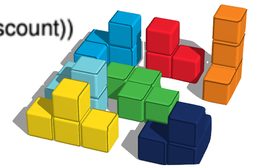
Link to paper (eprint:2025/1408)

On the Design of Modern Verifiable Databases

```
SELECT SUM(l_extendedprice* (1 - l_discount))
AS revenue
FROM lineitem, part
WHERE
(    p_partkey = l_partkey
  AND p_brand = 'Brand#41'
  AND p_container IN ('SM CASE', 'SM BOX', 'SM
      PACK', 'SM PKG')
```



Matteo Campanelli @ University of Tartu—October 7th 2025

Offchain Labs

matteo@offchainlabs.com

www.binarywhales.com

Integrity in Databases

Integrity in Databases

- Databases are at the hearth of our technological infrastructure

Integrity in Databases

- Databases are at the hearth of our technological infrastructure
- Outsourcing them is very common (for storage and processing)

Integrity in Databases

- Databases are at the hearth of our technological infrastructure
- Outsourcing them is very common (for storage and processing)
- This introduces risks:

Integrity in Databases

- Databases are at the hearth of our technological infrastructure
- Outsourcing them is very common (for storage and processing)
- This introduces risks:
  - “*AWS, would you give me the response to this query?*”

Integrity in Databases

- Databases are at the hearth of our technological infrastructure
- Outsourcing them is very common (for storage and processing)
- This introduces risks:
  - “*AWS, would you give me the response to this query?*”
  - **But how do we know the response is correct?**

Integrity in Databases

- Databases are at the hearth of our technological infrastructure
- Outsourcing them is very common (for storage and processing)
- This introduces risks:
  - “*AWS, would you give me the response to this query?*”
  - **But how do we know the response is correct?**
    - Arbitrary faults, malicious behavior,…

Integrity in Databases

- Databases are at the hearth of our technological infrastructure
- Outsourcing them is very common (for storage and processing)
- This introduces risks:
  - "*AWS, would you give me the response to this query?*"
  - **But how do we know the response is correct?**
    - Arbitrary faults, malicious behavior,…

"Verifiable" Databases (VDB)
are a cryptographic solution to this problem

Further Motivation for VDBs

Further Motivation for VDBs

- **Implication of Verifiable Databases:** not having to trust your DB provider

Further Motivation for VDBs

- **Implication of Verifiable Databases:** not having to trust your DB provider
- **Pipe dream** ≈ every data flow from every DB APIs authenticated through a verifiable DB

Further Motivation for VDBs

- **Implication of Verifiable Databases:** not having to trust your DB provider
- **Pipe dream** ≈ every data flow from every DB APIs authenticated through a verifiable DB
  - Analogy: HTTPS and its ubiquity

Further Motivation for VDBs

- **Implication of Verifiable Databases:** not having to trust your DB provider
- **Pipe dream** ≈ every data flow from every DB APIs authenticated through a verifiable DB
  - Analogy: HTTPS and its ubiquity
  - Potential outcome: Information flow that is fully certified cryptographically

Further Motivation for VDBs

- **Implication of Verifiable Databases:** not having to trust your DB provider
- **Pipe dream** ≈ every data flow from every DB APIs authenticated through a verifiable DB
  - Analogy: HTTPS and its ubiquity
  - Potential outcome: Information flow that is fully certified cryptographically
  - Even a “partial version” of the pipe dream might be useful, of course

Further Motivation for VDBs

- **Implication of Verifiable Databases:** not having to trust your DB provider
- **Pipe dream** ≈ every data flow from every DB APIs authenticated through a verifiable DB
  - Analogy: HTTPS and its ubiquity
  - Potential outcome: Information flow that is fully certified cryptographically
  - Even a “partial version” of the pipe dream might be useful, of course
- **Applications in addition to the above: blockchain settings**

Further Motivation for VDBs

- **Implication of Verifiable Databases:** not having to trust your DB provider
- **Pipe dream** ≈ every data flow from every DB APIs authenticated through a verifiable DB
  - Analogy: HTTPS and its ubiquity
  - Potential outcome: Information flow that is fully certified cryptographically
  - Even a “partial version” of the pipe dream might be useful, of course
- **Applications in addition to the above: blockchain settings**
  - providing proof for answers from “coprocessors”, etc.

Further Motivation for VDBs

- **Implication of Verifiable Databases:** not having to trust your DB provider
- **Pipe dream** ≈ every data flow from every DB APIs authenticated through a verifiable DB
  - Analogy: HTTPS and its ubiquity
  - Potential outcome: Information flow that is fully certified cryptographically
  - Even a “partial version” of the pipe dream might be useful, of course
- **Applications in addition to the above: blockchain settings**
  - providing proof for answers from “coprocessors”, etc.
  - coprocessor ≈ sends SomeAnalysis(chain) to the chain

Further Motivation for VDBs

- **Implication of Verifiable Databases:** not having to trust your DB provider
- **Pipe dream** ≈ every data flow from every DB APIs authenticated through a verifiable DB
  - Analogy: HTTPS and its ubiquity
  - Potential outcome: Information flow that is fully certified cryptographically
  - Even a “partial version” of the pipe dream might be useful, of course
- **Applications in addition to the above: blockchain settings**
  - providing proof for answers from “coprocessors”, etc.
  - coprocessor ≈ sends SomeAnalysis(chain) to the chain

lagrange

Axiom

SPACEANDTIME

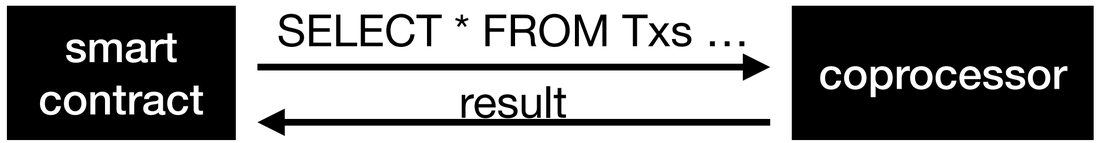
Further Motivation for VDBs

- **Implication of Verifiable Databases:** not having to trust your DB provider
- **Pipe dream** ≈ every data flow from every DB APIs authenticated through a verifiable DB
  - Analogy: HTTPS and its ubiquity
  - Potential outcome: Information flow that is fully certified cryptographically
  - Even a “partial version” of the pipe dream might be useful, of course
- **Applications in addition to the above: blockchain settings**
  - providing proof for answers from “coprocessors”, etc.
  - coprocessor ≈ sends SomeAnalysis(chain) to the chain

lagrange

Axiom

SPACE AND TIME



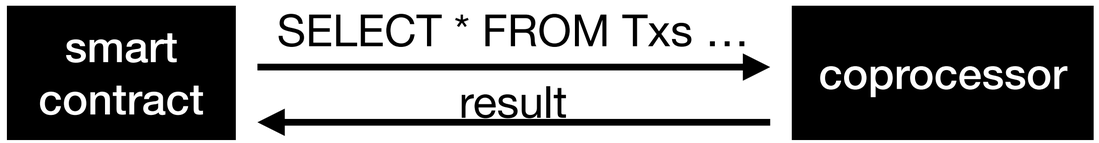
Further Motivation for VDBs

- **Implication of Verifiable Databases:** not having to trust your DB provider
- **Pipe dream** ≈ every data flow from every DB APIs authenticated through a verifiable DB
  - Analogy: HTTPS and its ubiquity
  - Potential outcome: Information flow that is fully certified cryptographically
  - Even a “partial version” of the pipe dream might be useful, of course
- **Applications in addition to the above: blockchain settings**
  - providing proof for answers from “coprocessors”, etc.
  - coprocessor ≈ sends SomeAnalysis(chain) to the chain

lagrange

Axiom

SPACE AND TIME



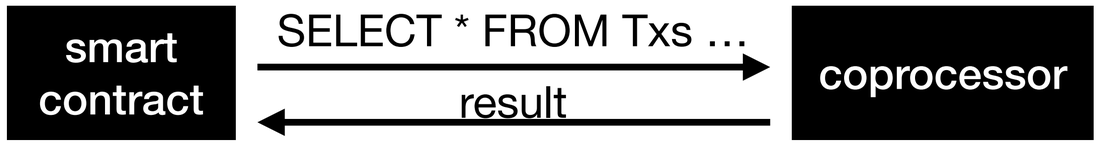
Further Motivation for VDBs

- **Implication of Verifiable Databases:** not having to trust your DB provider
- **Pipe dream** ≈ every data flow from every DB APIs authenticated through a verifiable DB
  - Analogy: HTTPS and its ubiquity
  - Potential outcome: Information flow that is fully certified cryptographically
  - Even a “partial version” of the pipe dream might be useful, of course
- **Applications in addition to the above: blockchain settings**
  - providing proof for answers from “coprocessors”, etc.
  - coprocessor ≈ sends SomeAnalysis(chain) to the chain

lagrange

Axiom

SPACE AND TIME



Before proceeding with verifiable databases, let’s have a quick cryptographic warm up.

Warm Up:
“Integrity” in Cryptography

Cryptographic “Integrity”—Common Examples

Cryptographic “Integrity”—Common Examples

- **Digital signatures**

Cryptographic “Integrity”—Common Examples

- **Digital signatures**
  - verifies a sender (and more)

Cryptographic “Integrity”—Common Examples

- **Digital signatures**
  - verifies a sender (and more)
- **Cryptographic hash functions, e.g., SHA, Blake**

Cryptographic “Integrity”—Common Examples

- **Digital signatures**
  - verifies a sender (and more)
- **Cryptographic hash functions, e.g., SHA, Blake**
  - verifies alterations on an object (and more)

Cryptographic “Integrity”—Common Examples

- **Digital signatures**
  - verifies a sender (and more)
- **Cryptographic hash functions, e.g., SHA, Blake**
  - verifies alterations on an object (and more)
  - Example (file sharing):

Cryptographic “Integrity”—Common Examples

- **Digital signatures**
  - verifies a sender (and more)
- **Cryptographic hash functions, e.g., SHA, Blake**
  - verifies alterations on an object (and more)
  - Example (file sharing):
    - “The file I expect should have hash *h*”

Cryptographic “Integrity”—Common Examples

- **Digital signatures**
  - verifies a sender (and more)
- **Cryptographic hash functions, e.g., SHA, Blake**
  - verifies alterations on an object (and more)
  - Example (file sharing):
    - “The file I expect should have hash *h*”
    - $\text{assert}\big(\text{H(fileReceived)} == h\big)$

Cryptographic “Integrity”—Less Common Examples (Merkle Trees)

Cryptographic “Integrity”—Less Common Examples (Merkle Trees)

- **Limitation with** $\mathrm{assert}\big(\mathrm{H}(\mathrm{fileReceived}) == h\big)$**?**

Cryptographic “Integrity”—Less Common Examples (Merkle Trees)

- **Limitation with** $\text{assert}\big(\text{H(fileReceived)} == h\big)$**?**
  - I must hash the *whole* object (the file) to check its integrity

Cryptographic “Integrity”—Less Common Examples (Merkle Trees)

- **Limitation with** $\text{assert}\big(\text{H(fileReceived)} == h\big)$**?**
  - I must hash the *whole* object (the file) to check its integrity
- **What if I simply want this for example?**

Cryptographic “Integrity”—Less Common Examples (Merkle Trees)

- **Limitation with** $\text{assert}\big(\text{H}(\text{fileReceived}) == h\big)$**?**
  - I must hash the *whole* object (the file) to check its integrity
- **What if I simply want this for example?**
  - I receive a claimed suffix of the file and want to check whether it actually is the suffix.

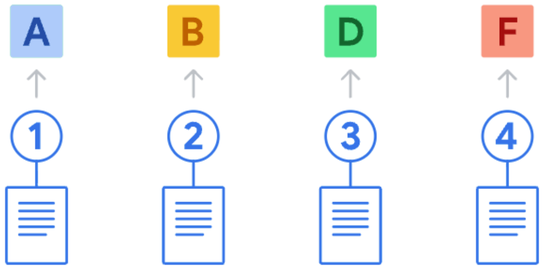
Cryptographic “Integrity”—Less Common Examples (Merkle Trees)

- **Limitation with** $\text{assert}\big(\text{H}(\text{fileReceived}) == h\big)$**?**
  - I must hash the *whole* object (the file) to check its integrity
- **What if I simply want this for example?**
  - I receive a claimed suffix of the file and want to check whether it actually is the suffix.
  - Checking whether a public key is in a list of a trusted keys (without reading the whole list)

Cryptographic "Integrity"—Less Common Examples (Merkle Trees)

- **Limitation with** $\text{assert}\big(\text{H}(\text{fileReceived}) == h\big)$**?**
  - I must hash the *whole* object (the file) to check its integrity
- **What if I simply want this for example?**
  - I receive a claimed suffix of the file and want to check whether it actually is the suffix.
  - Checking whether a public key is in a list of a trusted keys (without reading the whole list)

A non-solution:
just hash each piece.

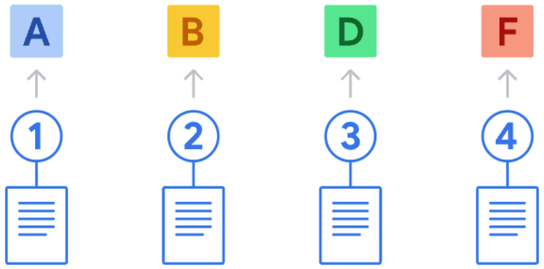


[Image source: transparency.dev]

Cryptographic "Integrity"—Less Common Examples (Merkle Trees)

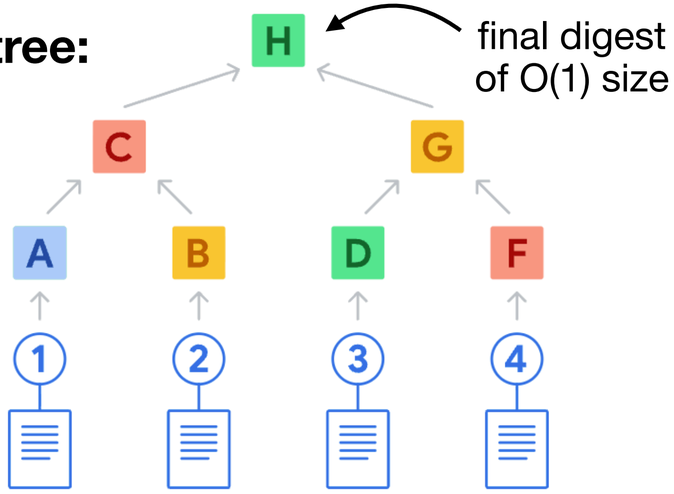
- **Limitation with** $\mathrm{assert}\big(\mathrm{H}(\mathrm{fileReceived}) == h\big)$**?**
  - I must hash the *whole* object (the file) to check its integrity
- **What if I simply want this for example?**
  - I receive a claimed suffix of the file and want to check whether it actually is the suffix.
  - Checking whether a public key is in a list of a trusted keys (without reading the whole list)

A non-solution:
just hash each piece.



[Image source: transparency.dev]

Instead,
make a tree:

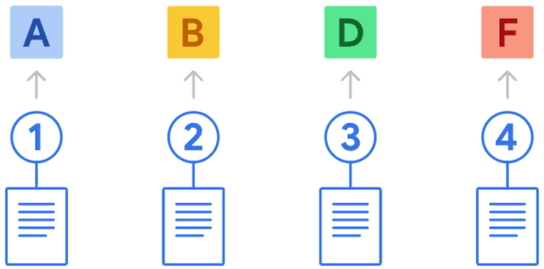


We concatenate before hashing. E.g. C = H(A||B)

Cryptographic "Integrity"—Less Common Examples (Merkle Trees)

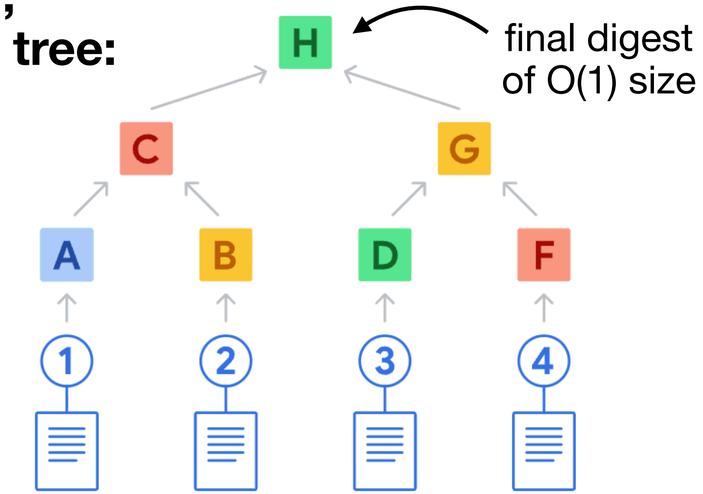
- **Limitation with** $\mathrm{assert}\big(\mathrm{H}(\mathrm{fileReceived}) == h\big)$**?**
  - I must hash the *whole* object (the file) to check its integrity
- **What if I simply want this for example?**
  - I receive a claimed suffix of the file and want to check whether it actually is the suffix.
  - Checking whether a public key is in a list of a trusted keys (without reading the whole list)

A non-solution: just hash each piece.



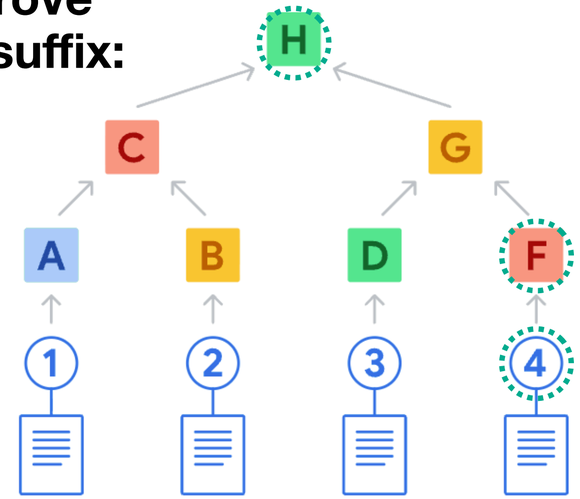
[Image source: transparency.dev]

Instead, make a tree:



We concatenate before hashing. E.g. C = H(A||B)

To prove 4 is suffix:

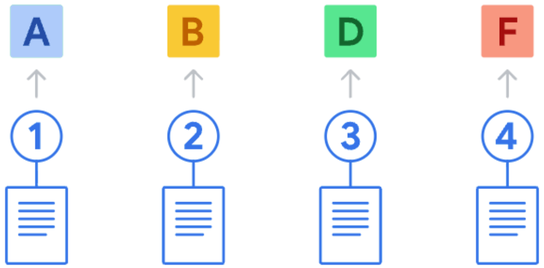


client has or can compute

Cryptographic "Integrity"—Less Common Examples (Merkle Trees)

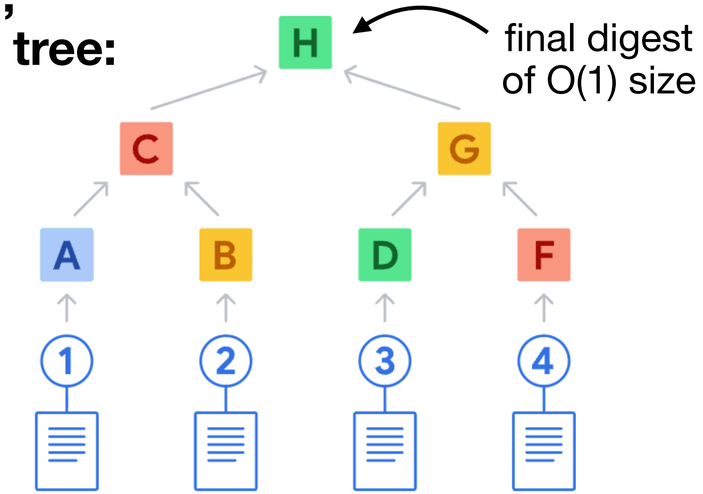
- **Limitation with** $\mathrm{assert}\big(\mathrm{H}(\mathrm{fileReceived}) == h\big)$**?**
  - I must hash the *whole* object (the file) to check its integrity
- **What if I simply want this for example?**
  - I receive a claimed suffix of the file and want to check whether it actually is the suffix.
  - Checking whether a public key is in a list of a trusted keys (without reading the whole list)

A non-solution:
just hash each piece.



