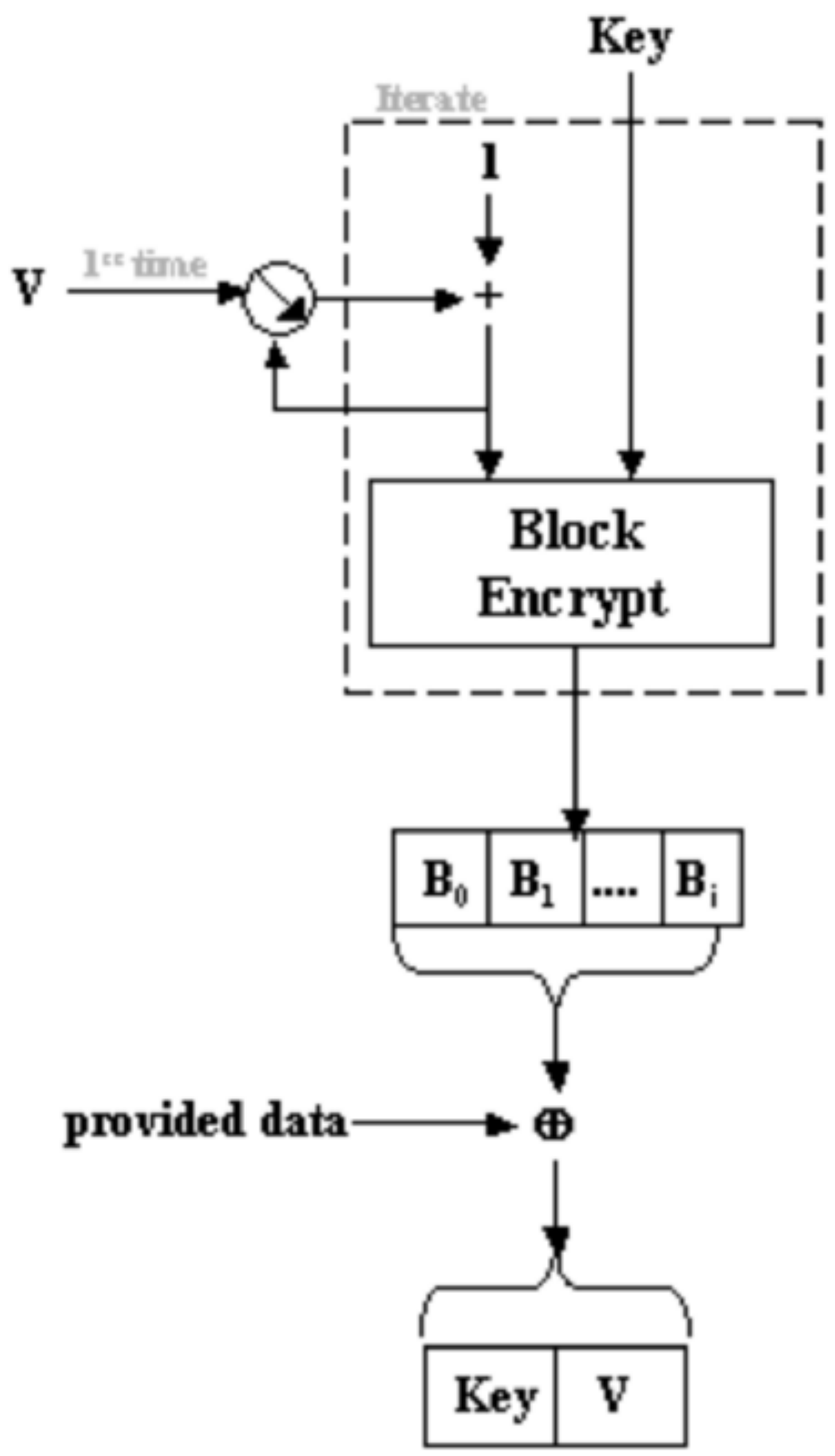
Short ***K***

+

Nonce ***N***

⇒

Synthetic, one time use key *K’*

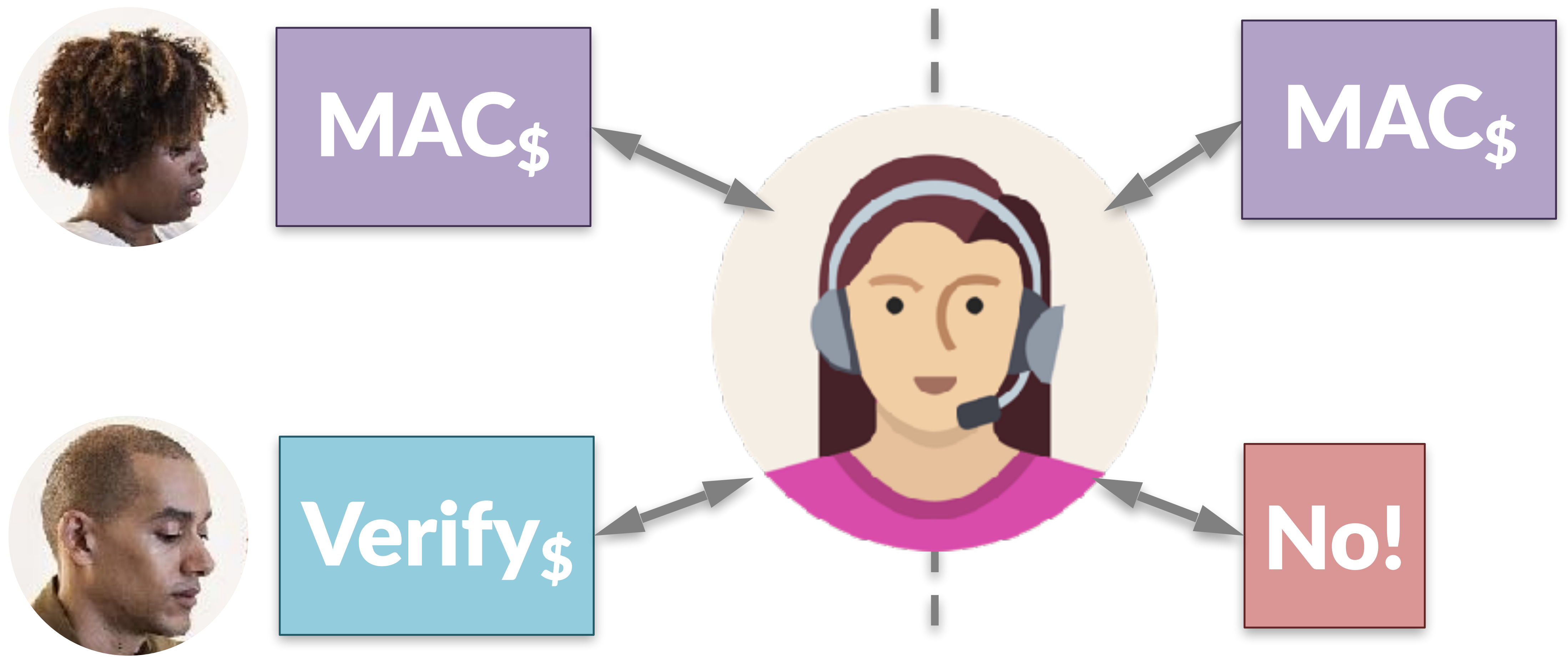


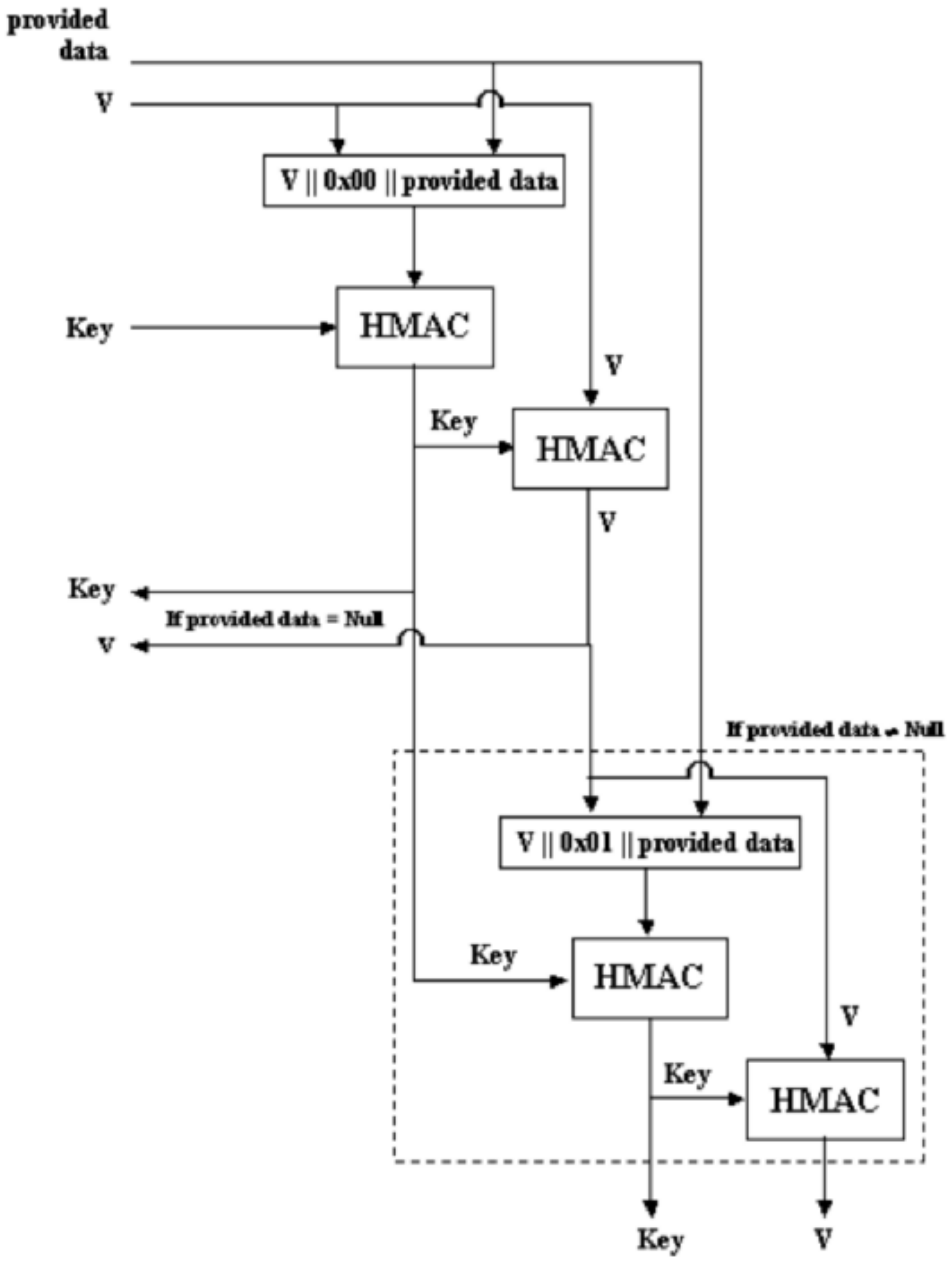
<u>Source:</u> NIST Special Publication 800-90A

Recommendation for Random Number Generation Using Deterministic Random Bit Generators

Step 3: NIST standards for DRBGs

- Use counter mode as a stream cipher (CTR_DRBG)
- A MAC is pseudorandom (HMAC_DRBG)