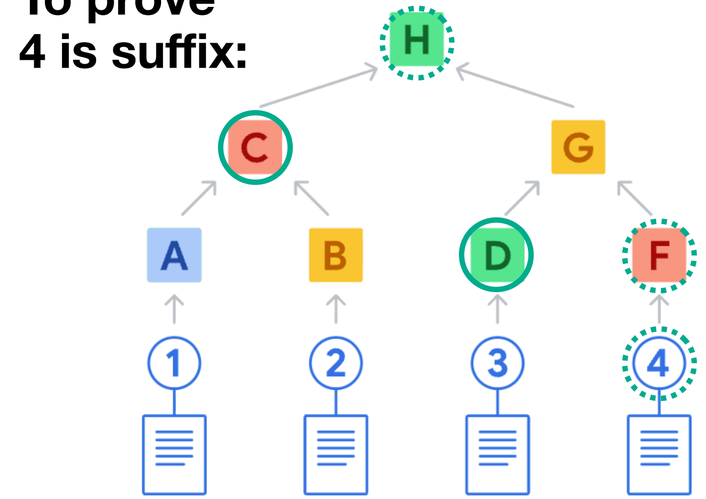
[Image source: transparency.dev]

Instead,
make a tree:



We concatenate before hashing. E.g. C = H(A||B)

To prove
4 is suffix:



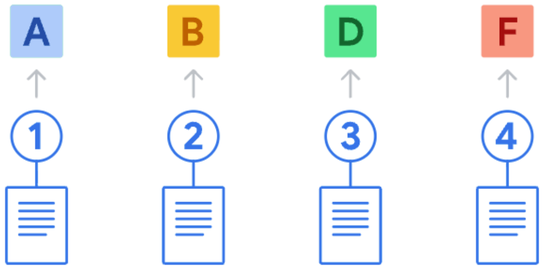
client has or can compute

client receives

Cryptographic "Integrity"—Less Common Examples (Merkle Trees)

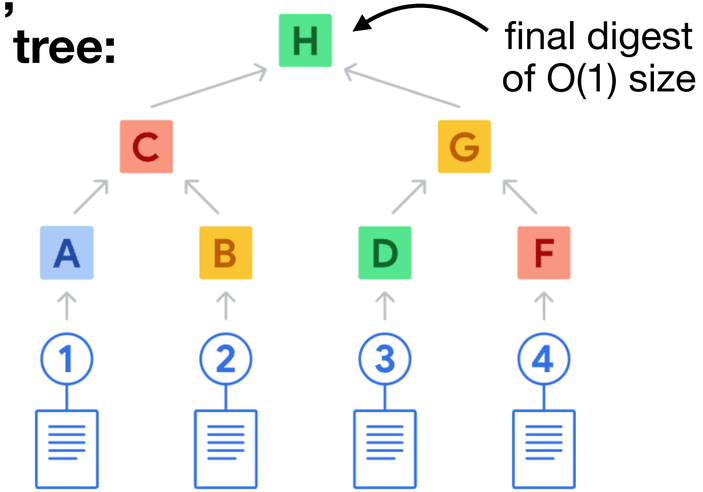
- **Limitation with** $\mathrm{assert}\big(\mathrm{H}(\mathrm{fileReceived}) == h\big)$**?**
  - I must hash the *whole* object (the file) to check its integrity
- **What if I simply want this for example?**
  - I receive a claimed suffix of the file and want to check whether it actually is the suffix.
  - Checking whether a public key is in a list of a trusted keys (without reading the whole list)

A non-solution:
just hash each piece.



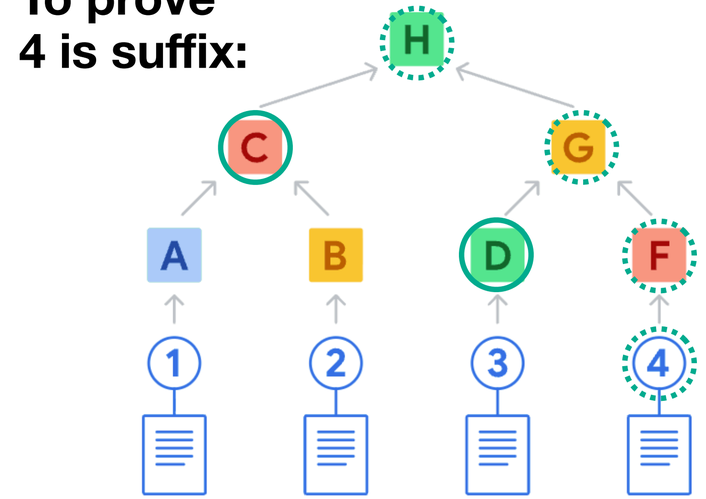
[Image source: transparency.dev]

Instead,
make a tree:



We concatenate before hashing. E.g. C = H(A||B)

To prove
4 is suffix:



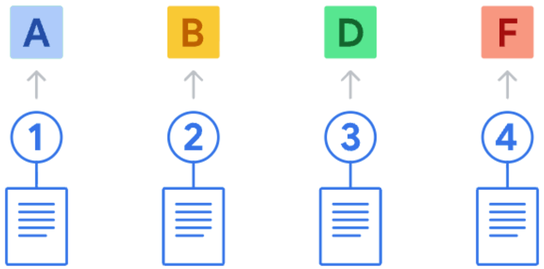
client has or can compute

client receives

Cryptographic "Integrity"—Less Common Examples (Merkle Trees)

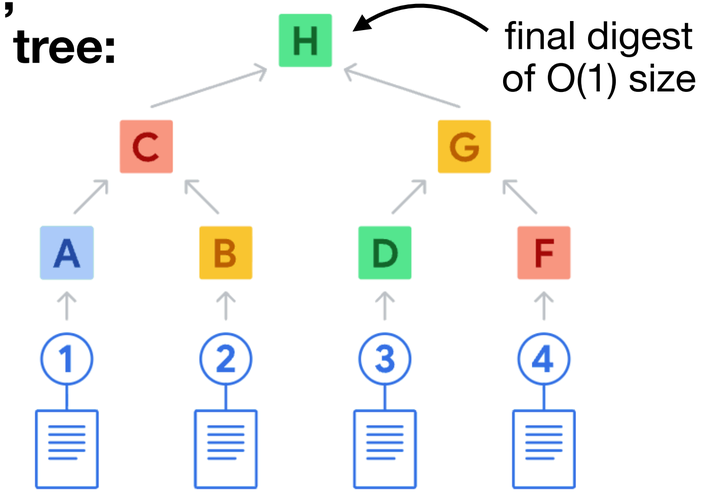
- **Limitation with** $\text{assert}\big(\text{H(fileReceived)} == h\big)$**?**
  - I must hash the *whole* object (the file) to check its integrity
- **What if I simply want this for example?**
  - I receive a claimed suffix of the file and want to check whether it actually is the suffix.
  - Checking whether a public key is in a list of a trusted keys (without reading the whole list)

A non-solution: just hash each piece.



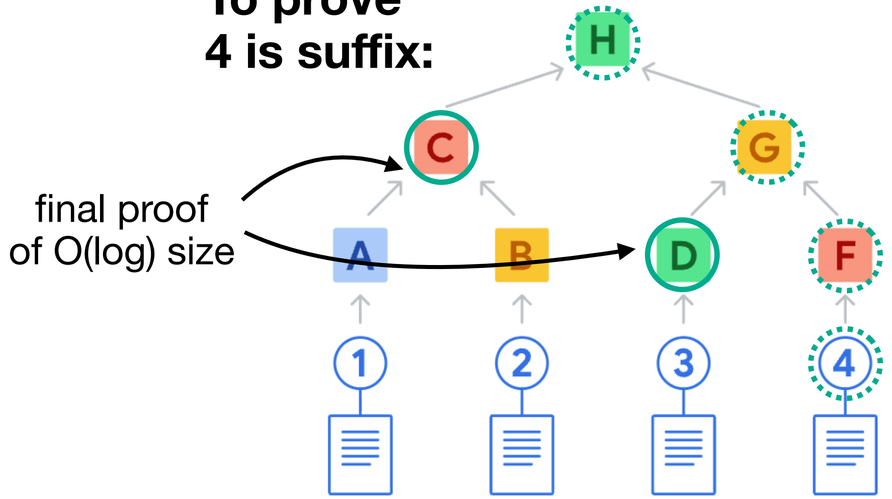
[Image source: transparency.dev]

Instead, make a tree:



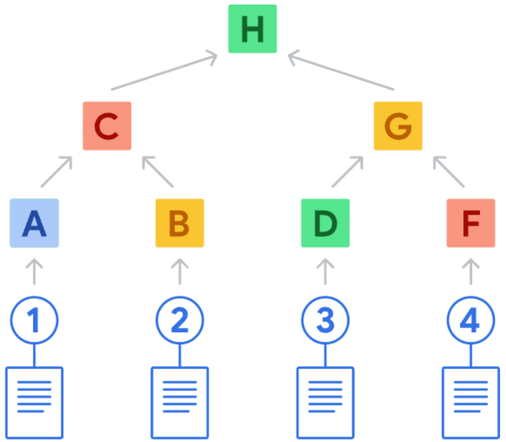
We concatenate before hashing. E.g. C = H(A||B)

To prove 4 is suffix:



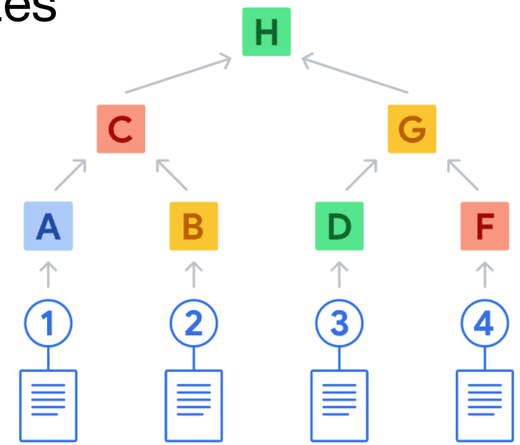
client has or can compute → client receives

From Merkle Trees to Authenticated Data Structures



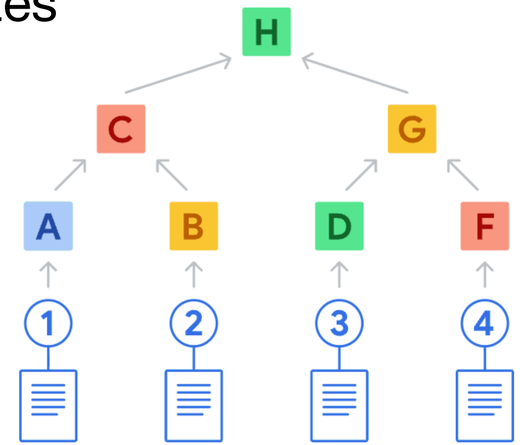
From Merkle Trees to Authenticated Data Structures

- An **Authenticated Data Structure** (ADS) is a construction that “authenticates” a data structure



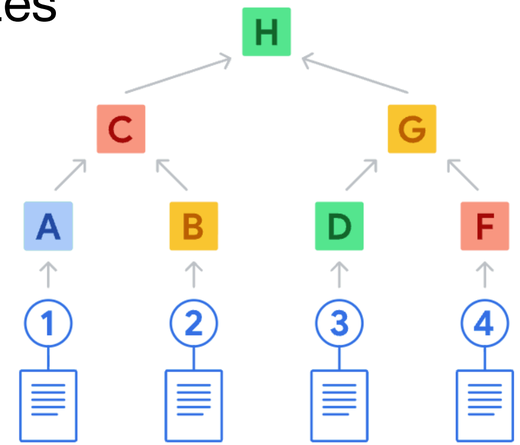
From Merkle Trees to Authenticated Data Structures

- An **Authenticated Data Structure** (ADS) is a construction that “authenticates” a data structure
  - can produce a **digest** to object of type X



From Merkle Trees to Authenticated Data Structures

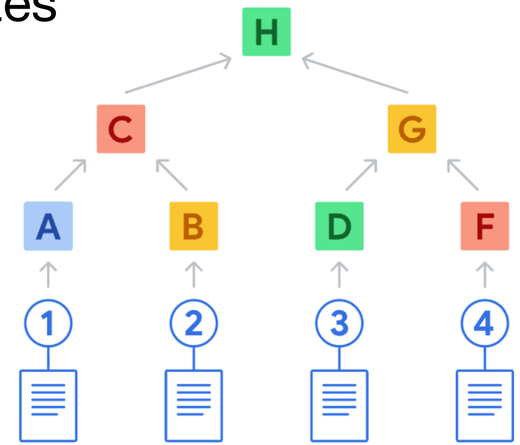
- An **Authenticated Data Structure** (ADS) is a construction that “authenticates” a data structure
  - can produce a **digest** to object of type X
  - can **prove** property Y about (without providing X in its entirety*)



* a property called succinctness

From Merkle Trees to Authenticated Data Structures

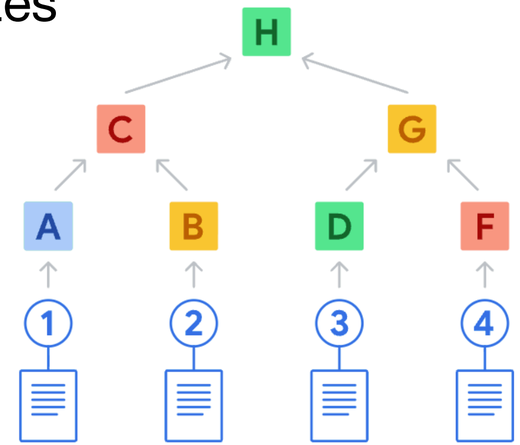
- An **Authenticated Data Structure** (ADS) is a construction that “authenticates” a data structure
  - can produce a **digest** to object of type X
  - can **prove** property Y about (without providing X in its entirety*)
- Merkle Trees are an example for:



* a property called succinctness

From Merkle Trees to Authenticated Data Structures

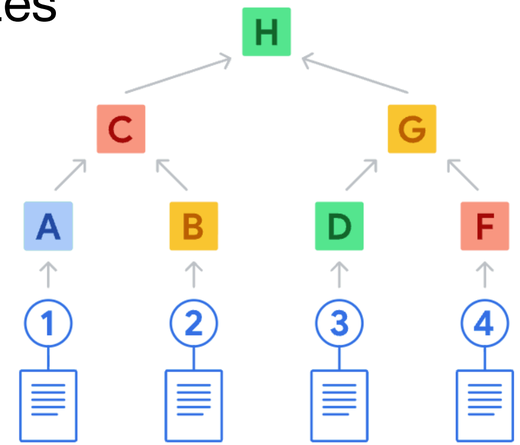
- An **Authenticated Data Structure** (ADS) is a construction that “authenticates” a data structure
  - can produce a **digest** to object of type X
  - can **prove** property Y about (without providing X in its entirety*)
- Merkle Trees are an example for:
  - **X = set** (root of tree is a digest to a set)



* a property called succinctness

From Merkle Trees to Authenticated Data Structures

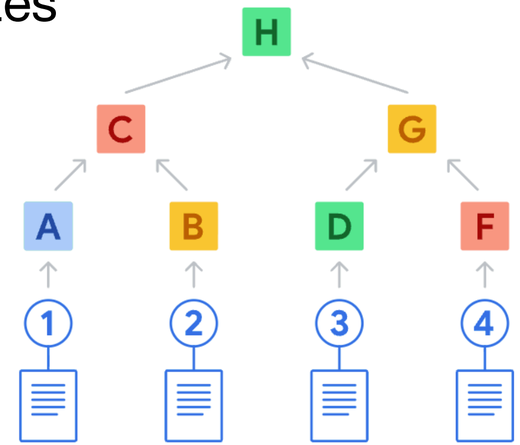
- An **Authenticated Data Structure** (ADS) is a construction that “authenticates” a data structure
  - can produce a **digest** to object of type X
  - can **prove** property Y about (without providing X in its entirety*)
- Merkle Trees are an example for:
  - **X = set** (root of tree is a digest to a set)
  - **Y = set membership**



* a property called succinctness

From Merkle Trees to Authenticated Data Structures

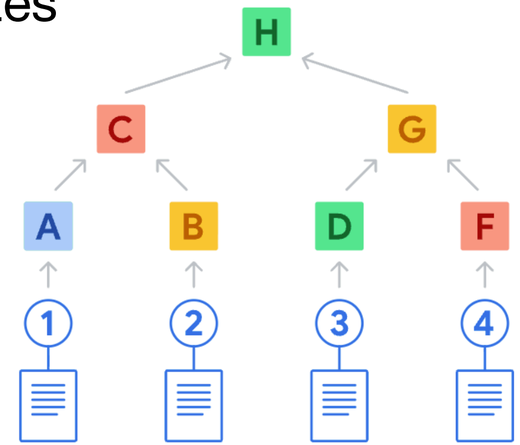
- An **Authenticated Data Structure** (ADS) is a construction that “authenticates” a data structure
  - can produce a **digest** to object of type X
  - can **prove** property Y about (without providing X in its entirety*)
- Merkle Trees are an example for:
  - **X = set** (root of tree is a digest to a set)
  - **Y = set membership**
  - such an ADS is called an **accumulator**



* a property called succinctness

From Merkle Trees to Authenticated Data Structures

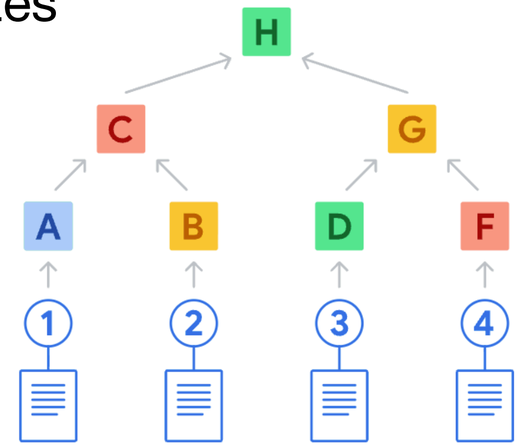
- An **Authenticated Data Structure** (ADS) is a construction that “authenticates” a data structure
  - can produce a **digest** to object of type X
  - can **prove** property Y about (without providing X in its entirety*)
- Merkle Trees are an example for:
  - **X = set** (root of tree is a digest to a set)
  - **Y = set membership**
  - such an ADS is called an **accumulator**
- Merkle Trees actually support also **X=vector**, **Y= lookup by index**



* a property called succinctness

From Merkle Trees to Authenticated Data Structures

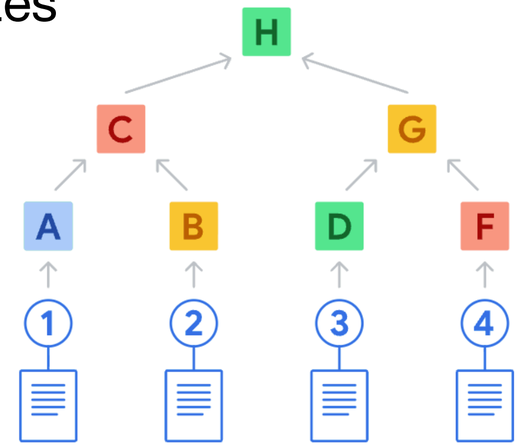
- An **Authenticated Data Structure** (ADS) is a construction that “authenticates” a data structure
  - can produce a **digest** to object of type X
  - can **prove** property Y about (without providing X in its entirety*)
- Merkle Trees are an example for:
  - **X = set** (root of tree is a digest to a set)
  - **Y = set membership**
  - such an ADS is called an **accumulator**
- Merkle Trees actually support also **X=vector**, **Y= lookup by index**
  - such an ADS is called a **vector commitment**



* a property called succinctness

From Merkle Trees to Authenticated Data Structures

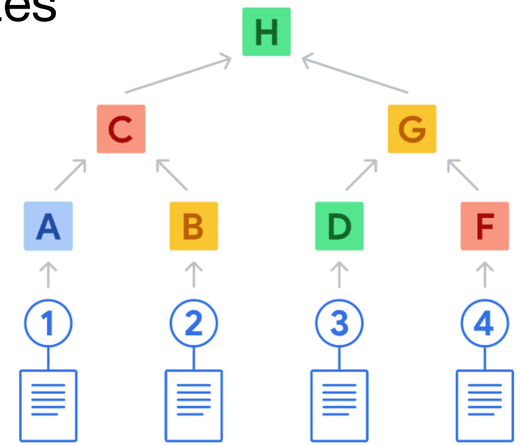
- An **Authenticated Data Structure** (ADS) is a construction that “authenticates” a data structure
  - can produce a **digest** to object of type X
  - can **prove** property Y about (without providing X in its entirety*)
- Merkle Trees are an example for:
  - **X = set** (root of tree is a digest to a set)
  - **Y = set membership**
  - such an ADS is called an **accumulator**
- Merkle Trees actually support also **X=vector**, **Y= lookup by index**
  - such an ADS is called a **vector commitment**
- But there are more examples of ADS:



* a property called succinctness

From Merkle Trees to Authenticated Data Structures

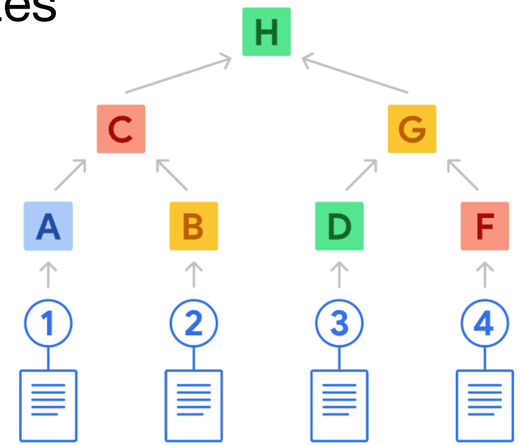
- An **Authenticated Data Structure** (ADS) is a construction that “authenticates” a data structure
  - can produce a **digest** to object of type X
  - can **prove** property Y about (without providing X in its entirety*)
- Merkle Trees are an example for:
  - **X = set** (root of tree is a digest to a set)
  - **Y = set membership**
  - such an ADS is called an **accumulator**
- Merkle Trees actually support also **X=vector**, **Y= lookup by index**
  - such an ADS is called a **vector commitment**
- But there are more examples of ADS:
  - **Authenticated Range Trees** (what the name suggests)



* a property called succinctness

From Merkle Trees to Authenticated Data Structures

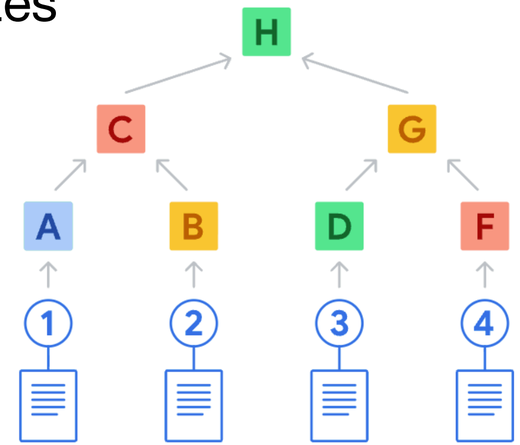
- An **Authenticated Data Structure** (ADS) is a construction that "authenticates" a data structure
  - can produce a **digest** to object of type X
  - can **prove** property Y about (without providing X in its entirety*)
- Merkle Trees are an example for:
  - **X = set** (root of tree is a digest to a set)
  - **Y = set membership**
  - such an ADS is called an **accumulator**
- Merkle Trees actually support also **X=vector**, **Y= lookup by index**
  - such an ADS is called a **vector commitment**
- But there are more examples of ADS:
  - **Authenticated Range Trees** (what the name suggests)
  - **Polynomial Commitments** (X = polynomial, Y = poly evaluation)



* a property called succinctness

From Merkle Trees to Authenticated Data Structures

- An **Authenticated Data Structure** (ADS) is a construction that "authenticates" a data structure
  - can produce a **digest** to object of type X
  - can **prove** property Y about (without providing X in its entirety*)
- Merkle Trees are an example for:
  - **X = set** (root of tree is a digest to a set)
  - **Y = set membership**
  - such an ADS is called an **accumulator**
- Merkle Trees actually support also **X=vector**, **Y= lookup by index**
  - such an ADS is called a **vector commitment**
- But there are more examples of ADS:
  - **Authenticated Range Trees** (what the name suggests)
  - **Polynomial Commitments** (X = polynomial, Y = poly evaluation)
  - …



* a property called succinctness

Wait. Weren’t We Talking About Signatures a Second Ago?

Integrity is a ductile notion

Wait. Weren’t We Talking About Signatures a Second Ago?

Integrity is a ductile notion

Integrity ≈ “is this what I expect?”

Wait. Weren't We Talking About Signatures a Second Ago?

Integrity is a ductile notion

Integrity ≈ "is this what I expect?"

More traditional notions of integrity

Signatures

Hashing

Wait. Weren't We Talking About Signatures a Second Ago?

Integrity is a ductile notion

Integrity ≈ "is this what I expect?"

More traditional notions of integrity → **More fine-grained notions of integrity**
(integrity as guarantee of a data structure satisfying a property)

Signatures

Hashing

Authenticated Data Structures (Merkle Trees, …)

Wait. Weren't We Talking About Signatures a Second Ago?

Integrity is a ductile notion

Integrity ≈ "is this what I expect?"

More traditional notions of integrity → **More fine-grained notions of integrity** →

(integrity as guarantee of a data structure satisfying a property)

Signatures

Hashing

Authenticated Data Structures (Merkle Trees, …)

Wait. Weren't We Talking About Signatures a Second Ago?

Integrity is a ductile notion

Integrity ≈ "is this what I expect?"

More traditional notions of integrity → **More fine-grained notions of integrity**
(integrity as guarantee of a data structure satisfying a property) → **?**

Signatures

Hashing

Authenticated Data Structures (Merkle Trees, …)

Wait. Weren't We Talking About Signatures a Second Ago?

Integrity is a ductile notion

Integrity ≈ "is this what I expect?"

More traditional notions of integrity → **More fine-grained notions of integrity** →

(integrity as guarantee of a data structure satisfying a property)

Signatures

Hashing

Authenticated Data Structures (Merkle Trees, …)

Wait. Weren't We Talking About Signatures a Second Ago?

Integrity is a ductile notion

Integrity ≈ "is this what I expect?"

More traditional notions of integrity → **More fine-grained notions of integrity** *(integrity as guarantee of a data structure satisfying a property)* → **Computational Integrity**

Signatures

Hashing

Authenticated Data Structures (Merkle Trees, …)

Wait. Weren't We Talking About Signatures a Second Ago?

Integrity is a ductile notion

Integrity ≈ "is this what I expect?"

More traditional notions of integrity → **More fine-grained notions of integrity** *(integrity as guarantee of a data structure satisfying a property)* → **Computational Integrity**

Signatures

Hashing

Authenticated Data Structures (Merkle Trees, …)

General Cryptographic Proofs

General Cryptographic Proofs

“Computational Integrity”

Server (Prover)

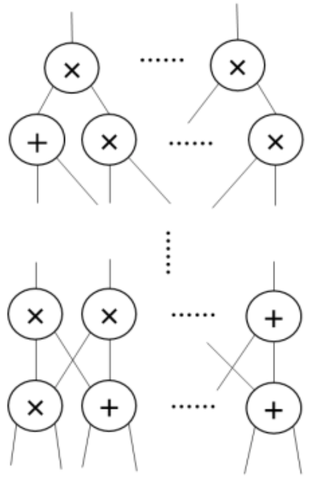
Client (Verifier)

General Cryptographic Proofs

“Computational Integrity”

Server (Prover)

Client (Verifier)



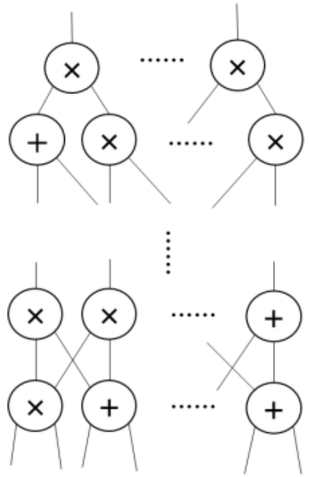
Some program *F*

General Cryptographic Proofs

“Computational Integrity”

Server (Prover)

Client (Verifier)



Some program *F*

Client would like to learn
the value of F(someInput)

General Cryptographic Proofs

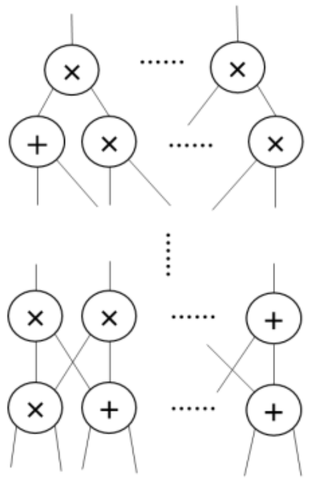
“Computational Integrity”

Server (Prover)

“digest”

h_F

Client (Verifier)



Some program *F*

Client would like to learn
the value of $F(\texttt{someInput})$

General Cryptographic Proofs

“Computational Integrity”

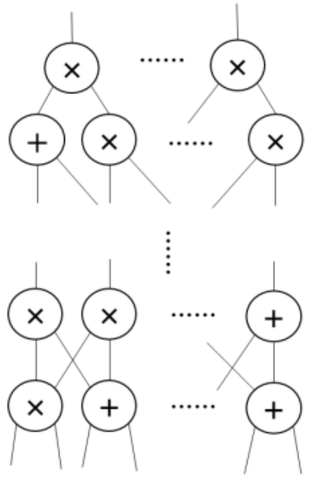
Server (Prover)

y, π

Client (Verifier)

h_F

“digest”



Some program *F*

Client would like to learn the value of $F(\text{someInput})$

General Cryptographic Proofs

"Computational Integrity"

Server (Prover)

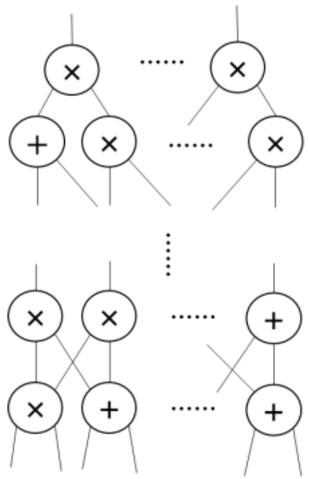
Claimed result

y, π

Client (Verifier)

h_F

"digest"



Some program *F*

Client would like to learn the value of $F(\text{someInput})$

General Cryptographic Proofs

"Computational Integrity"

Server (Prover)

Claimed result

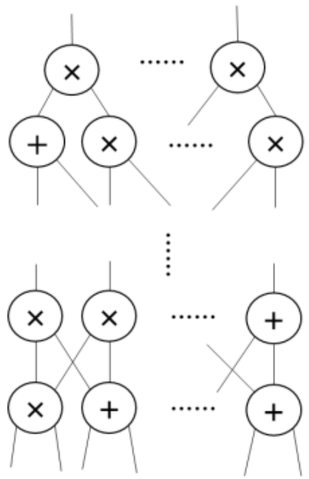
y, π

Proof that response is correct

Client (Verifier)

"digest"

h_F



Some program *F*

Client would like to learn the value of $F(\text{someInput})$

General Cryptographic Proofs

"Computational Integrity"

Server (Prover)

Claimed result

y, π

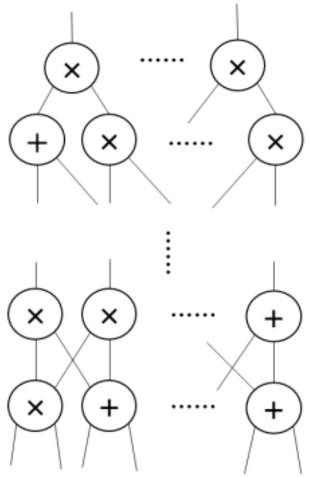
Proof that response is correct

Verify(h_F, someInput, y, π)

Client (Verifier)

"digest"

h_F



Some program *F*

Client would like to learn the value of F(someInput)

General Cryptographic Proofs

"Computational Integrity"

Server (Prover)

Claimed result

y, π

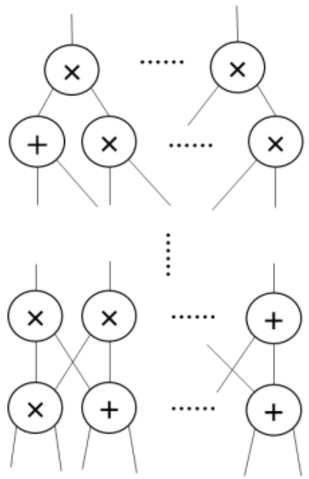
Proof that response is correct

Verify(h_F, someInput, y, π)

Client (Verifier)

"digest"

h_F



Some program *F*

Common requirement: **Succinctness**

(π is very small; Verify is very fast)

Client would like to learn the value of F(someInput)

General Cryptographic Proofs

"Computational Integrity"

Server (Prover)

Claimed result

y, π

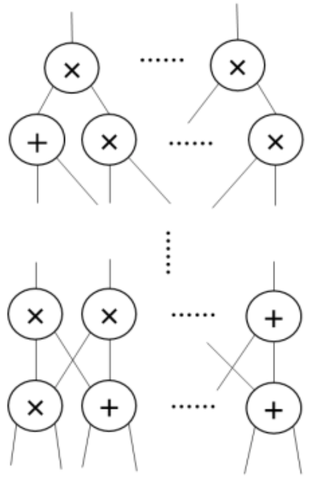
Proof that response is correct

Verify(h_F, someInput, y, π)

Client (Verifier)

"digest"

h_F



Some program *F*

Common requirement: **Succinctness**

(π is very small; Verify is very fast)

Client would like to learn the value of F(someInput)

* Fine print for the cryptographers: this slide mostly refers to SNARKs.

Back to Verifiable Databases

Verifiable Databases (VDB)

Server (Prover)

Client (Verifier)

Computationally “weak” client;
not going to store the DB

Verifiable Databases (VDB)

Server (Prover)

during some offline stage

DB, digest(DB)

Client (Verifier)

Computationally “weak” client;
not going to store the DB

Verifiable Databases (VDB)

Server (Prover)

during some offline stage

DB, digest(DB)

query

Client (Verifier)

Computationally “weak” client;
not going to store the DB

Verifiable Databases (VDB)

Server (Prover)

during some offline stage

DB, digest(DB)

query

response, π

Client (Verifier)

Computationally “weak” client;
not going to store the DB

Verifiable Databases (VDB)

Server (Prover)

during some offline stage

DB, digest(DB)

query

response, π

Proof that response is correct

Client (Verifier)

Computationally “weak” client;
not going to store the DB

Desirable Features in Verifiable Databases

Efficiency-related

Security-related

Desirable Features in Verifiable Databases

Efficiency-related

- Efficient (prover and verifier)

Security-related

Desirable Features in Verifiable Databases

Efficiency-related

- Efficient (prover and verifier)
- Publicly-verifiable

Security-related

Desirable Features in Verifiable Databases

Efficiency-related

- Efficient (prover and verifier)
- Publicly-verifiable
  - important to establish trust levels of data traces

Security-related

Desirable Features in Verifiable Databases

Efficiency-related

- Efficient (prover and verifier)
- Publicly-verifiable
  - important to establish trust levels of data traces
- Non-interactive, and with short proofs

Security-related

Desirable Features in Verifiable Databases

Efficiency-related

- Efficient (prover and verifier)
- Publicly-verifiable
  - important to establish trust levels of data traces
- Non-interactive, and with short proofs
  - especially important in smart contracts

Security-related

Desirable Features in Verifiable Databases

Efficiency-related

- Efficient (prover and verifier)
- Publicly-verifiable
  - important to establish trust levels of data traces
- Non-interactive, and with short proofs
  - especially important in smart contracts

Security-related

- Based on solid cryptographic assumptions (of course)

Desirable Features in Verifiable Databases

Efficiency-related

- Efficient (prover and verifier)
- Publicly-verifiable
  - important to establish trust levels of data traces
- Non-interactive, and with short proofs
  - especially important in smart contracts

Security-related

- Based on solid cryptographic assumptions (of course)
- Also simple

Desirable Features in Verifiable Databases

Efficiency-related

- Efficient (prover and verifier)
- Publicly-verifiable
  - important to establish trust levels of data traces
- Non-interactive, and with short proofs
  - especially important in smart contracts

Security-related

- Based on solid cryptographic assumptions (of course)
- Also simple
  - → easily auditable; easier to reason about

Desirable Features in Verifiable Databases

Efficiency-related

- Efficient (prover and verifier)
- Publicly-verifiable
  - important to establish trust levels of data traces
- Non-interactive, and with short proofs
  - especially important in smart contracts

Security-related

- Based on solid cryptographic assumptions (of course)
- Also simple
  - → easily auditable; easier to reason about
  - → less vulnerable

Desirable Features in Verifiable Databases

Efficiency-related

- Efficient (prover and verifier)
- Publicly-verifiable
  - important to establish trust levels of data traces
- Non-interactive, and with short proofs
  - especially important in smart contracts

Security-related

- Based on solid cryptographic assumptions (of course)
- Also simple
  - → easily auditable; easier to reason about
  - → less vulnerable
  - → more maintainable; easier to patch

Common Tradeoffs in Verifiable DB

Common Tradeoffs in Verifiable DB

More/Less Expressive

More/Less Practical

More/Less Simple
& Secure

Common Tradeoffs in Verifiable DB

More/Less Expressive

simple queries ⟷ complex queries

```
1. SELECT SUM(l_extendedprice* (1 - l_discount))
2. AS revenue
3. FROM lineitem, part
4. WHERE
5. (    p_partkey = l_partkey
6.    AND p_brand = 'Brand#41'
7.    AND p_container IN ('SM CASE', 'SM BOX', 'SM
          PACK', 'SM PKG')
8.    AND l_quantity >= 7 AND l_quantity <= 7 + 10
9.    AND p_size BETWEEN 1 AND 5
10.   AND l_shipmode IN ('AIR', 'AIR REG')
11.   AND l_shipinstruct = 'DELIVER IN PERSON' )
12. OR
13. (  p_partkey = l_partkey
14.   AND p_brand = 'Brand#14'
15.   AND p_container IN ('MED BAG', 'MED BOX',
          'MED PKG', 'MED PACK')
16.   AND l_quantity >= 14 AND l_quantity <= 14 + 10
```

More/Less Practical

More/Less Simple & Secure

Common Tradeoffs in Verifiable DB

More/Less Expressive

simple queries ⟷ complex queries

(this talk: only “SQL” DBs)

```
1. SELECT SUM(l_extendedprice* (1 - l_discount))
2. AS revenue
3. FROM lineitem, part
4. WHERE
5. (    p_partkey = l_partkey
6.    AND p_brand = ‘Brand#41’
7.    AND p_container IN (‘SM CASE’, ‘SM BOX’, ‘SM
          PACK’, ‘SM PKG’)
8.    AND l_quantity >= 7 AND l_quantity <= 7 + 10
9.    AND p_size BETWEEN 1 AND 5
10.   AND l_shipmode IN (‘AIR’, ‘AIR REG’)
11.   AND l_shipinstruct = ‘DELIVER IN PERSON’ )
12. OR
13. (   p_partkey = l_partkey
14.   AND p_brand = ‘Brand#14’
15.   AND p_container IN (‘MED BAG’, ‘MED BOX’,
          ‘MED PKG’, ‘MED PACK’)
16.   AND l_quantity >= 14 AND l_quantity <= 14 + 10
```

More/Less Practical

More/Less Simple & Secure

Common Tradeoffs in Verifiable DB

More/Less Expressive

simple queries ⟷ complex queries

(this talk: only “SQL” DBs)

```
1. SELECT SUM(l_extendedprice* (1 - l_discount))
2. AS revenue
3. FROM lineitem, part
4. WHERE
5. (    p_partkey = l_partkey
6.    AND p_brand = 'Brand#41'
7.    AND p_container IN ('SM CASE', 'SM BOX', 'SM
          PACK', 'SM PKG')
8.    AND l_quantity >= 7 AND l_quantity <= 7 + 10
9.    AND p_size BETWEEN 1 AND 5
10.   AND l_shipmode IN ('AIR', 'AIR REG')
11.   AND l_shipinstruct = 'DELIVER IN PERSON' )
12. OR
13. (  p_partkey = l_partkey
14.   AND p_brand = 'Brand#14'
15.   AND p_container IN ('MED BAG', 'MED BOX',
          'MED PKG', 'MED PACK')
16.   AND l_quantity >= 14 AND l_quantity <= 14 + 10
```

More/Less Practical

small proofs, fast prover,… ⟷ slower, larger proofs

More/Less Simple & Secure

Common Tradeoffs in Verifiable DB

More/Less Expressive

simple queries ⟷ complex queries

(this talk: only “SQL” DBs)

```
1. SELECT SUM(l_extendedprice* (1 - l_discount))
2. AS revenue
3. FROM lineitem, part
4. WHERE
5. (    p_partkey = l_partkey
6.    AND p_brand = 'Brand#41'
7.    AND p_container IN ('SM CASE', 'SM BOX', 'SM
          PACK', 'SM PKG')
8.    AND l_quantity >= 7 AND l_quantity <= 7 + 10
9.    AND p_size BETWEEN 1 AND 5
10.   AND l_shipmode IN ('AIR', 'AIR REG')
11.   AND l_shipinstruct = 'DELIVER IN PERSON' )
12. OR
13. (  p_partkey = l_partkey
14.   AND p_brand = 'Brand#14'
15.   AND p_container IN ('MED BAG', 'MED BOX',
          'MED PKG', 'MED PACK')
16.   AND l_quantity >= 14 AND l_quantity <= 14 + 10
```

More/Less Practical

small proofs, fast prover,… ⟷ slower, larger proofs

More/Less Simple & Secure

reliable assumptions, small tech stack ⟷ heuristic assumptions, lots of moving parts

The existing landscape of verifiable databases

How do we build verifiable databases?