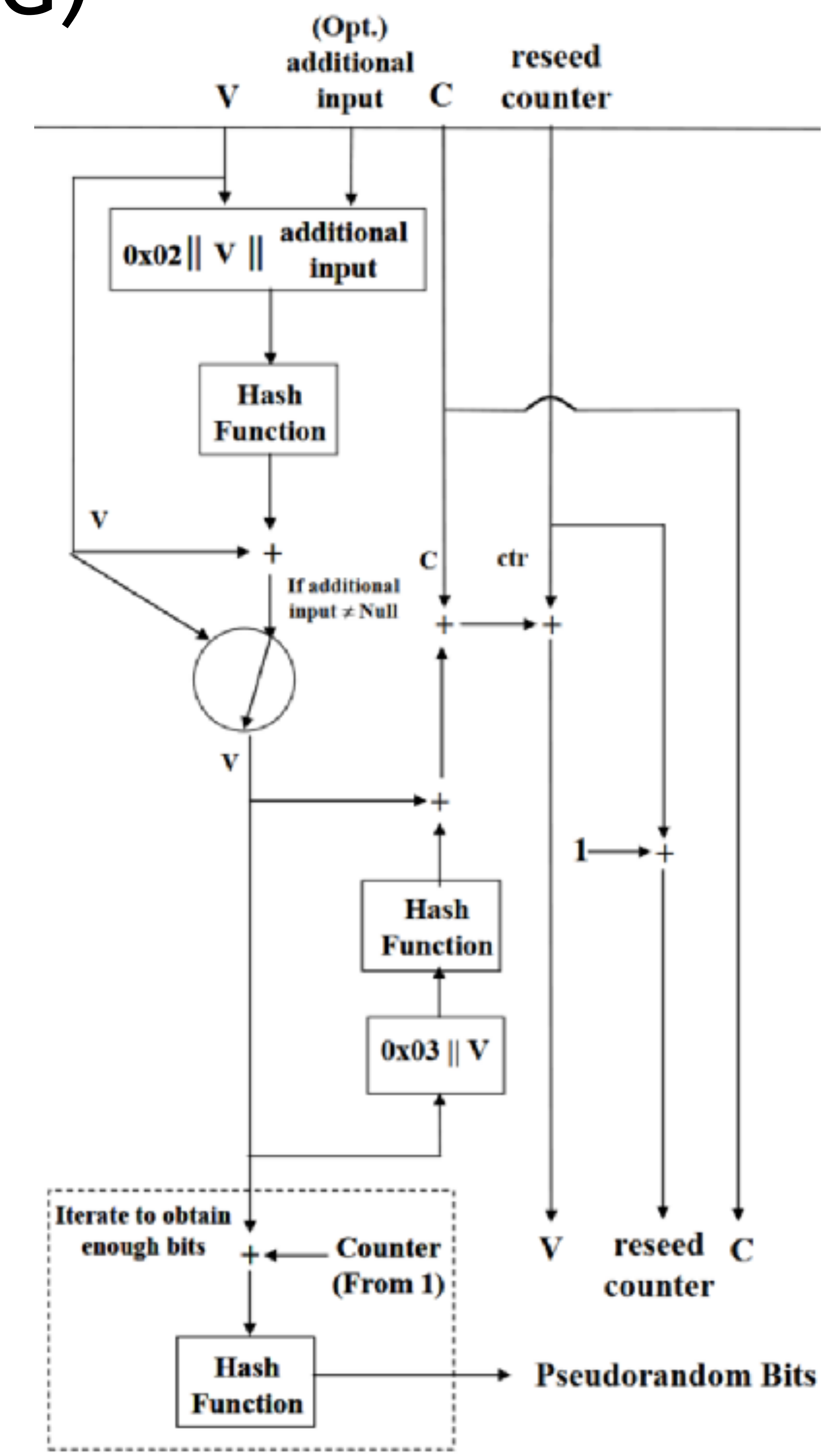


Source: NIST Special Publication 800-90A

Recommendation for Random Number Generation
Using Deterministic Random Bit Generators

Step 3: NIST standards for DRBGs

- Use counter mode as a stream cipher (CTR_DRBG)
- A MAC is pseudorandom (HMAC_DRBG)
- Use a hash function (Hash_DRBG)



Source: NIST Special Publication 800-90A

Recommendation for Random Number Generation
Using Deterministic Random Bit Generators

Step 2: Extraction of uniform-looking bits

Statistical tests to see
if entropy is “sufficient”

Step 1: Harvest
Reliably produce a “pool”
of bits with true *entropy*

Step 2: Extract
Produce ~128 nearly
uniform bits from the pool

Step 3: Expand
Create a large sequence
of pseudorandom bits

Hashing as an extractor?

- Let’s try to use a hash function H as an extractor (spoiler: it won’t work)
- Extractors operate on the principle that including more entropy sources can't hurt: H(x,y,z) is at least as good a random number as H(x,y), no matter how awful z is
- Issue: the entity that chooses z can strongly influence the resulting “random” number
    1. Generate a random z
    2. Try computing H(x,y,z)
    3. If H(x,y,z) doesn't start with bits 0000, go back to step 1
    4. Else, output this value of z
- Result: H(x,y,z) begins with four known bits of 0000, even if x and y were perfectly random
- Also, this attack is fast: it only takes 16 computations of H on average

Extraction is hard

Statistical tests to see
if entropy is "sufficient"