How do we build verifiable databases?

More traditional notions of integrity

Signatures

Hashing

→

More fine-grained notions of integrity

(integrity as guarantee of a data structure satisfying a property)

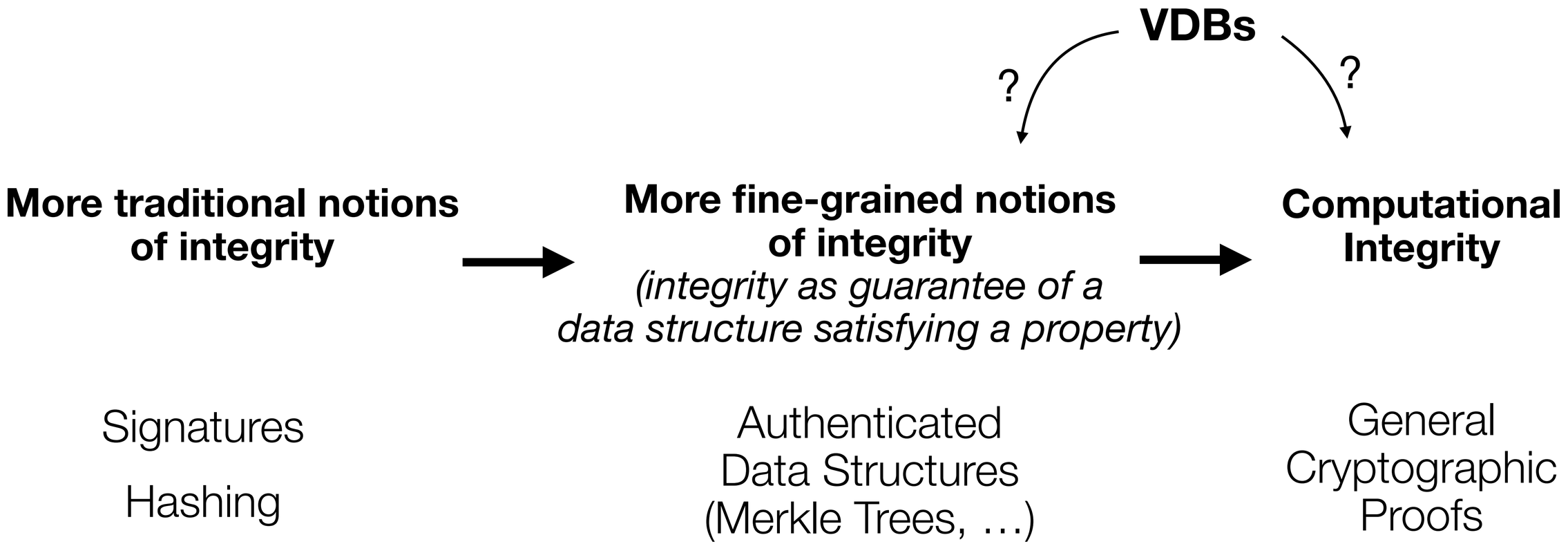
Authenticated Data Structures (Merkle Trees, …)

→

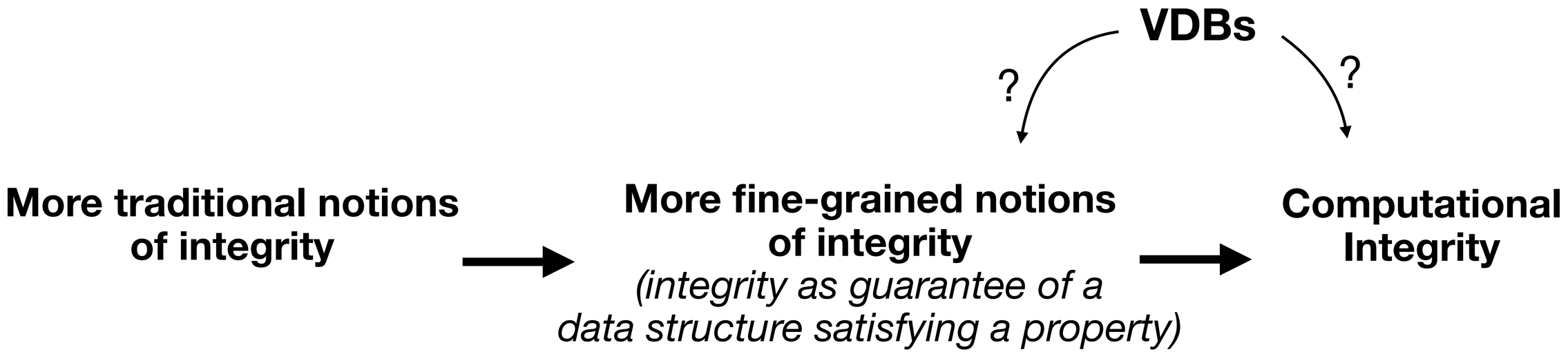
Computational Integrity

General Cryptographic Proofs

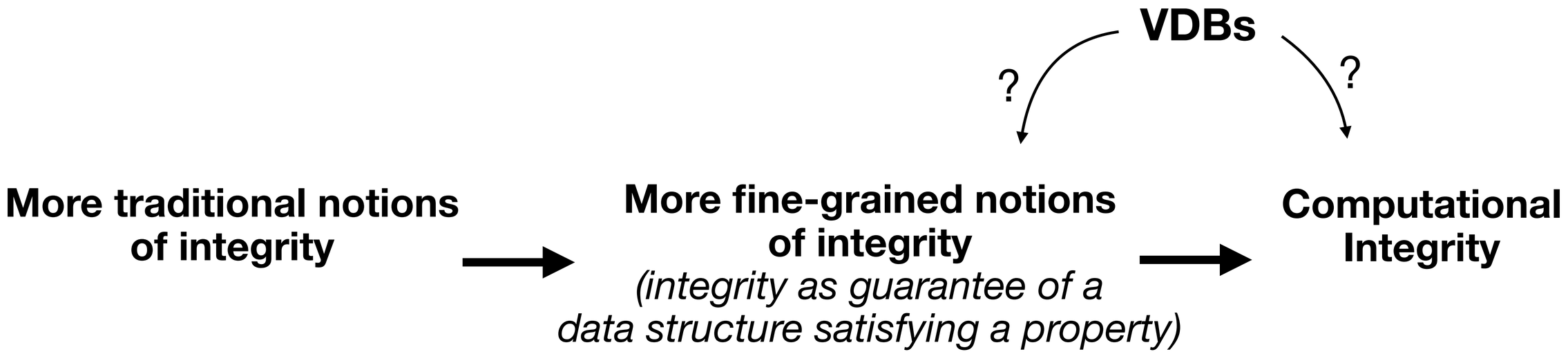
How do we build verifiable databases?



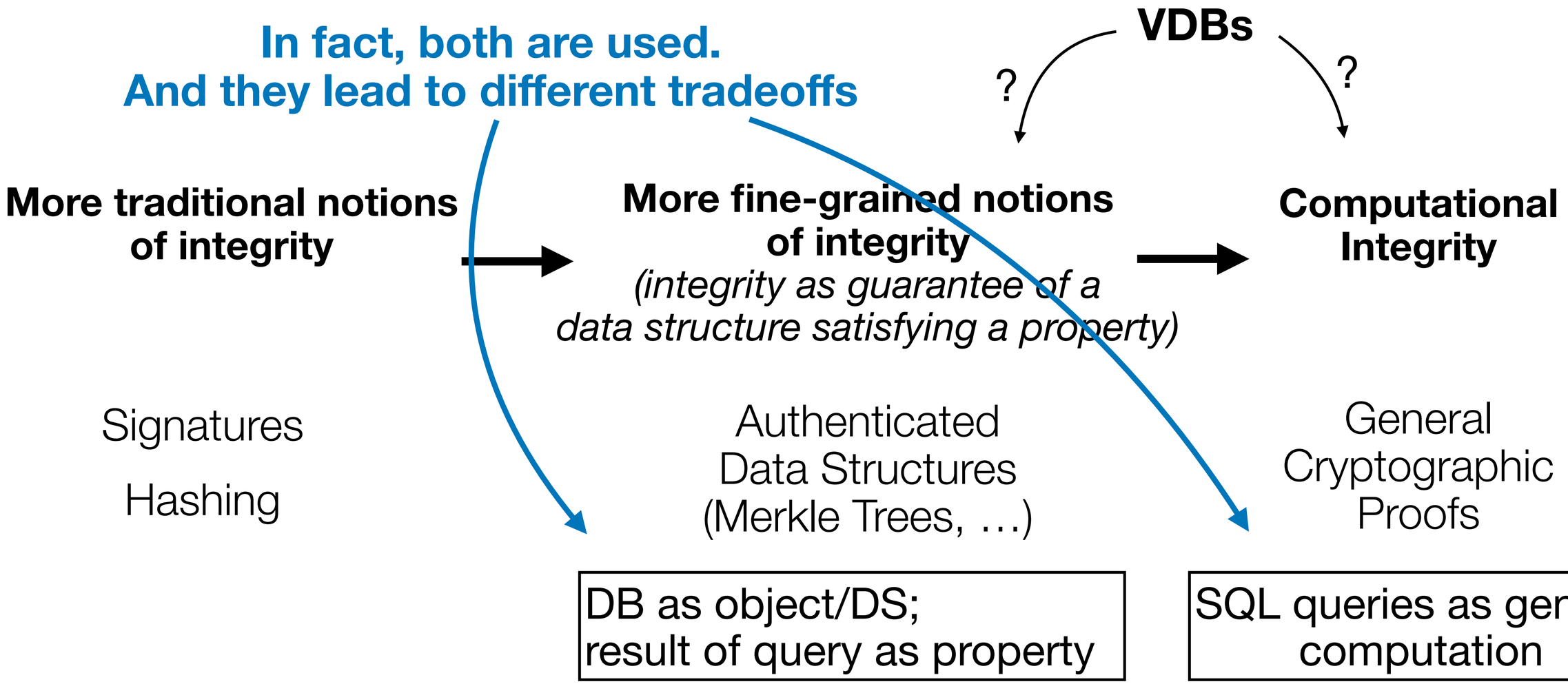
How do we build verifiable databases?



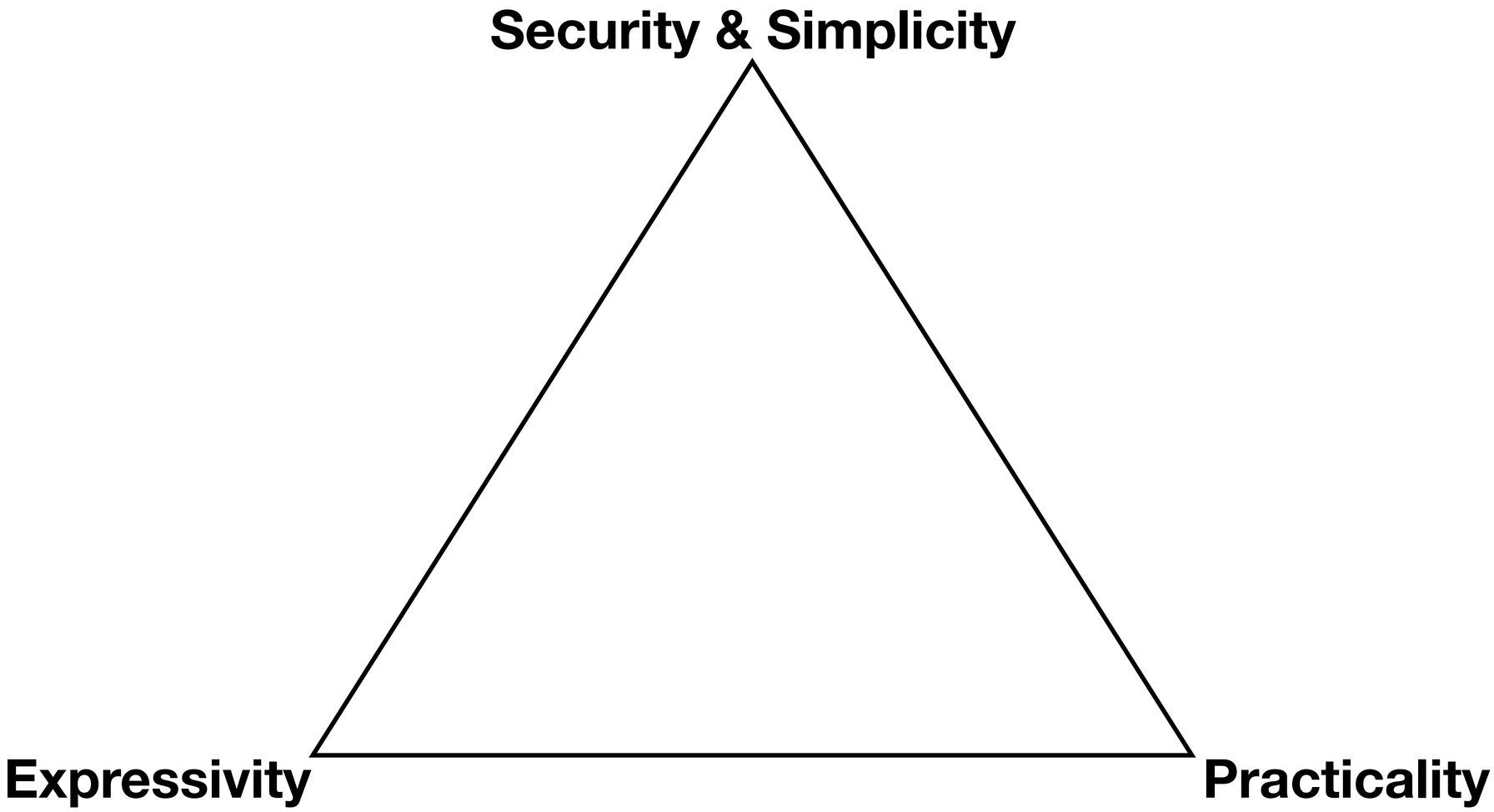
How do we build verifiable databases?



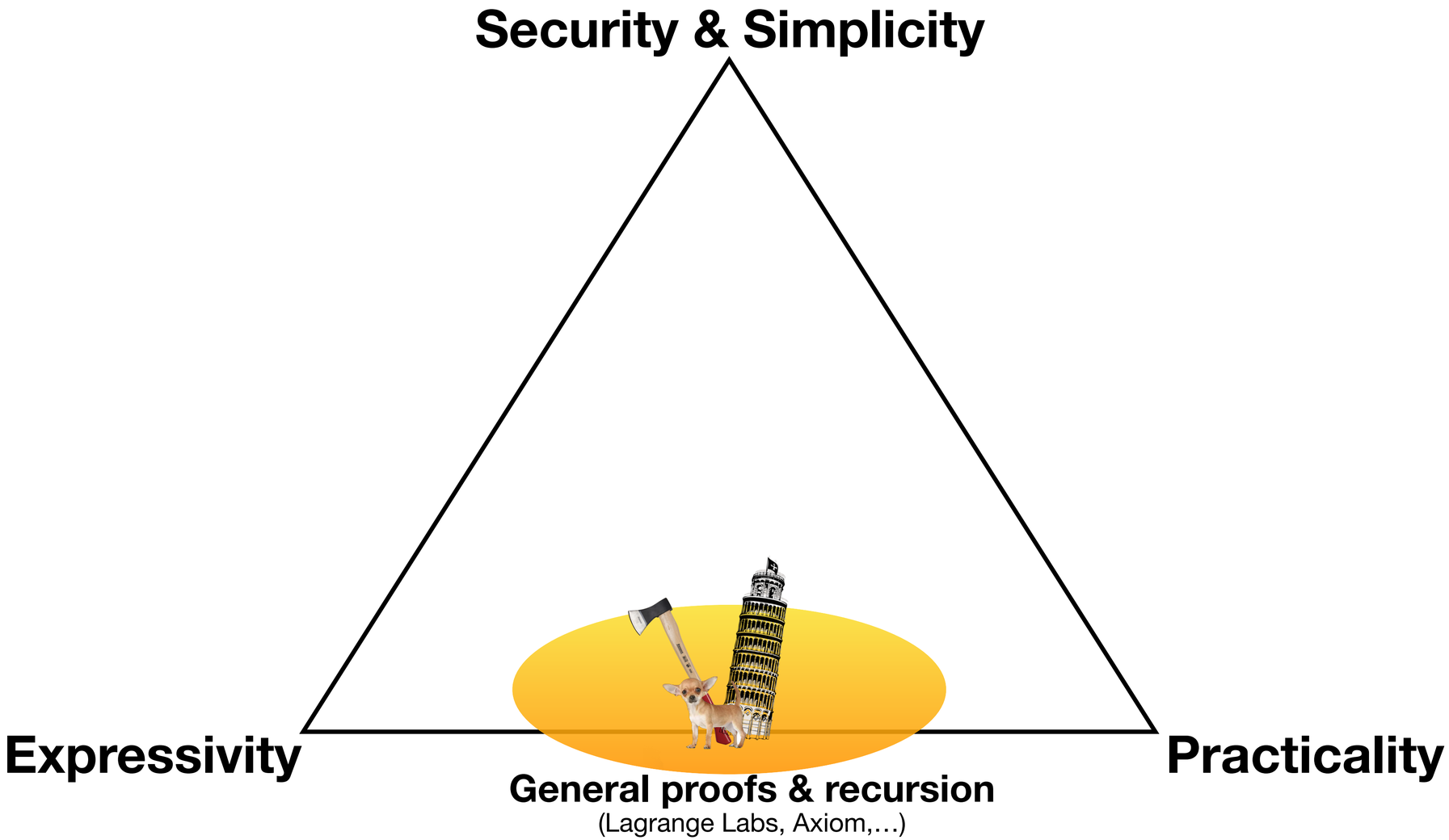
How do we build verifiable databases?



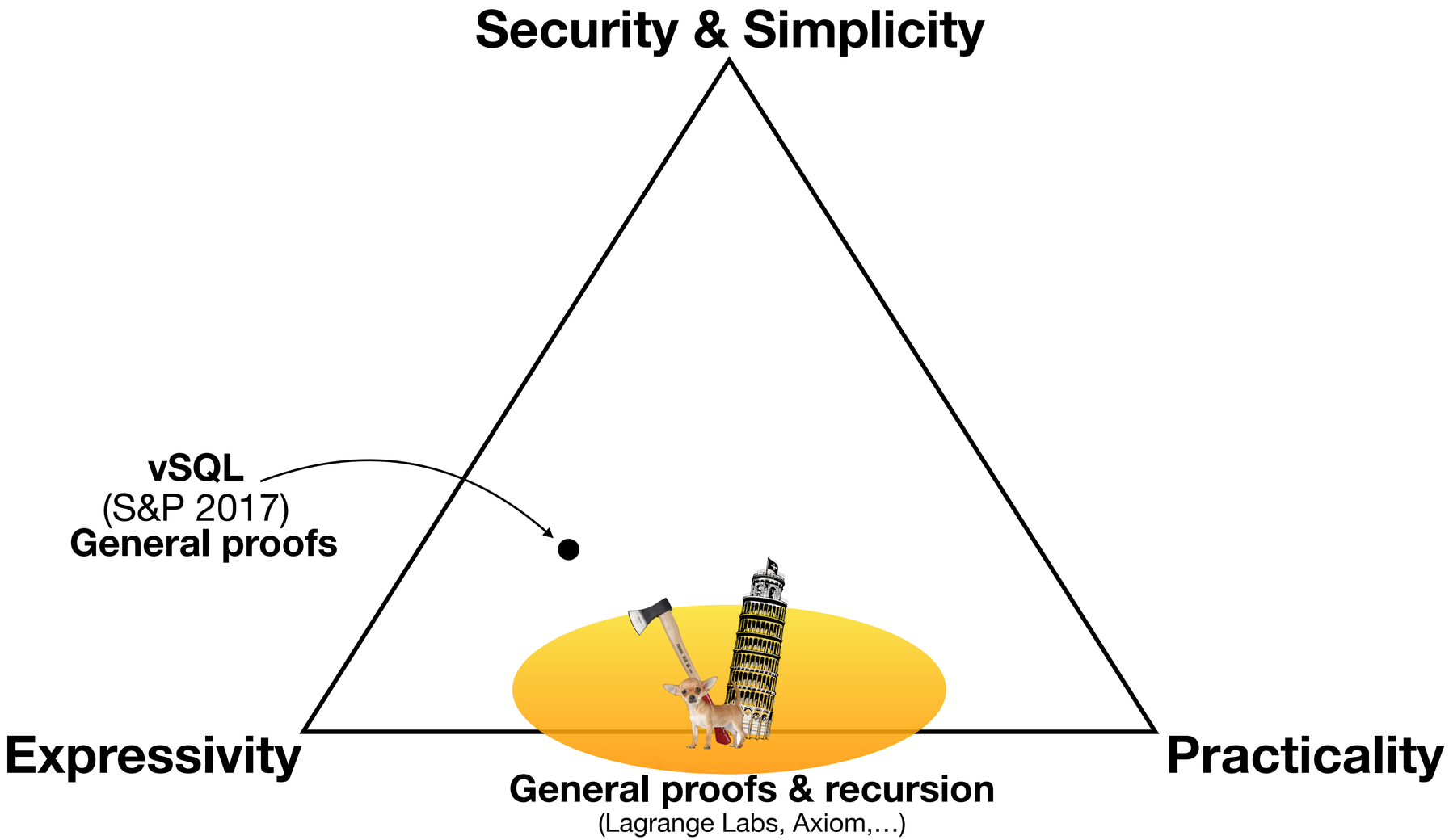
The Landscape of Verifiable DBs



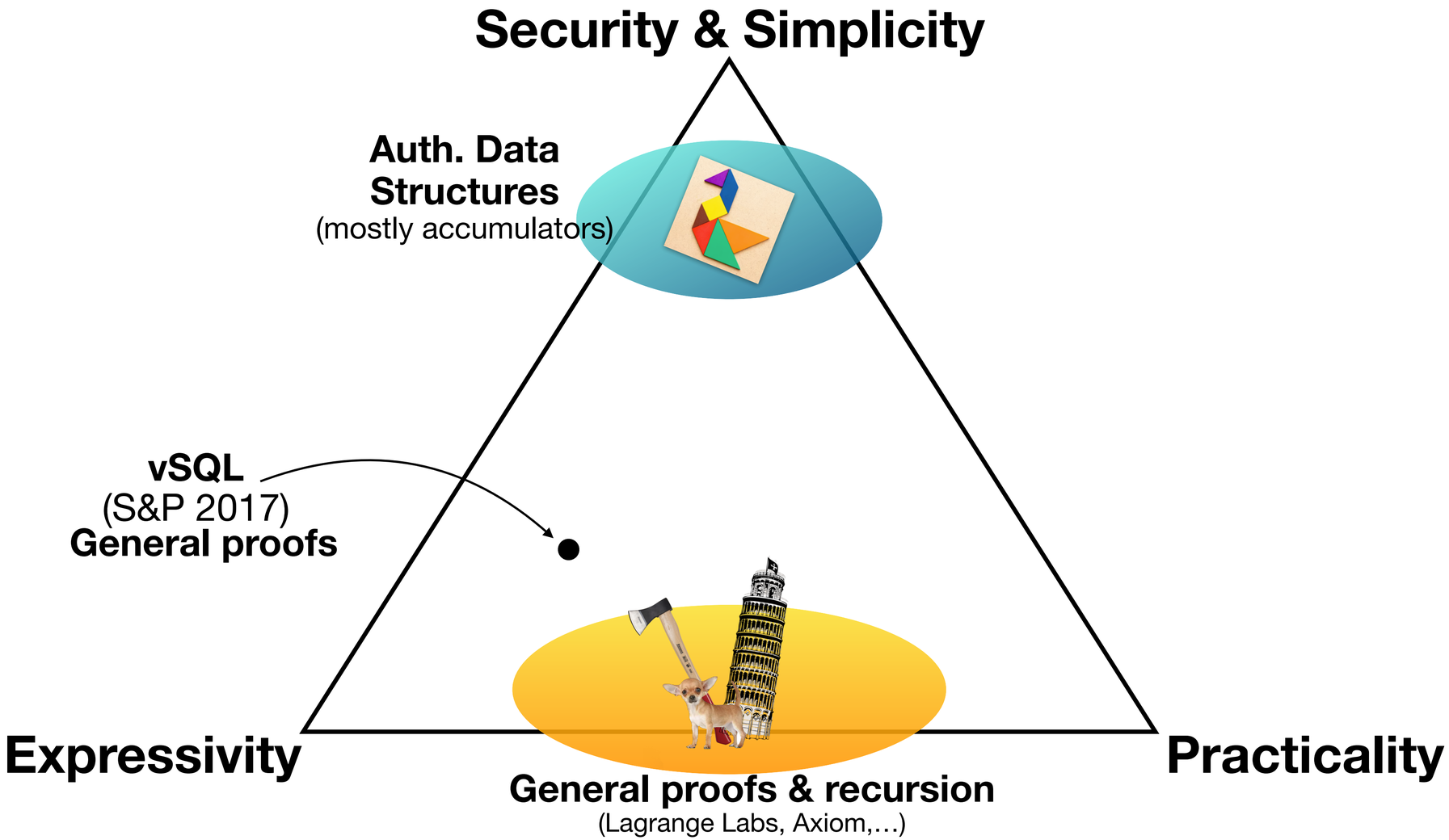
The Landscape of Verifiable DBs



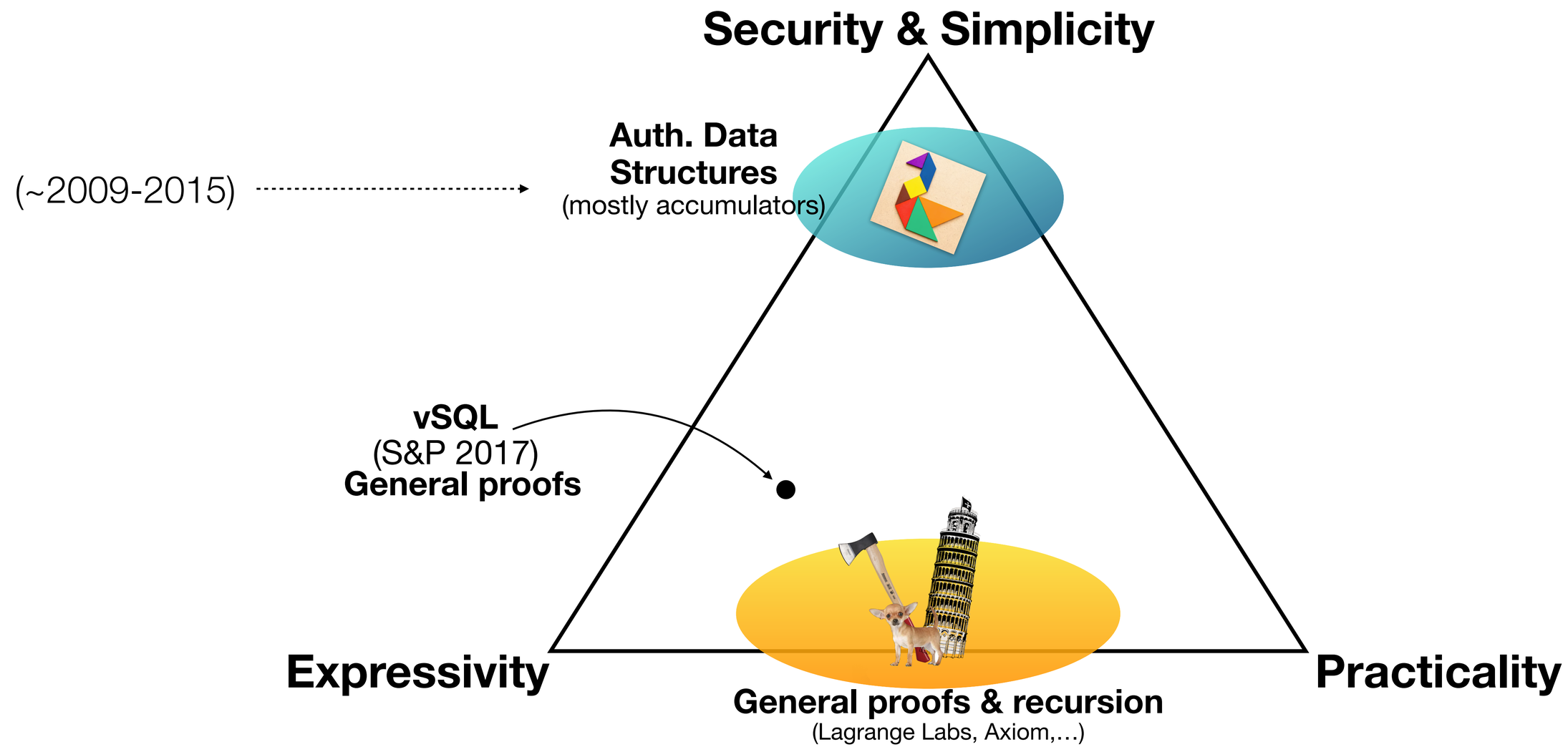
The Landscape of Verifiable DBs



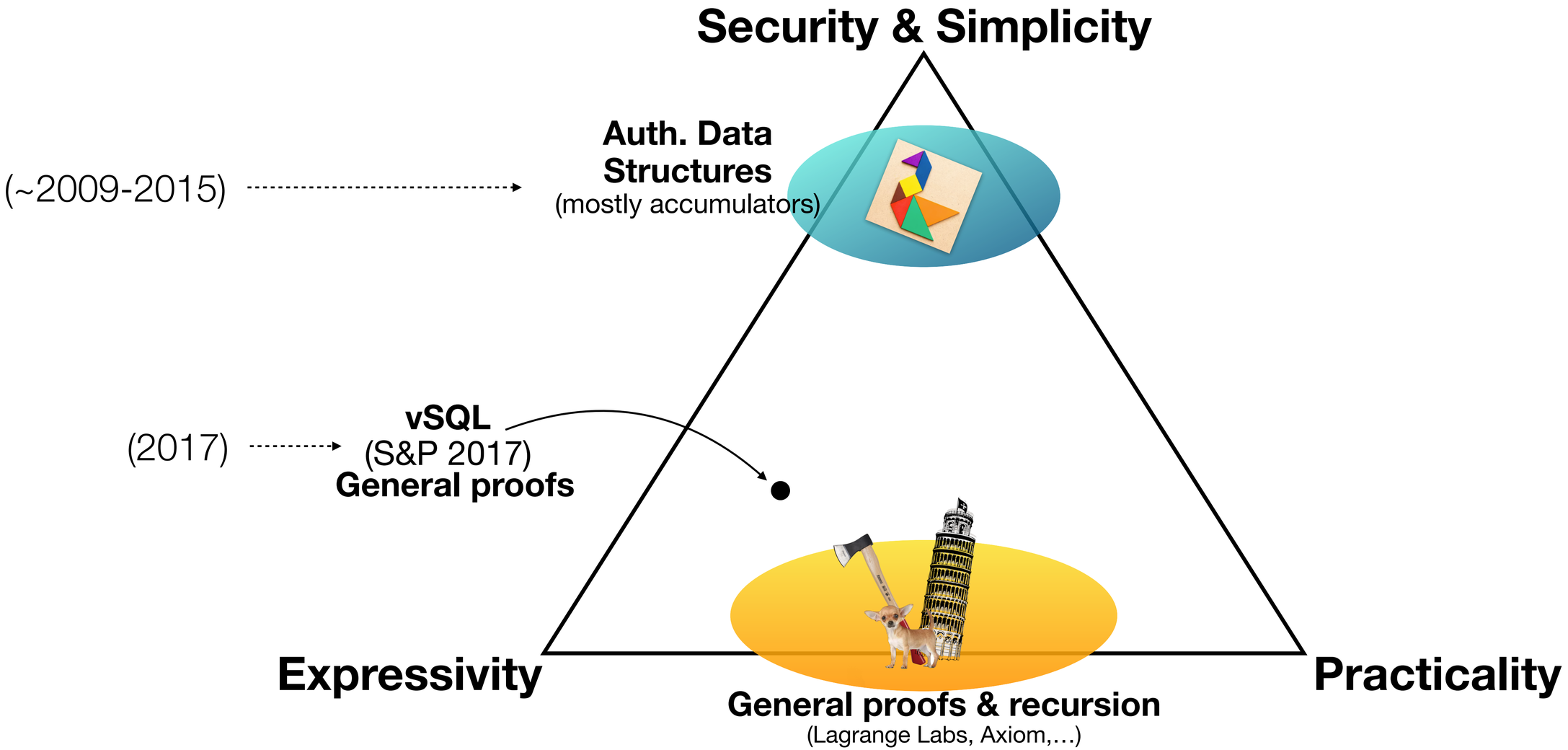
The Landscape of Verifiable DBs



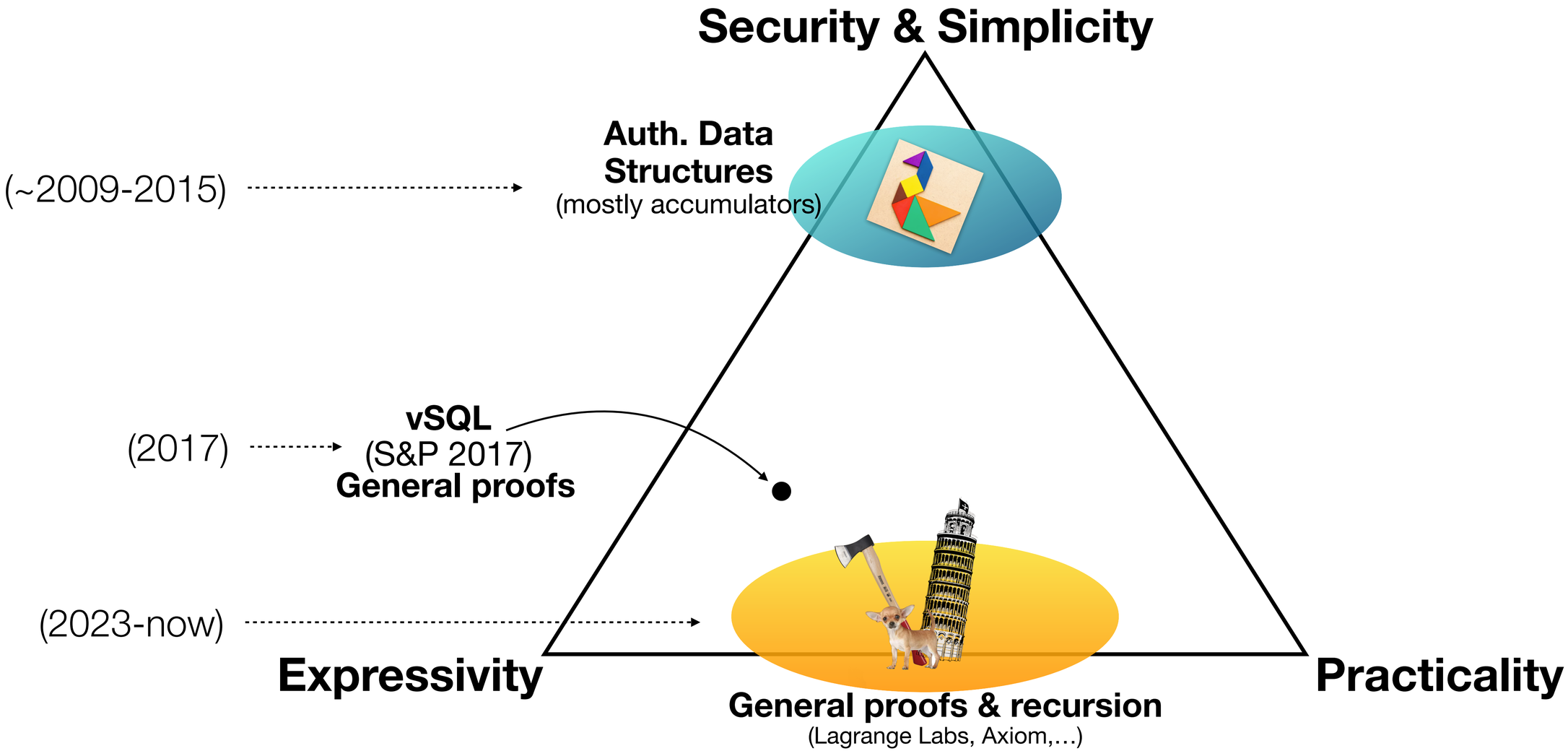
The Landscape of Verifiable DBs



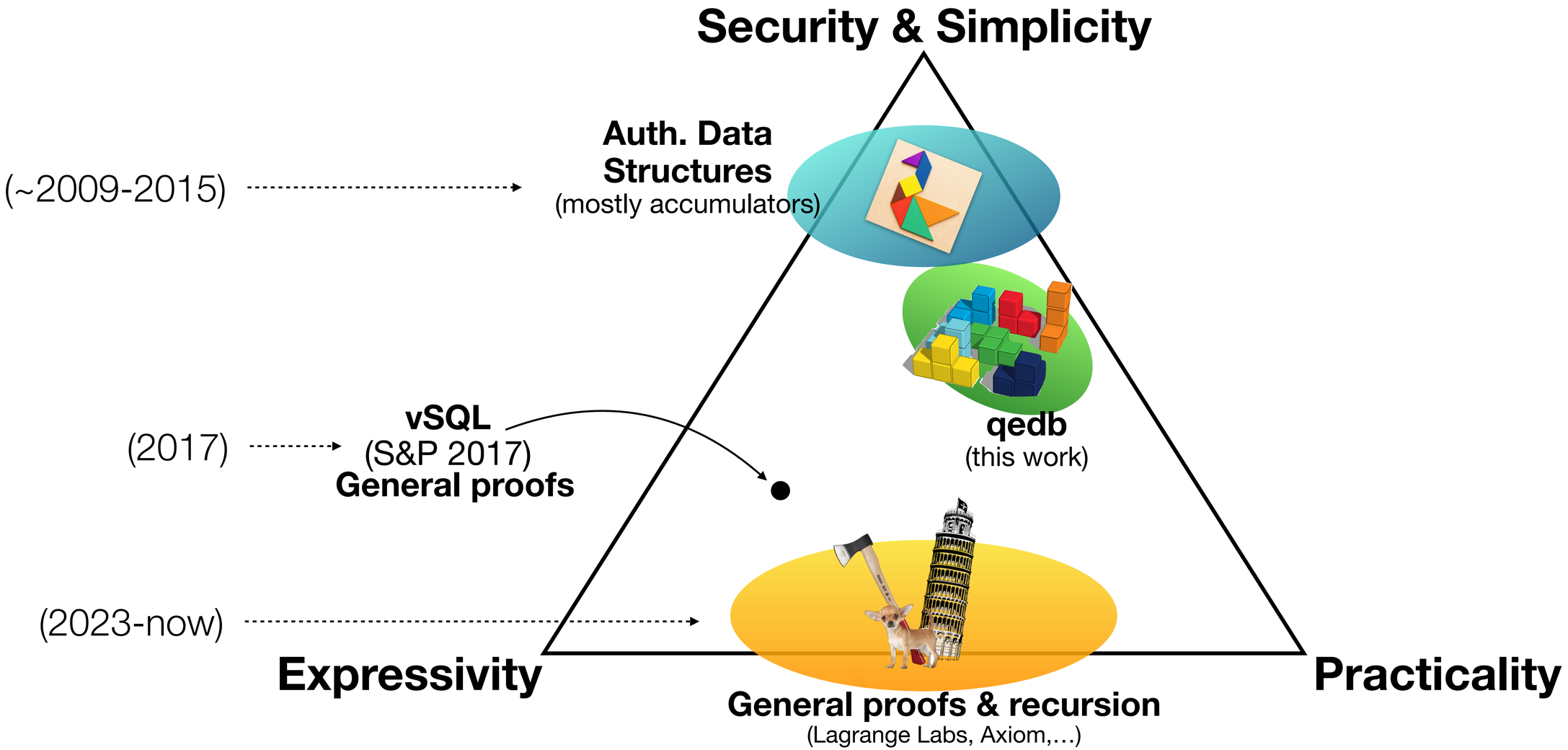
The Landscape of Verifiable DBs



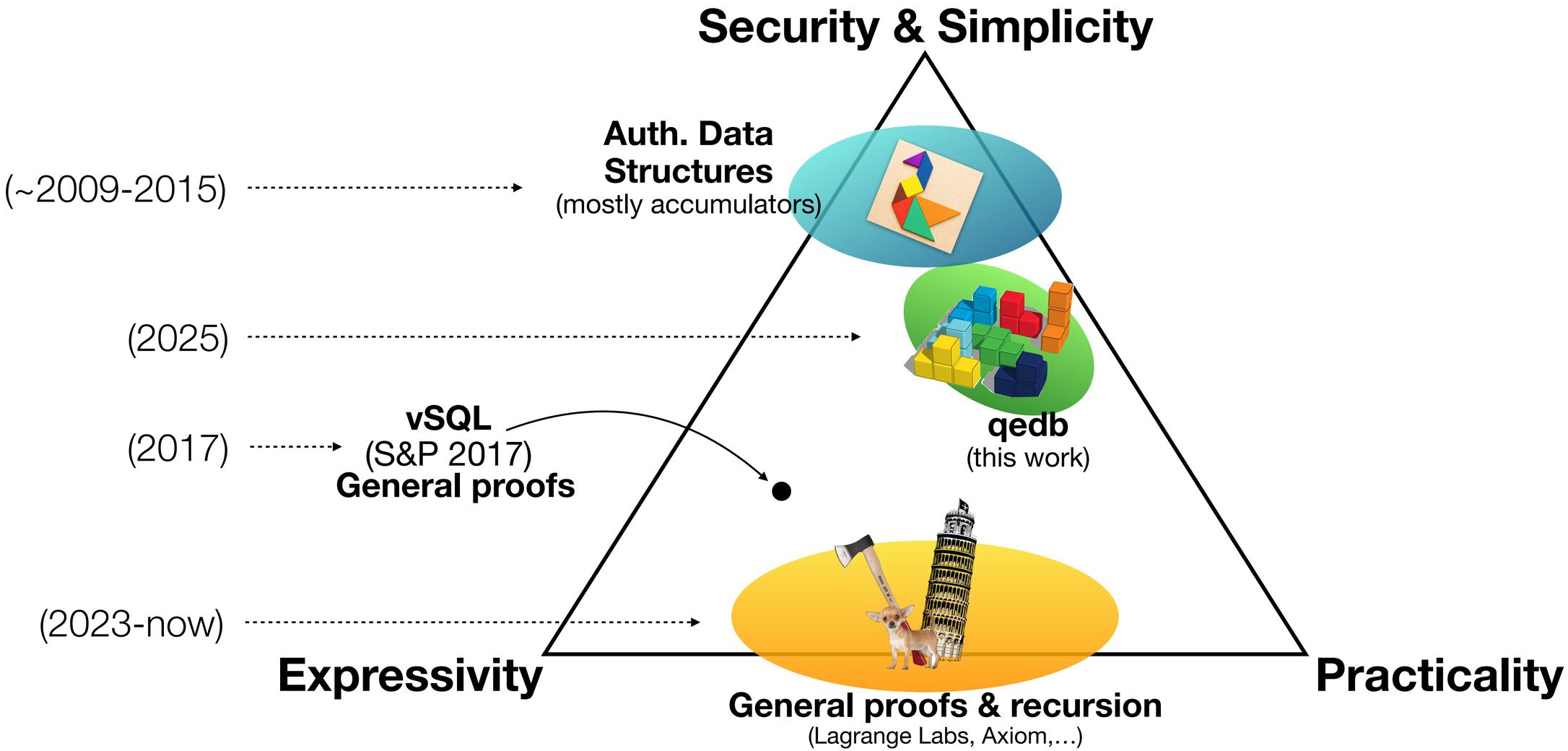
The Landscape of Verifiable DBs



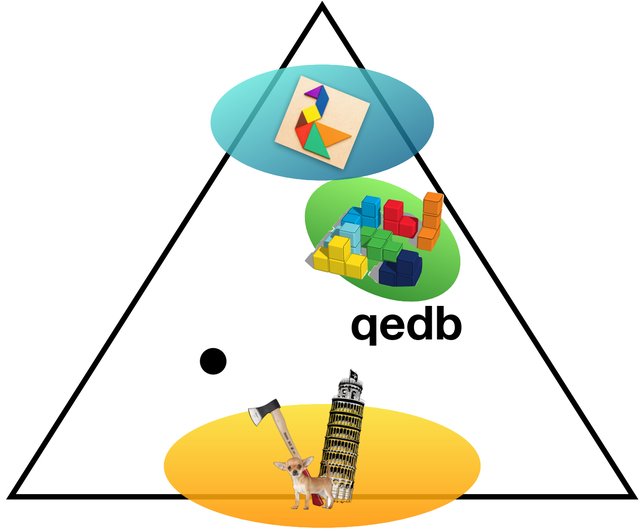
The Landscape of Verifiable DBs



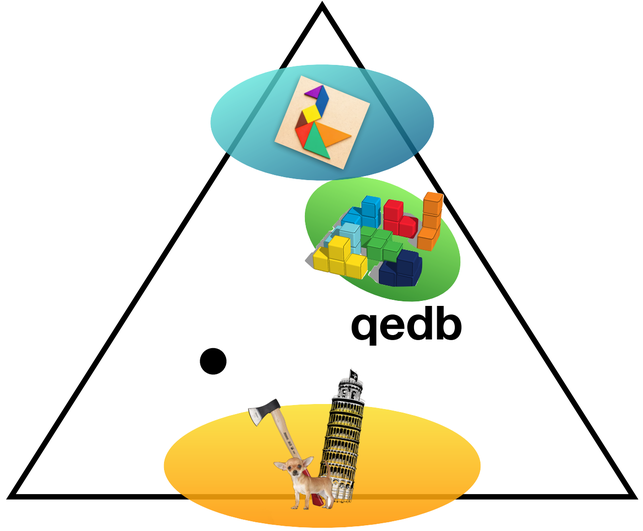
The Landscape of Verifiable DBs



This Talk's Thesis

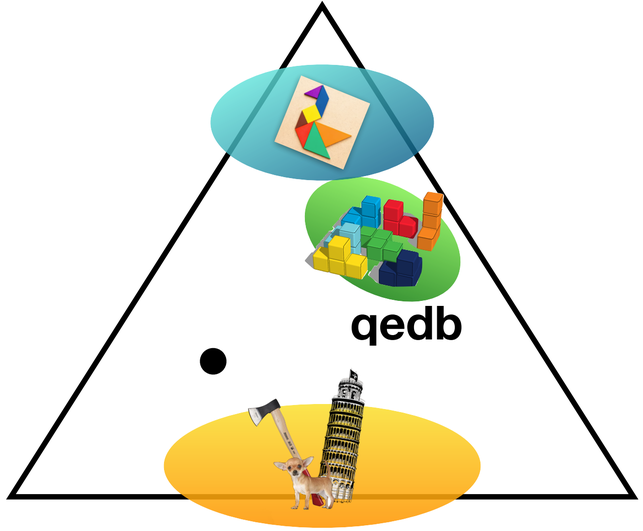


This Talk's Thesis



VDBs can be simple, expressive and efficient

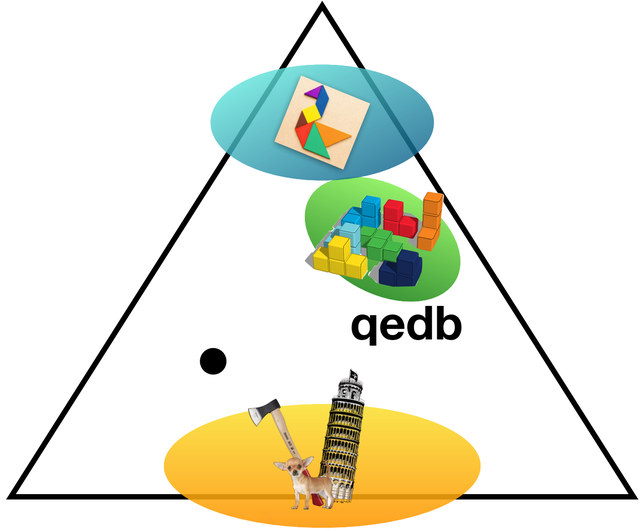
This Talk's Thesis



VDBs can be simple, expressive and efficient

- New construction for the SQL setting (without SNARKs)

This Talk's Thesis

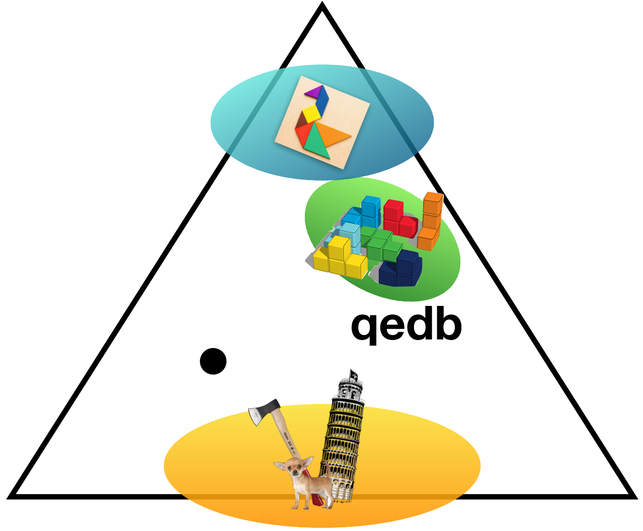


VDBs can be simple, expressive and efficient

- New construction for the SQL setting (without SNARKs)
  - qedb*

* qedb is a recursive acronym standing for "qedb error-checks databases". It is also a shameless pun on it being a *proof* system for DBs.

This Talk's Thesis

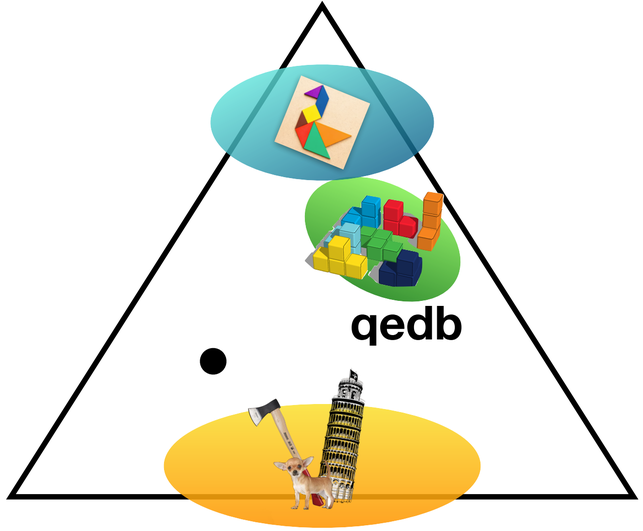


VDBs can be simple, expressive and efficient

- New construction for the SQL setting (without SNARKs)
  - qedb*
- New techniques

* qedb is a recursive acronym standing for "qedb error-checks databases". It is also a shameless pun on it being a *proof* system for DBs.

This Talk's Thesis

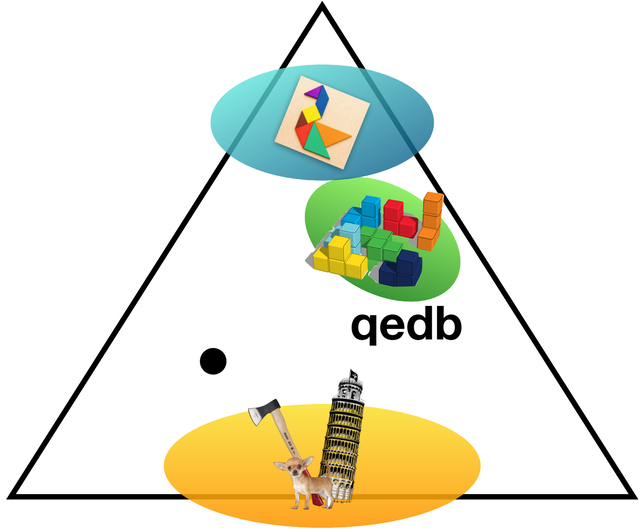


VDBs can be simple, expressive and efficient

- New construction for the SQL setting (without SNARKs)
  - qedb*
- New techniques
- New foundations

* qedb is a recursive acronym standing for "qedb error-checks databases". It is also a shameless pun on it being a *proof* system for DBs.

This Talk's Thesis



VDBs can be simple, expressive and efficient

- New construction for the SQL setting (without SNARKs)
  - qedb*
- New techniques
- New foundations

qedb is a joint work with
V. Botta, S. Bottoni, E. Ragnoli, A. Trombetta.

* qedb is a recursive acronym standing for "qedb error-checks databases".
It is also a shameless pun on it being a *proof* system for DBs.

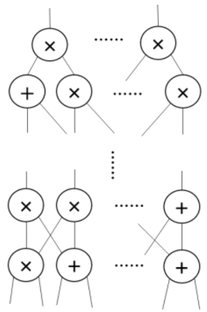
Zooming in on General-Purpose Solutions

Some Preliminaries on General-Purpose Proofs

Some Preliminaries on General-Purpose Proofs

(~2010s)

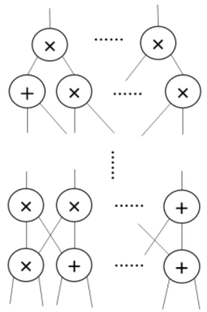
Many advancements (circuits-based)



Some Preliminaries on General-Purpose Proofs

(~2010s)

Many advancements (circuits-based)

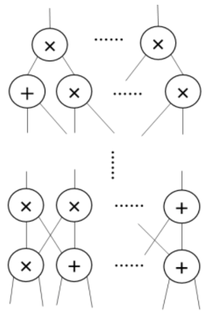


Great **proof size**: < 0.2 KB

Some Preliminaries on General-Purpose Proofs

(~2010s)

Many advancements (circuits-based)



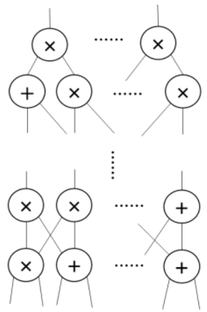
Great **proof size**: < 0.2 KB

Great **verification time**: few ms

Some Preliminaries on General-Purpose Proofs

(~2010s)

Many advancements (circuits-based)



Great **proof size**: < 0.2 KB

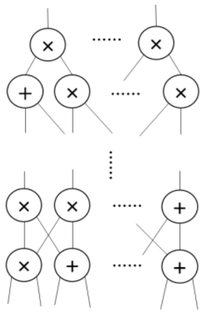
Great **verification time**: few ms

OK **proving time**: e.g., ~ 2 minutes to prove SHA256 of a 15KB file

Some Preliminaries on General-Purpose Proofs

(~2010s)

Many advancements (circuits-based)



Great **proof size**: < 0.2 KB

Great **verification time**: few ms

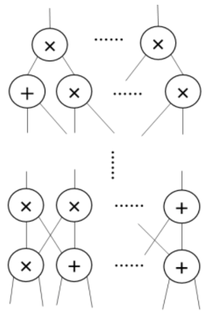
few 10s of seconds with GPU

OK **proving time**: e.g., ~ 2 minutes to prove SHA256 of a 15KB file

Some Preliminaries on General-Purpose Proofs

(~2010s)

Many advancements (circuits-based)



Great **proof size**: < 0.2 KB