Step 1: Harvest
Reliably produce a "pool" of bits with true *entropy*

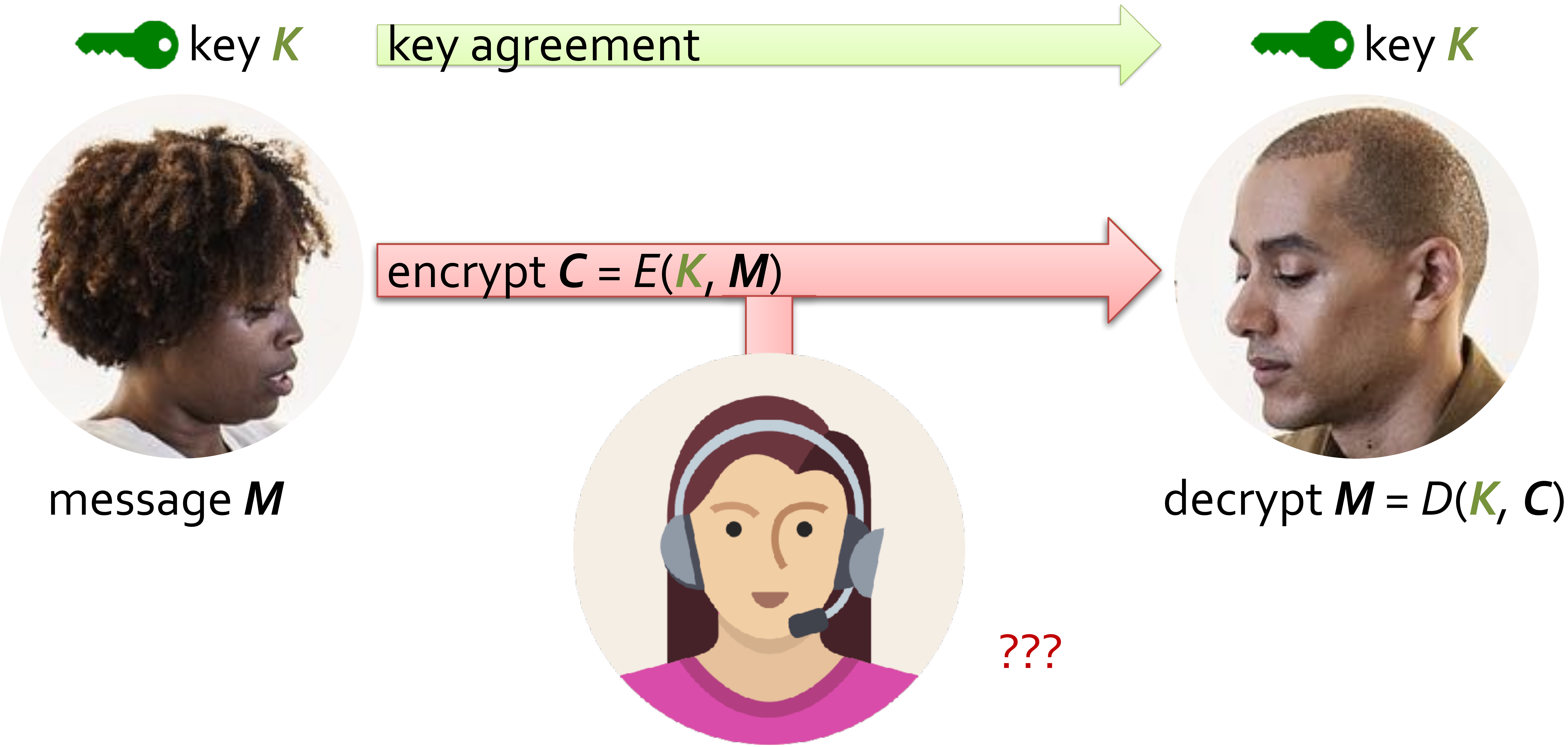
→

Step 2: Extract
Produce ~128 nearly uniform bits from the pool

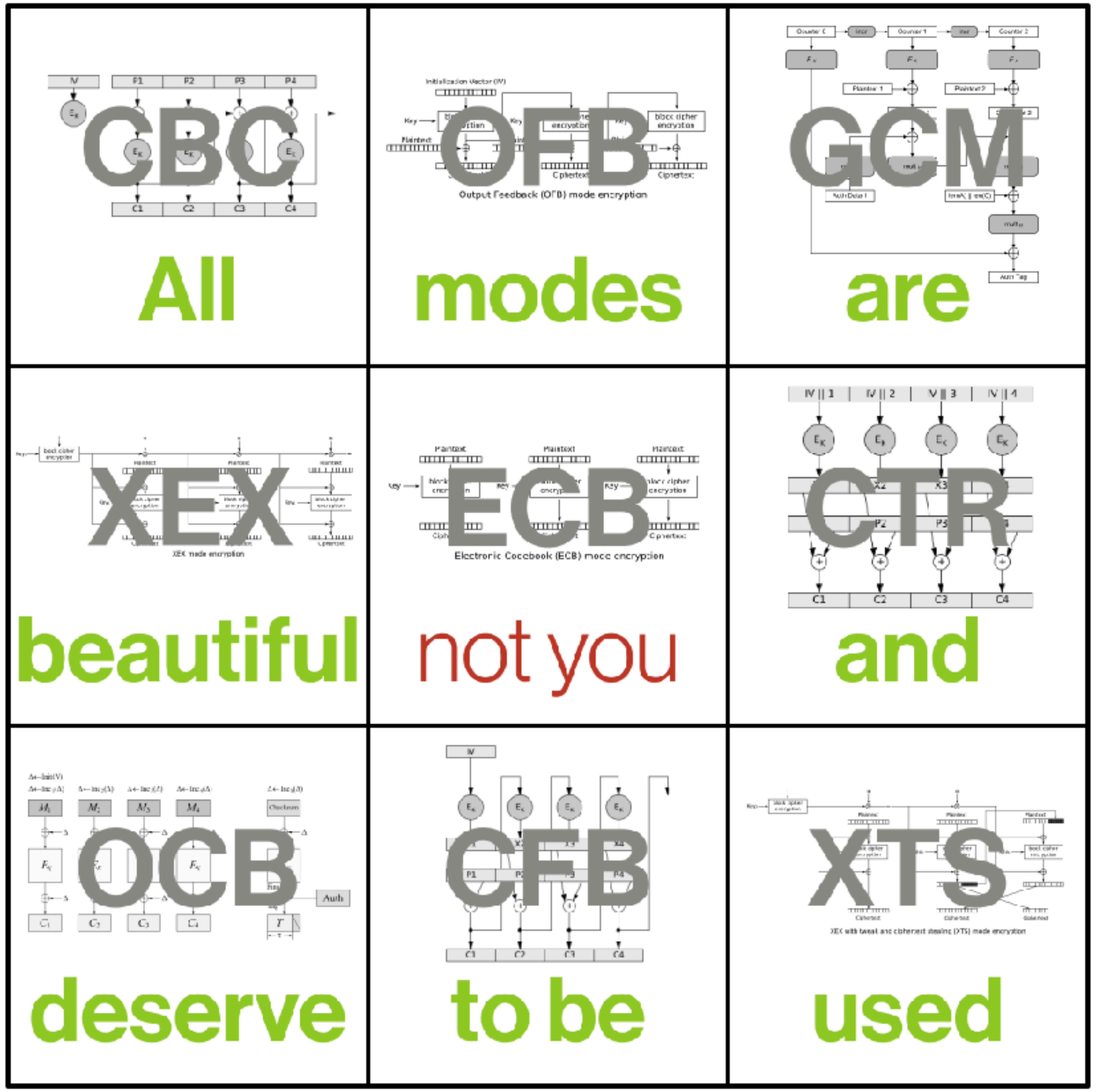
→

Step 3: Expand
Create a large sequence of pseudorandom bits

Goal: Secure communication in presence of adversary



Modes of operation for authentication and/or encryption

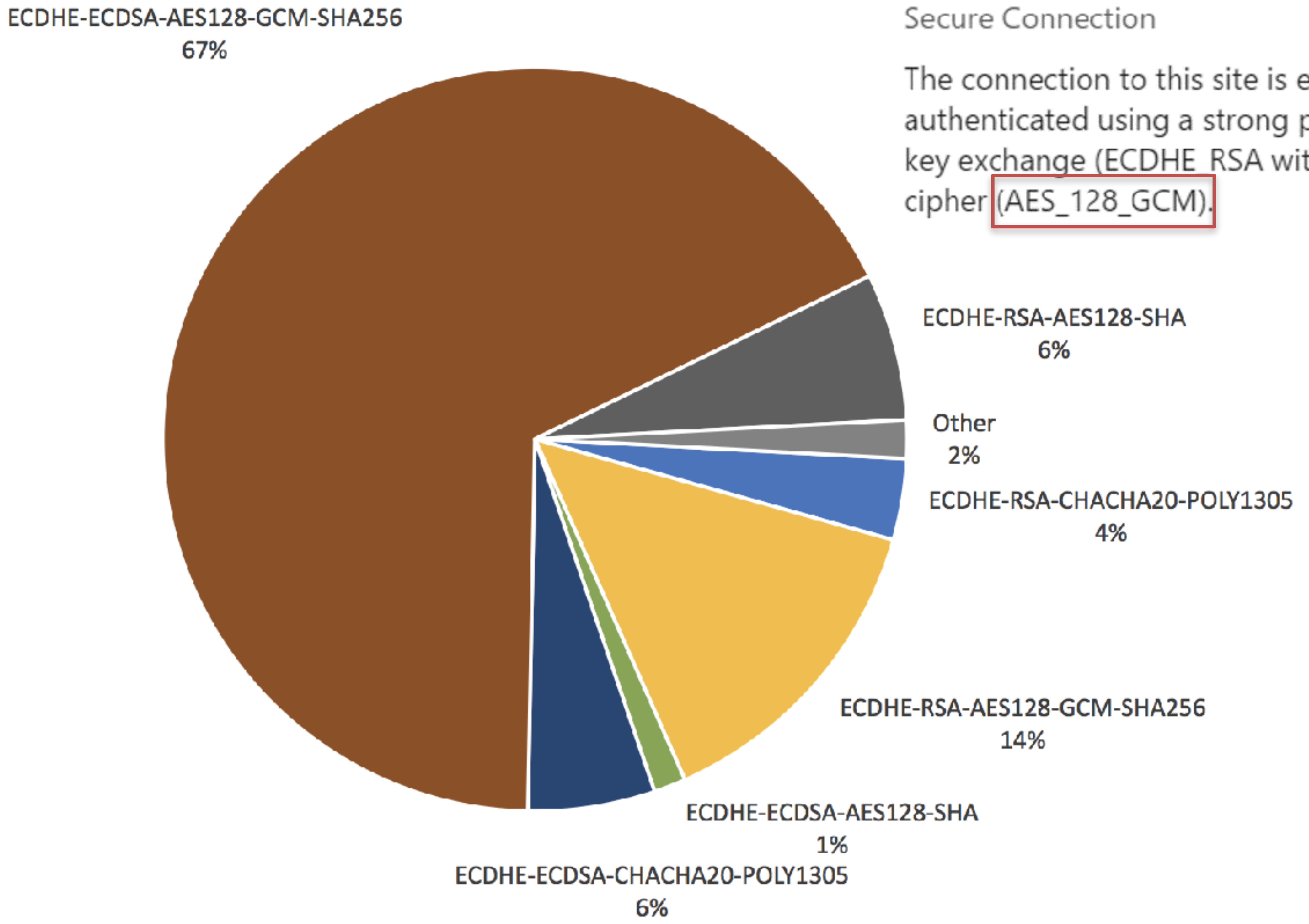


Cryptographic doom principle

If you have to perform any crypto operation before verifying the MAC on a message you've received, it will somehow inevitably lead to doom!