Great **verification time**: few ms

few 10s of seconds with GPU

OK **proving time**: e.g., ~ 2 minutes to prove SHA256 of a 15KB file

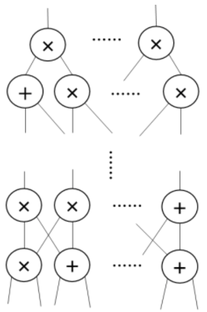
small proofs
worse proving

larger proofs
better proving

Some Preliminaries on General-Purpose Proofs

(~2010s)

Many advancements (circuits-based)



* Great **proof size**: < 0.2 KB

* Great **verification time**: few ms

few 10s of seconds with GPU

* OK **proving time**: e.g., ~ 2 minutes to prove SHA256 of a 15KB file

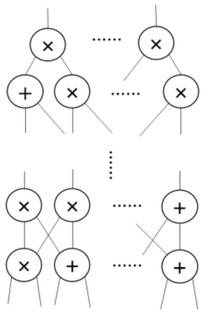
small proofs worse proving *

larger proofs better proving

Some Preliminaries on General-Purpose Proofs

(~2010s)

Many advancements (circuits-based)



* Great **proof size**: < 0.2 KB

Great **verification time**: few ms

OK **proving time**: e.g., ~ 2 minutes to prove SHA256 of a 15KB file

few 10s of seconds with GPU

small proofs worse proving *

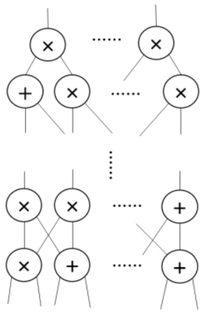
larger proofs better proving

Main pain points:

Some Preliminaries on General-Purpose Proofs

(~2010s)

Many advancements (circuits-based)



* Great **proof size**: < 0.2 KB

Great **verification time**: few ms

OK **proving time**: e.g., ~ 2 minutes to prove SHA256 of a 15KB file

few 10s of seconds with GPU

small proofs worse proving *

larger proofs better proving

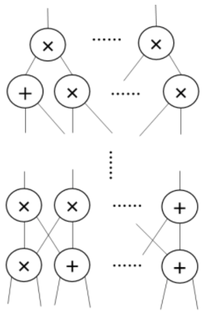
Main pain points:

- prover hard to parallelize + high-memory

Some Preliminaries on General-Purpose Proofs

(~2010s)

Many advancements (circuits-based)



* Great **proof size**: < 0.2 KB

Great **verification time**: few ms

OK **proving time**: e.g., ~ 2 minutes to prove SHA256 of a 15KB file

few 10s of seconds with GPU

small proofs
worse proving
*

larger proofs
better proving

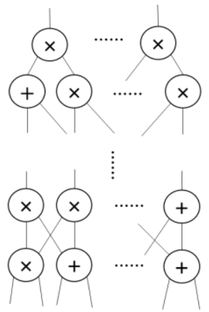
Main pain points:

- prover hard to parallelize + high-memory
- developer experience (circuits)

Some Preliminaries on General-Purpose Proofs

(~2010s)

Many advancements (circuits-based)



(from ~2022)

From circuits to virtual machines (through recursion)

* Great **proof size**: < 0.2 KB

Great **verification time**: few ms

OK **proving time**: e.g., ~ 2 minutes to prove SHA256 of a 15KB file

few 10s of seconds with GPU

small proofs worse proving *

larger proofs better proving

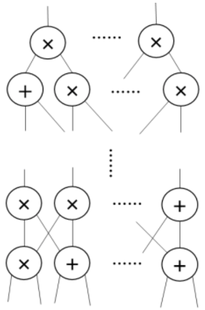
Main pain points:

- prover hard to parallelize + high-memory
- developer experience (circuits)

Some Preliminaries on General-Purpose Proofs

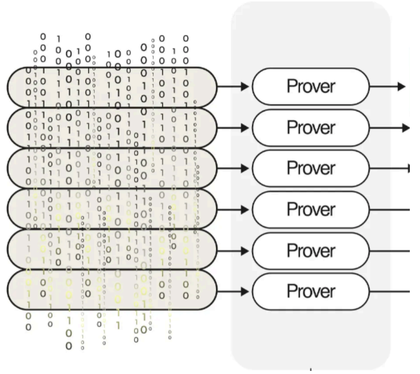
(~2010s)

Many advancements (circuits-based)



(from ~2022)

From circuits to virtual machines (through recursion)



* Great **proof size**: < 0.2 KB

Great **verification time**: few ms

OK **proving time**: e.g., ~ 2 minutes to prove SHA256 of a 15KB file

few 10s of seconds with GPU

small proofs
worse proving
*

larger proofs
better proving

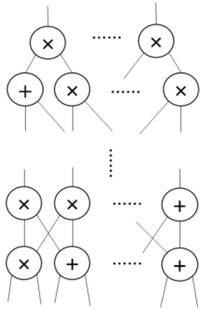
Main pain points:

- prover hard to parallelize + high-memory
- developer experience (circuits)

Some Preliminaries on General-Purpose Proofs

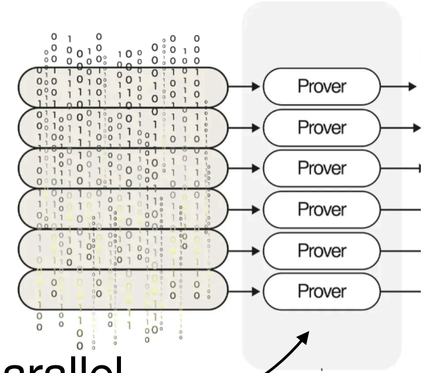
(~2010s)

Many advancements (circuits-based)



(from ~2022)

From circuits to virtual machines (through recursion)



parallel proving

* Great **proof size**: < 0.2 KB

Great **verification time**: few ms

OK **proving time**: e.g., ~ 2 minutes to prove SHA256 of a 15KB file

few 10s of seconds with GPU

small proofs worse proving *

larger proofs better proving

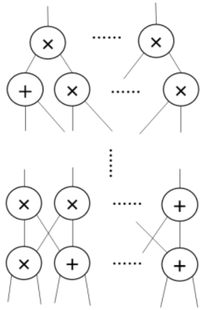
Main pain points:

- prover hard to parallelize + high-memory
- developer experience (circuits)

Some Preliminaries on General-Purpose Proofs

(~2010s)

Many advancements (circuits-based)



* Great **proof size**: < 0.2 KB

Great **verification time**: few ms

OK **proving time**: e.g., ~ 2 minutes to prove SHA256 of a 15KB file

few 10s of seconds with GPU

small proofs worse proving *

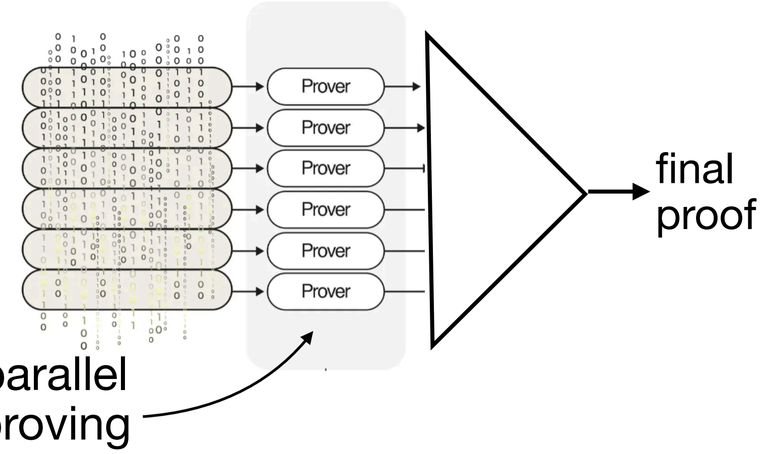
larger proofs better proving

Main pain points:

- prover hard to parallelize + high-memory
- developer experience (circuits)

(from ~2022)

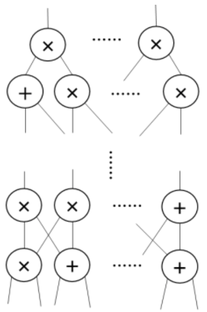
From circuits to virtual machines (through recursion)



Some Preliminaries on General-Purpose Proofs

(~2010s)

Many advancements (circuits-based)



* Great **proof size**: < 0.2 KB

Great **verification time**: few ms

OK **proving time**: e.g., ~ 2 minutes to prove SHA256 of a 15KB file

few 10s of seconds with GPU

small proofs worse proving *

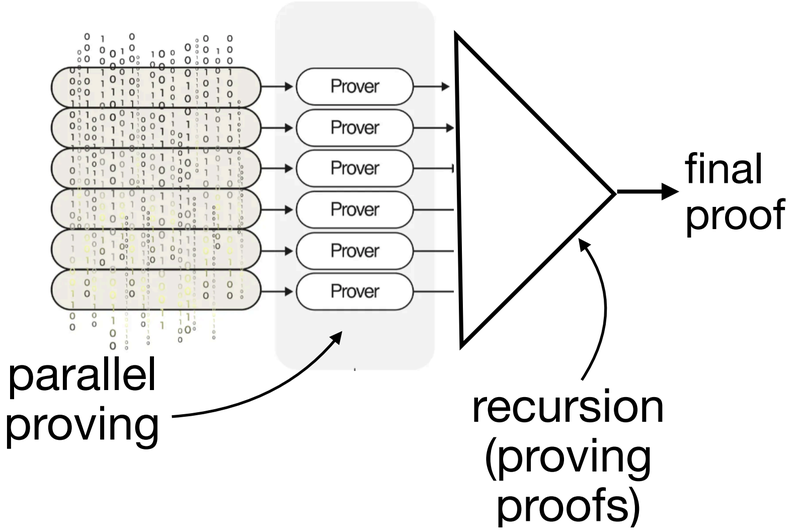
larger proofs better proving

Main pain points:

- prover hard to parallelize + high-memory
- developer experience (circuits)

(from ~2022)

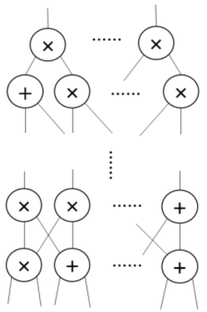
From circuits to virtual machines (through recursion)



Some Preliminaries on General-Purpose Proofs

(~2010s)

Many advancements (circuits-based)



* Great **proof size**: < 0.2 KB

Great **verification time**: few ms

OK **proving time**: e.g., ~ 2 minutes to prove SHA256 of a 15KB file

few 10s of seconds with GPU

small proofs worse proving *

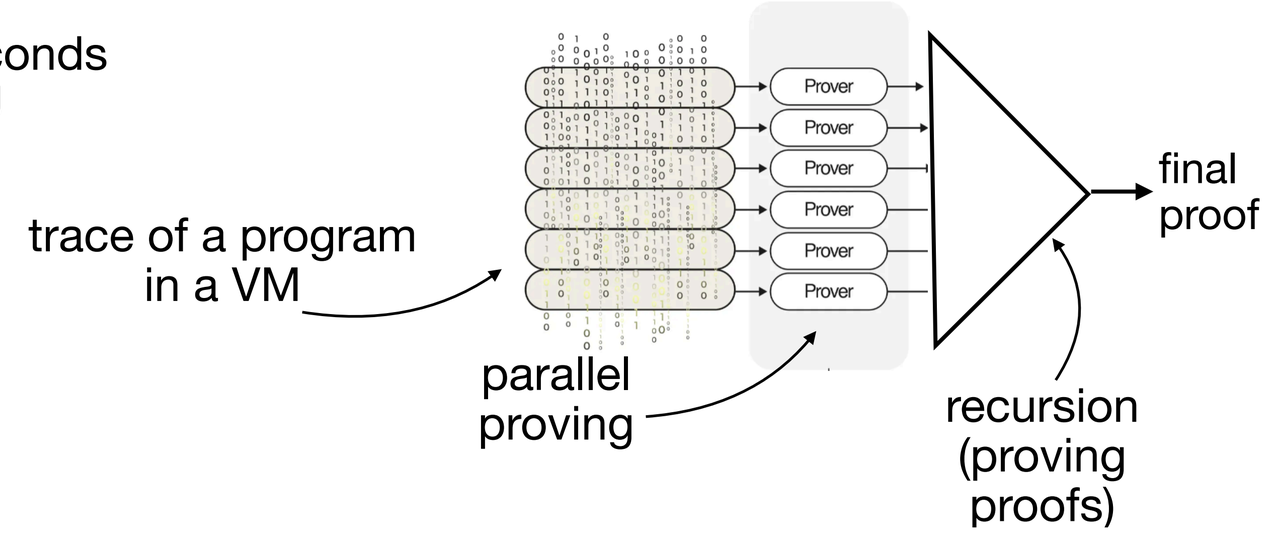
larger proofs better proving

Main pain points:

- prover hard to parallelize + high-memory
- developer experience (circuits)

(from ~2022)

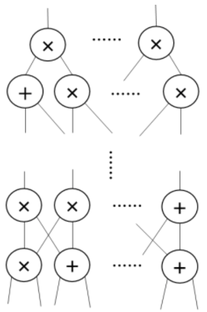
From circuits to virtual machines (through recursion)



Some Preliminaries on General-Purpose Proofs

(~2010s)

Many advancements (circuits-based)



* Great **proof size**: < 0.2 KB

Great **verification time**: few ms

OK **proving time**: e.g., ~ 2 minutes to prove SHA256 of a 15KB file

few 10s of seconds with GPU

small proofs worse proving *

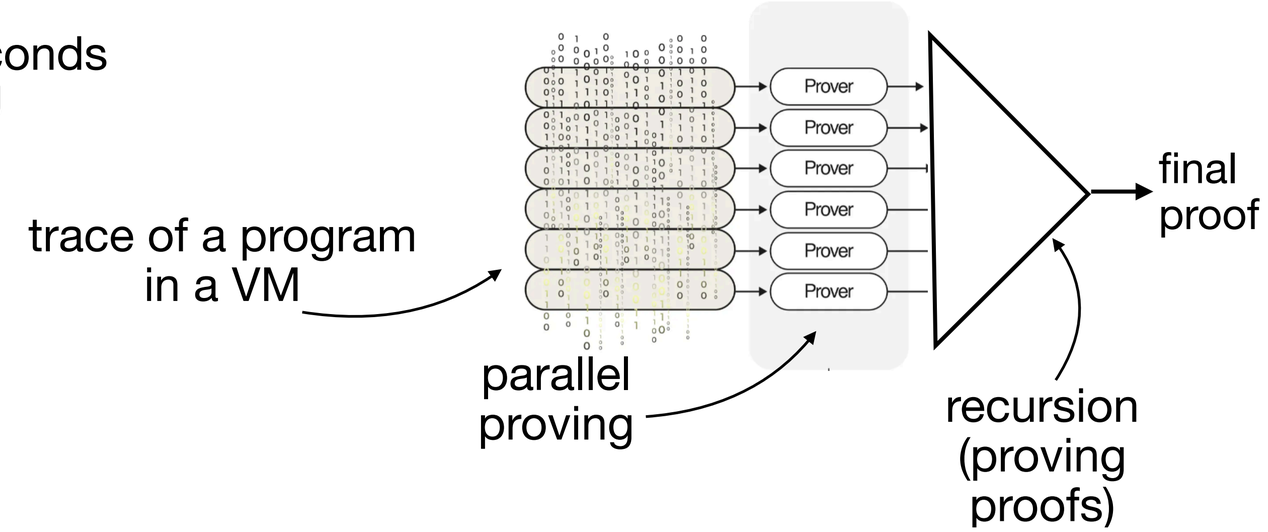
larger proofs better proving

Main pain points:

- prover hard to parallelize + high-memory
- developer experience (circuits)

(from ~2022)

From circuits to virtual machines (through recursion)



Because of the VM trace, developers can just write code (e.g., Rust) that gets compiled to the VM (instead of circuits)

vSQL

[IEEE S&P 2017]

vSQL
[IEEE S&P 2017]

- **Key observation of vSQL:**

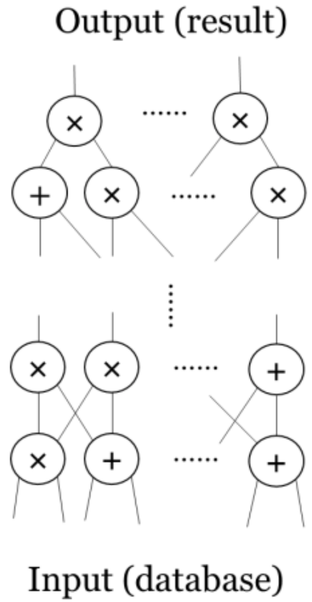
vSQL

[IEEE S&P 2017]

- **Key observation of vSQL:**
  - to save on proving time, use a “lightweight” general-purpose proof system (based on circuits) and customize it a little bit to the DB setting

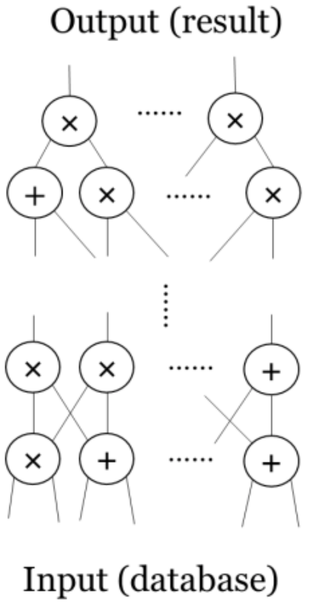
vSQL
[IEEE S&P 2017]

- **Key observation of vSQL:**
  - to save on proving time, use a “lightweight” general-purpose proof system (based on circuits) and customize it a little bit to the DB setting
    - [for the cryptographers: essentially “GKR specialized to DBs”]



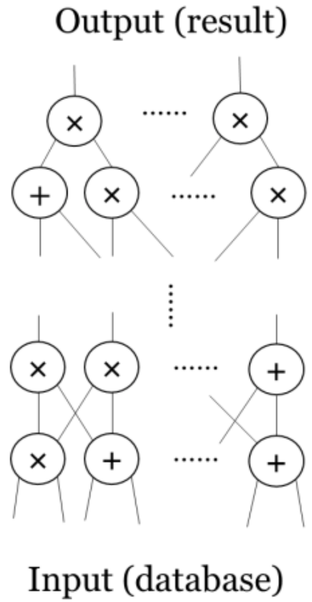
vSQL
[IEEE S&P 2017]

- **Key observation of vSQL:**
  - to save on proving time, use a “lightweight” general-purpose proof system (based on circuits) and customize it a little bit to the DB setting
    - [for the cryptographers: essentially “GKR specialized to DBs”]
- At the time, this was a big improvement on the proving performance of other solutions [e.g. Groth16]



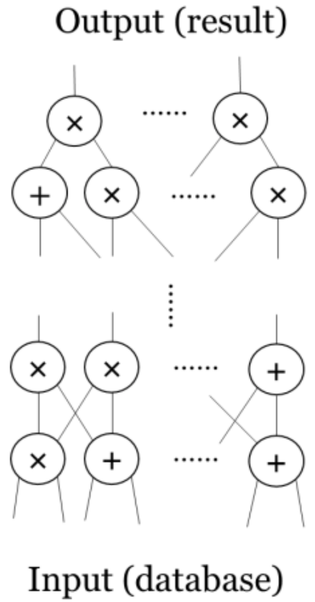
vSQL
[IEEE S&P 2017]

- **Key observation of vSQL:**
  - to save on proving time, use a “lightweight” general-purpose proof system (based on circuits) and customize it a little bit to the DB setting
    - [for the cryptographers: essentially “GKR specialized to DBs”]
- At the time, this was a big improvement on the proving performance of other solutions [e.g. Groth16]
- vSQL is very expressive (any SQL query); based on standard tools (e.g. [PST])



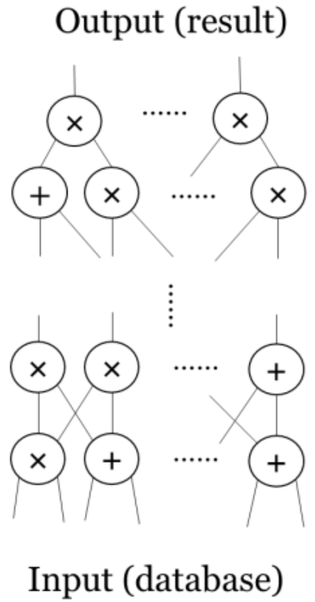
vSQL
[IEEE S&P 2017]

- **Key observation of vSQL:**
  - to save on proving time, use a “lightweight” general-purpose proof system (based on circuits) and customize it a little bit to the DB setting
    - [for the cryptographers: essentially “GKR specialized to DBs”]
- At the time, this was a big improvement on the proving performance of other solutions [e.g. Groth16]
- vSQL is very expressive (any SQL query); based on standard tools (e.g. [PST])
- **Drawbacks:**



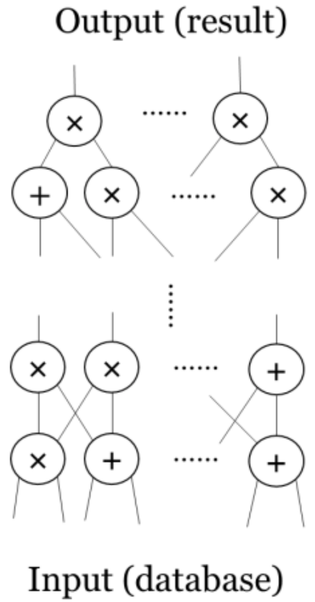
vSQL
[IEEE S&P 2017]

- **Key observation of vSQL:**
  - to save on proving time, use a “lightweight” general-purpose proof system (based on circuits) and customize it a little bit to the DB setting
    - [for the cryptographers: essentially “GKR specialized to DBs”]
- At the time, this was a big improvement on the proving performance of other solutions [e.g. Groth16]
- vSQL is very expressive (any SQL query); based on standard tools (e.g. [PST])
- **Drawbacks:**
  - Requires implementing circuits emulating SQL



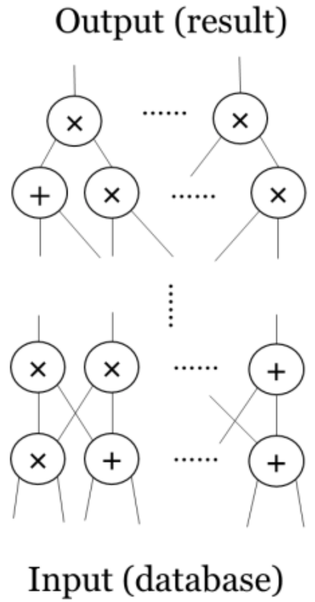
vSQL
[IEEE S&P 2017]

- **Key observation of vSQL:**
  - to save on proving time, use a “lightweight” general-purpose proof system (based on circuits) and customize it a little bit to the DB setting
    - [for the cryptographers: essentially “GKR specialized to DBs”]
- At the time, this was a big improvement on the proving performance of other solutions [e.g. Groth16]
- vSQL is very expressive (any SQL query); based on standard tools (e.g. [PST])
- **Drawbacks:**
  - Requires implementing circuits emulating SQL
    - Not great for developer experience



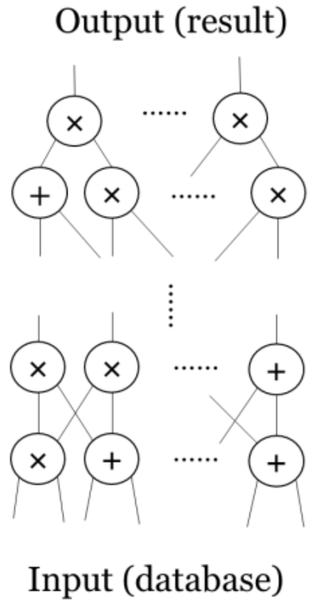
vSQL
[IEEE S&P 2017]

- **Key observation of vSQL:**
  - to save on proving time, use a “lightweight” general-purpose proof system (based on circuits) and customize it a little bit to the DB setting
    - [for the cryptographers: essentially “GKR specialized to DBs”]
- At the time, this was a big improvement on the proving performance of other solutions [e.g. Groth16]
- vSQL is very expressive (any SQL query); based on standard tools (e.g. [PST])
- **Drawbacks:**
  - Requires implementing circuits emulating SQL
    - Not great for developer experience
    - Also cumbersome: a naive set intersection in a circuit has a quadratic overhead

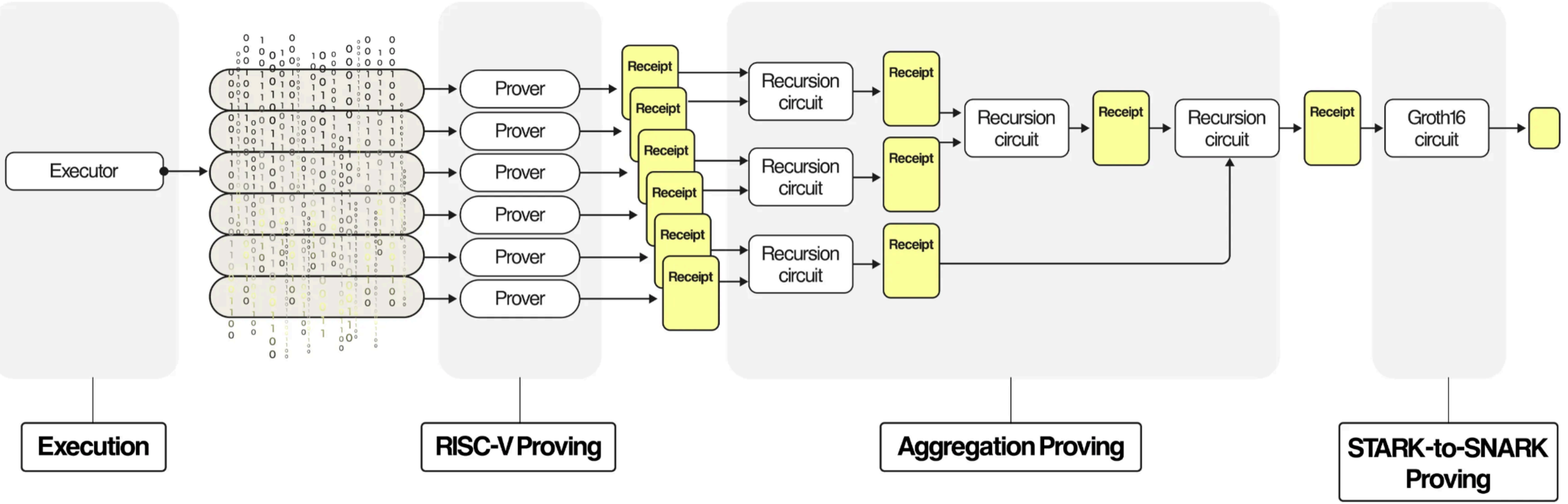


vSQL
[IEEE S&P 2017]

- **Key observation of vSQL:**
  - to save on proving time, use a “lightweight” general-purpose proof system (based on circuits) and customize it a little bit to the DB setting
    - [for the cryptographers: essentially “GKR specialized to DBs”]
- At the time, this was a big improvement on the proving performance of other solutions [e.g. Groth16]
- vSQL is very expressive (any SQL query); based on standard tools (e.g. [PST])
- **Drawbacks:**
  - Requires implementing circuits emulating SQL
    - Not great for developer experience
    - Also cumbersome: a naive set intersection in a circuit has a quadratic overhead
  - Relatively large proof size (100s of KB); other performance limitations