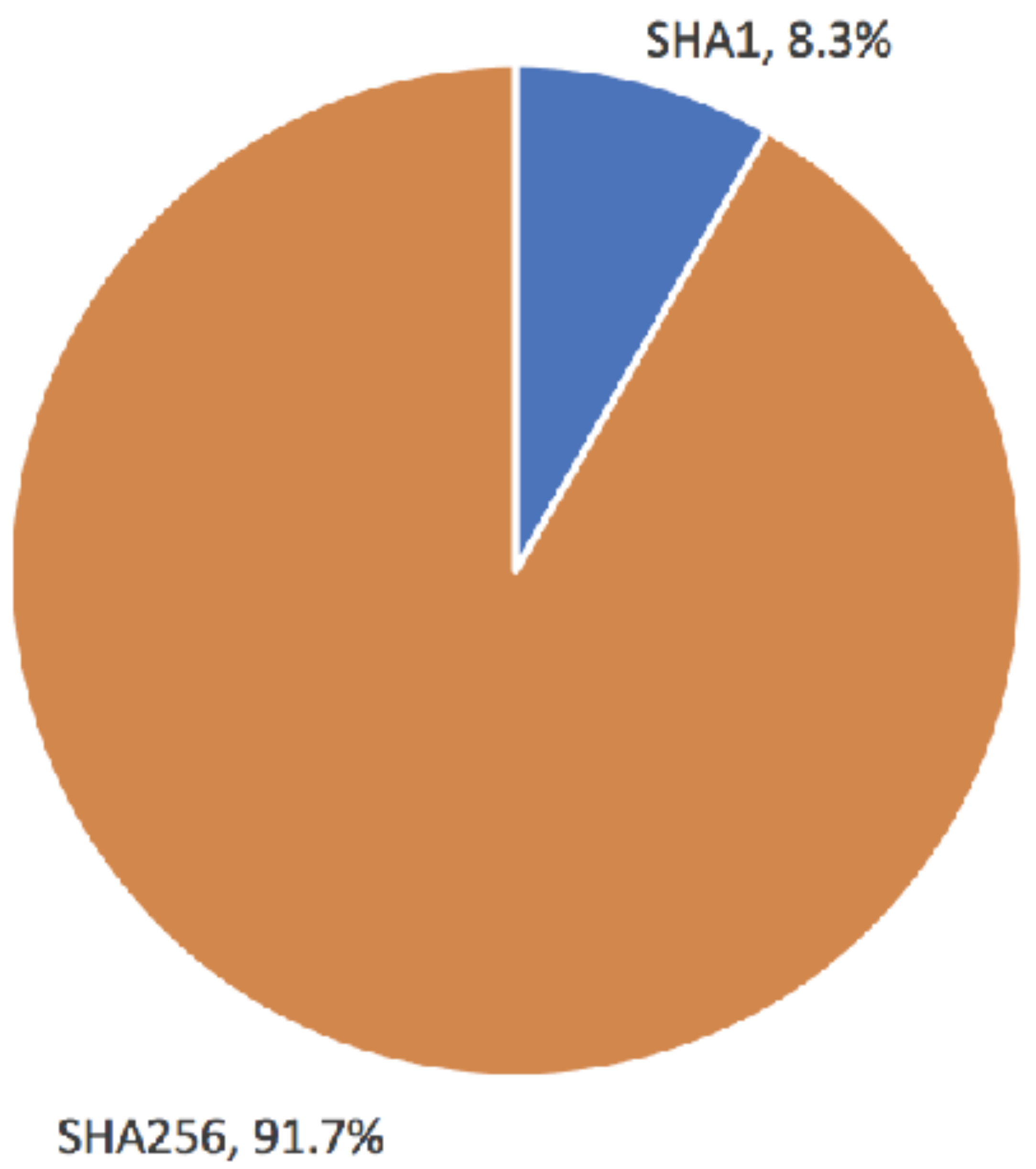
– *Moxie Marlinspike*

Widespread use of good crypto: Auth Enc, SHA-2, …



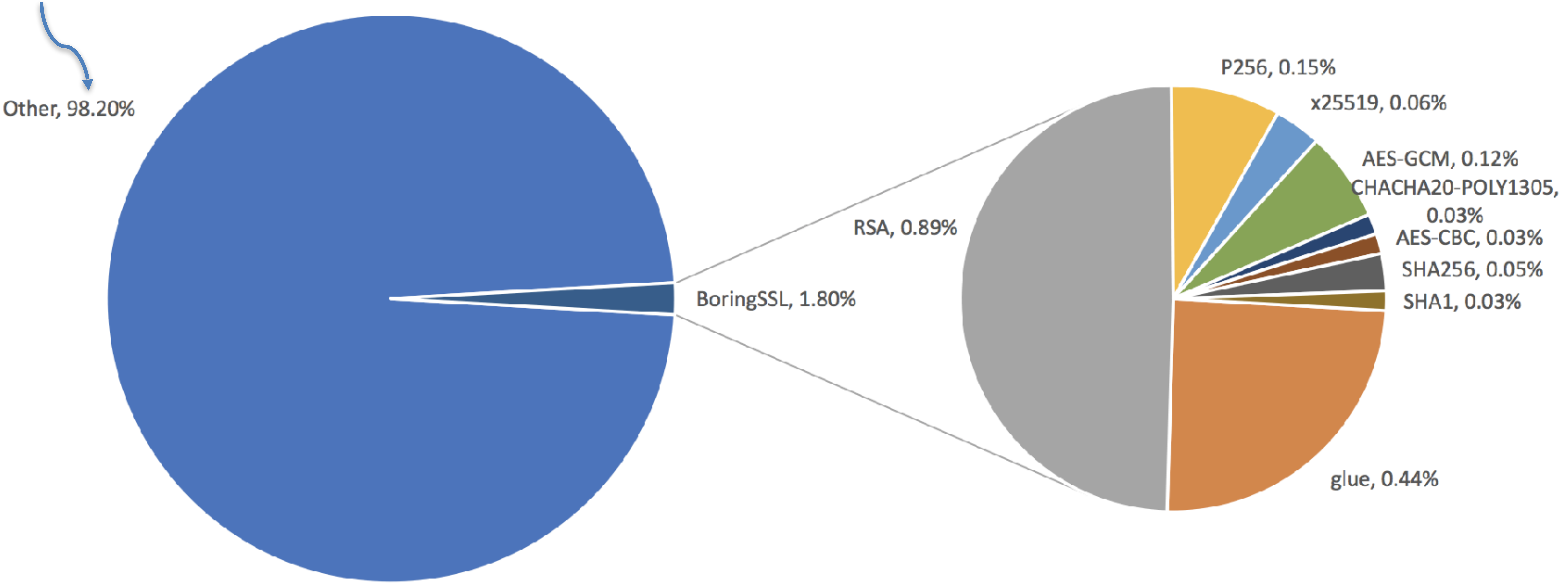
Secure Connection

The connection to this site is encrypted and authenticated using a strong protocol (QUIC), a strong key exchange (ECDHE_RSA with X25519), and a strong cipher (AES_128_GCM).

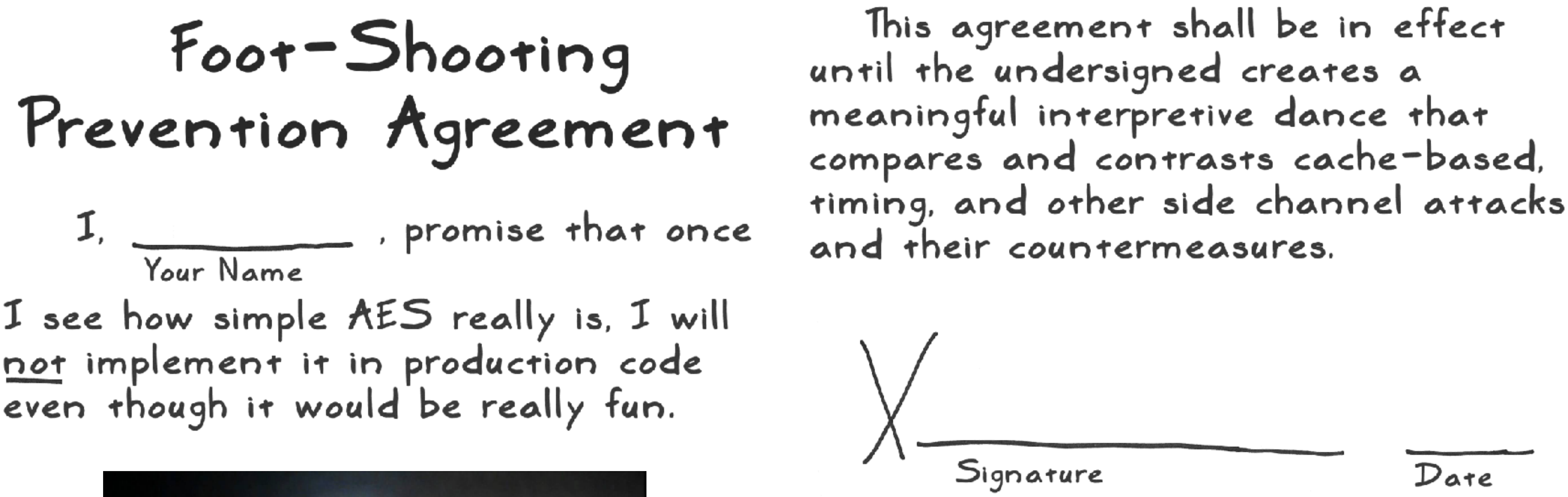


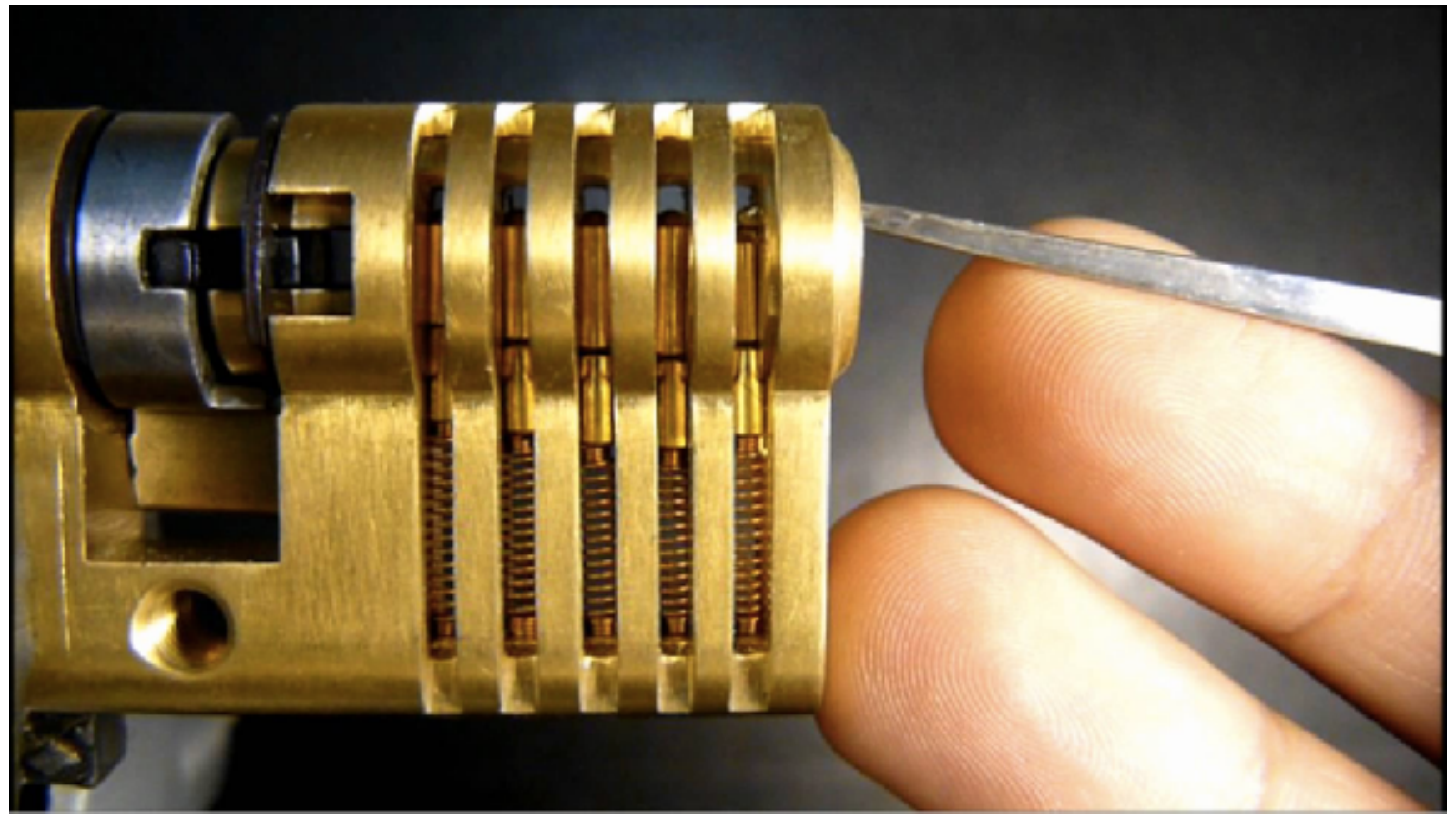
Crypto in TLS == really fast!

everything else the server does



Divide-and-conquer ⇒ Side channels + cryptanalysis





Source:
moserware.com/2009/09/stick-figure-guide-to-advanced.html

Key management ⇒ Access control

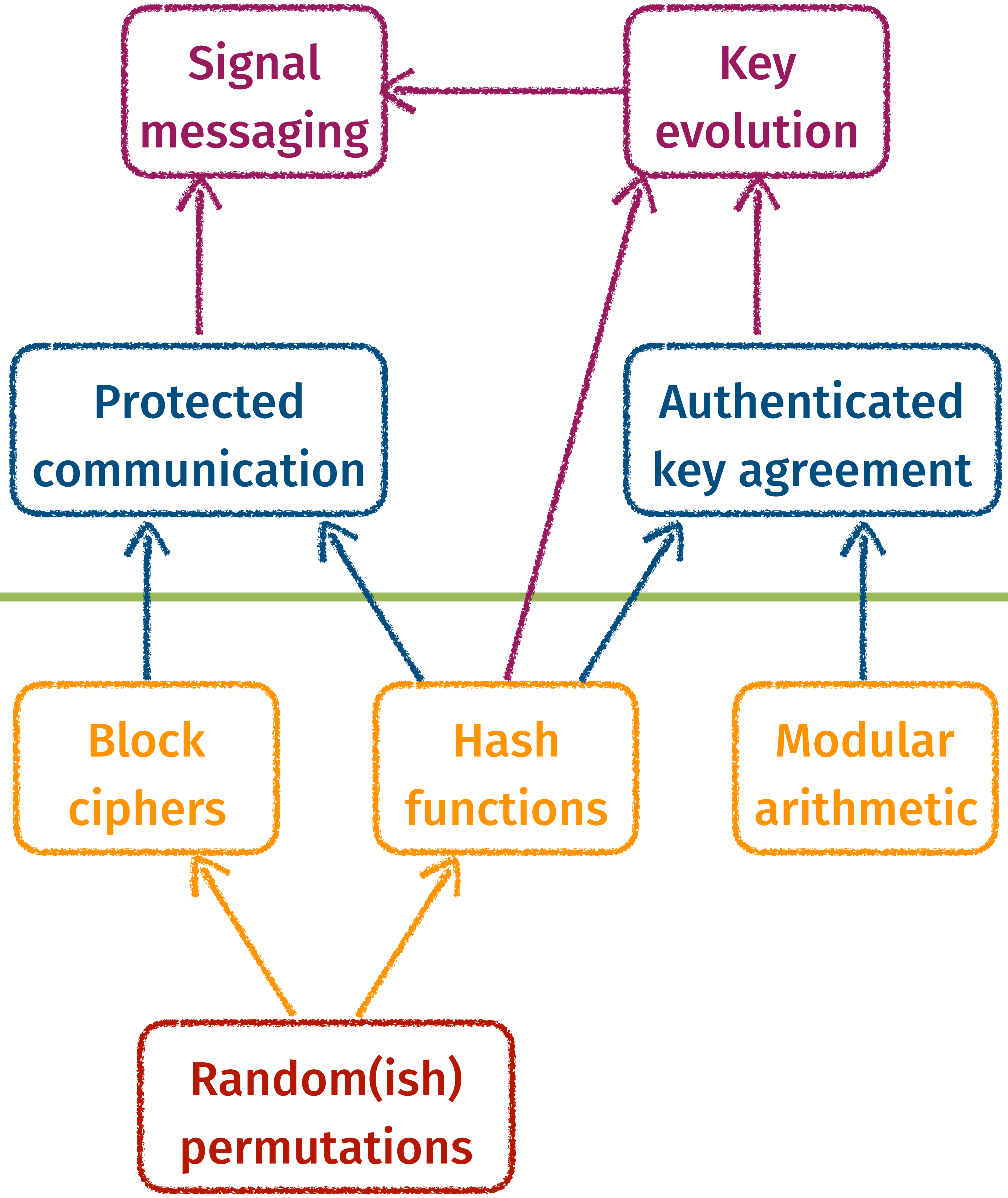
"This bar is pretty good, but you have to go stand in line for a ticket before they serve you."

Source:
twitter.com/sweis/status/982272891948421120

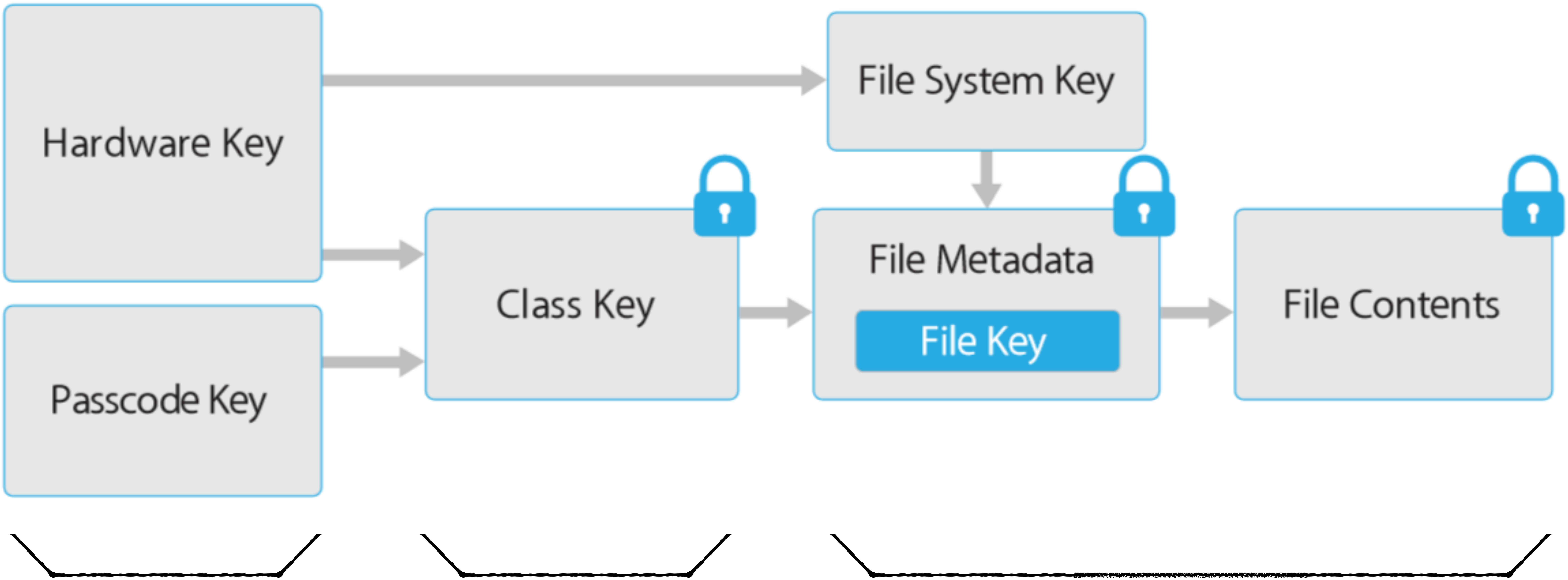
Signal: Deniability, forward + backward secrecy

Elegant protocols

Utilitarian tools



Deriving keys from a password



Master key generated from things that you know, are, and have

Derive keys that live only for limited time

Use these to wrap a unique key for each file

Cryptography *enables* **data analysis** *without* **data sharing**