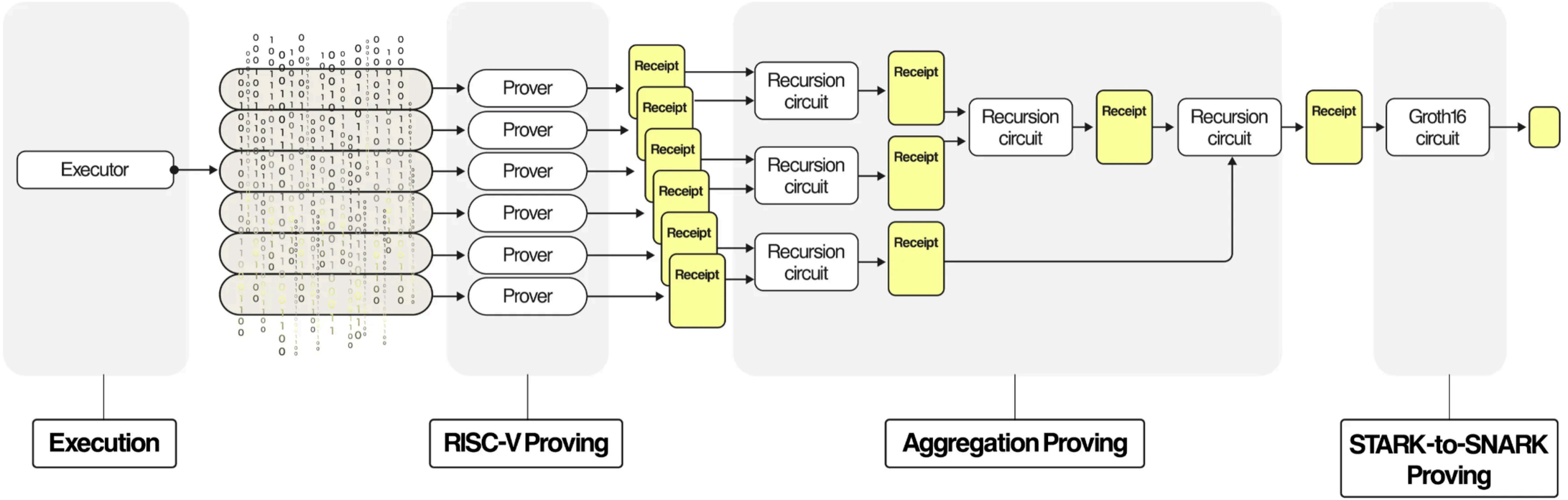


The approach from SNARKs&Recursion



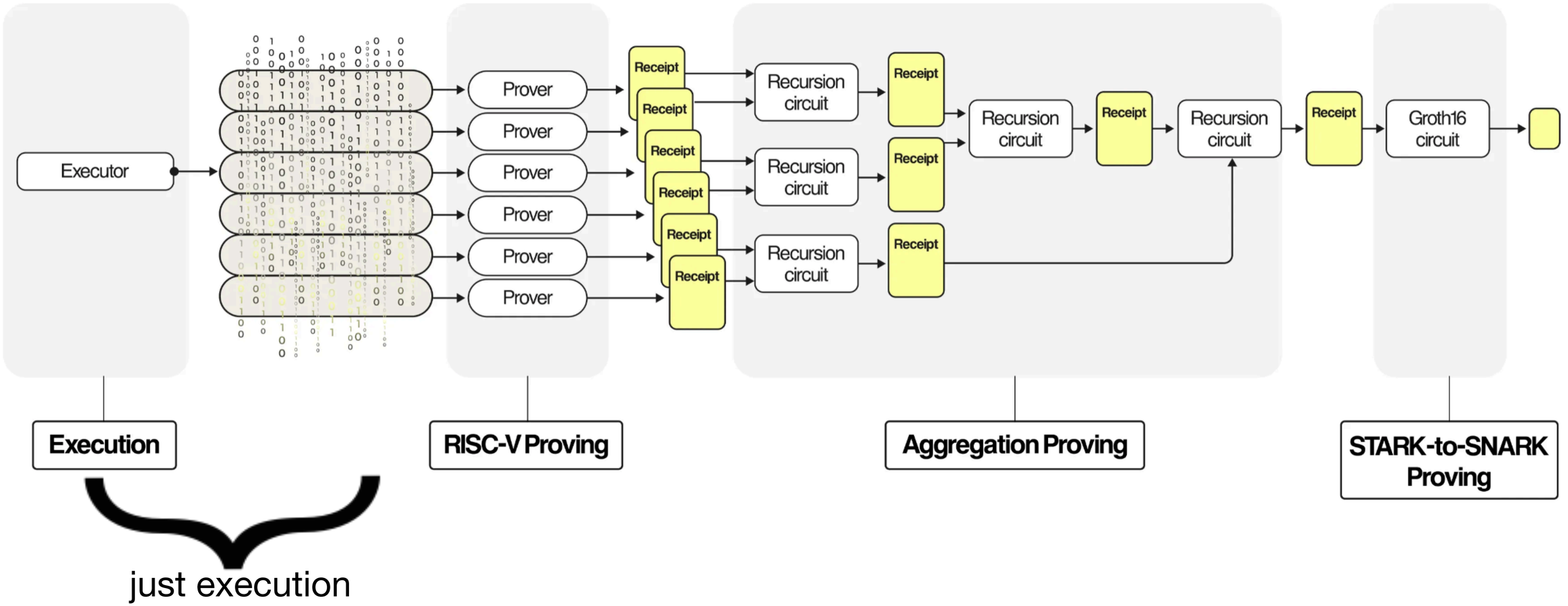
[Image credit: RiscZero]

The approach from SNARKs&Recursion



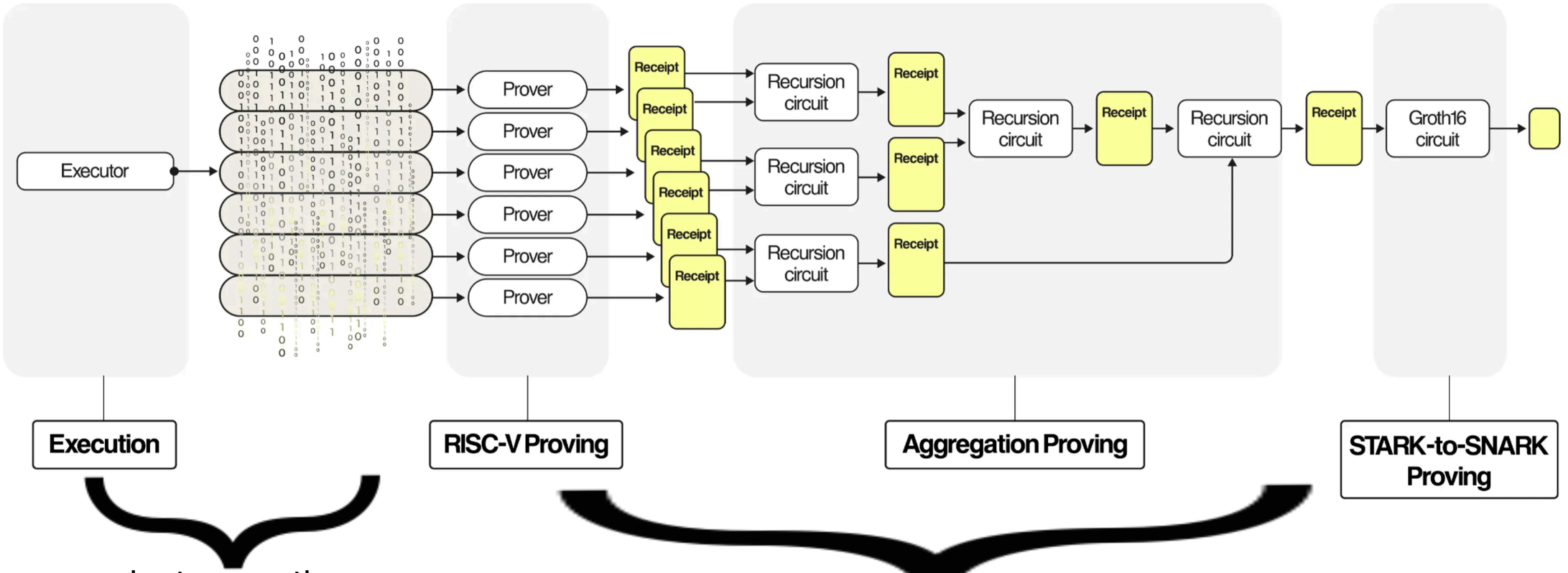
[Image credit: RiscZero]

The approach from SNARKs&Recursion



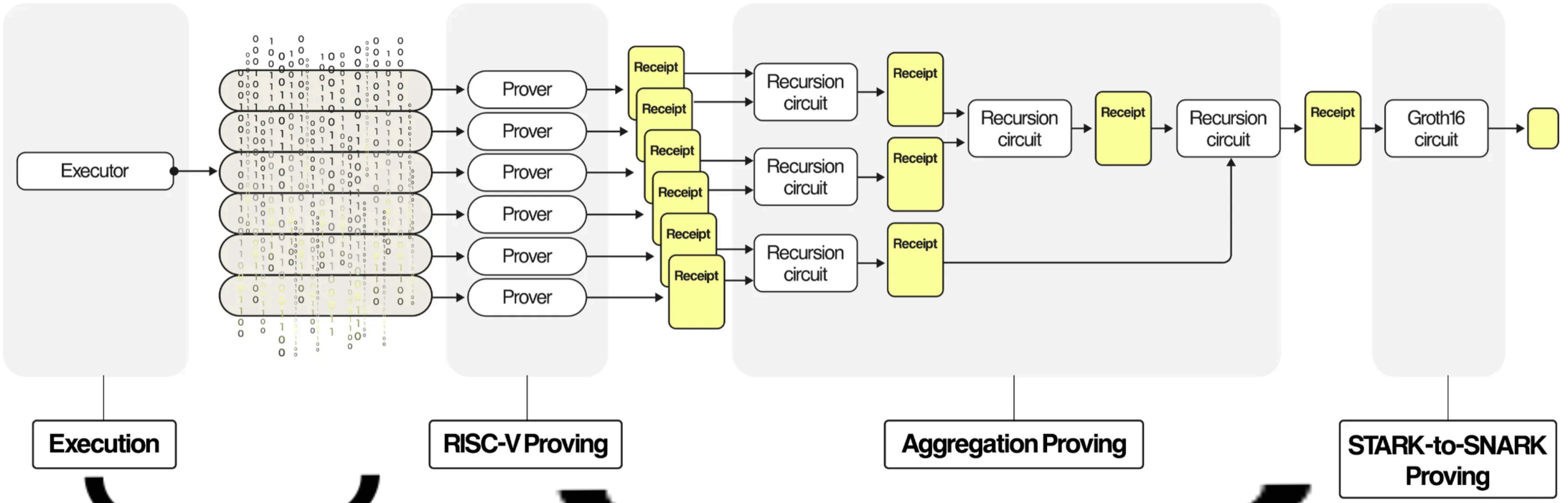
[Image credit: RiscZero]

The approach from SNARKs&Recursion



[Image credit: RiscZero]

The approach from SNARKs&Recursion



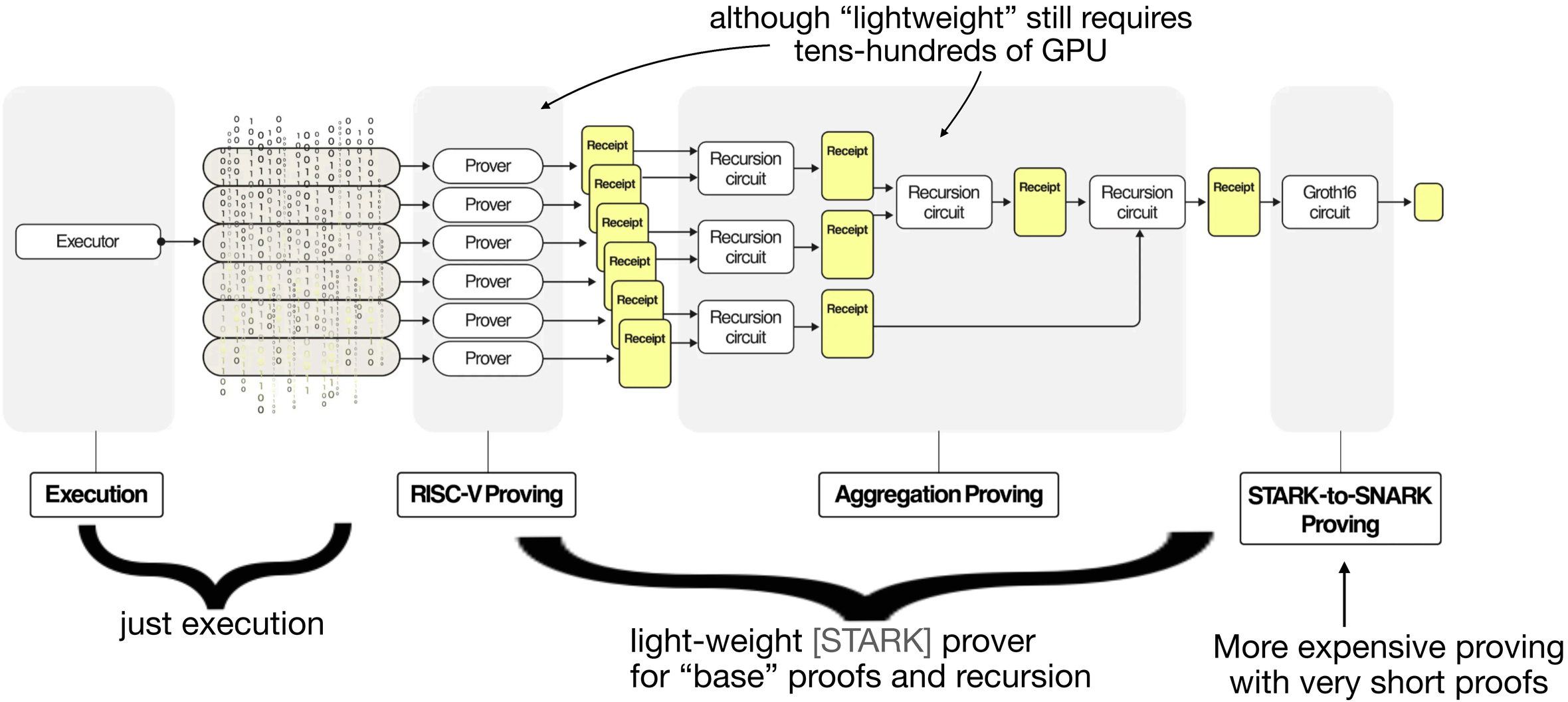
just execution

light-weight [STARK] prover
for “base” proofs and recursion

More expensive proving
with very short proofs

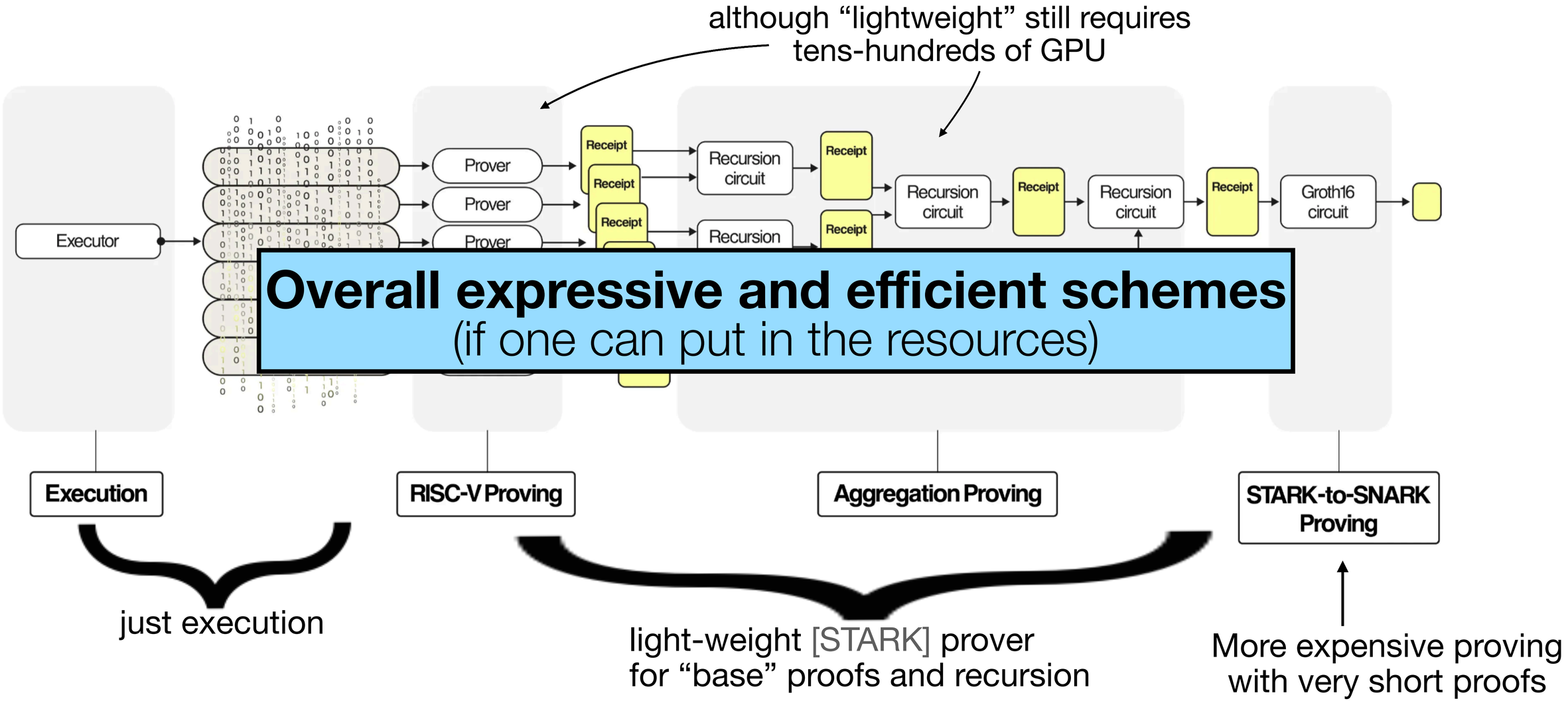
[Image credit: RiscZero]

The approach from SNARKs&Recursion



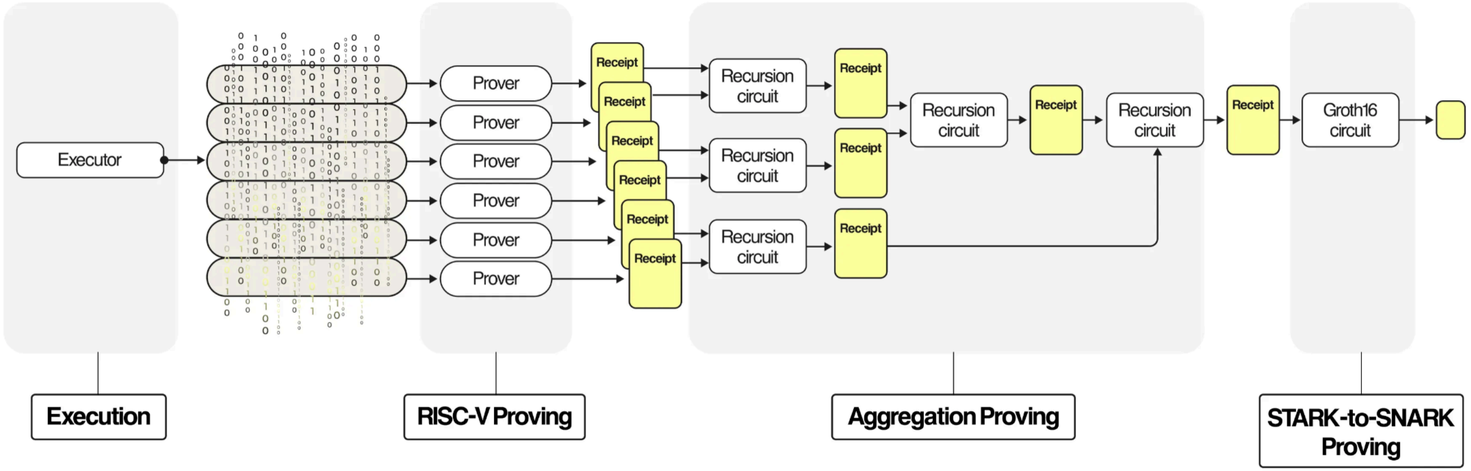
[Image credit: RiscZero]

The approach from SNARKs&Recursion

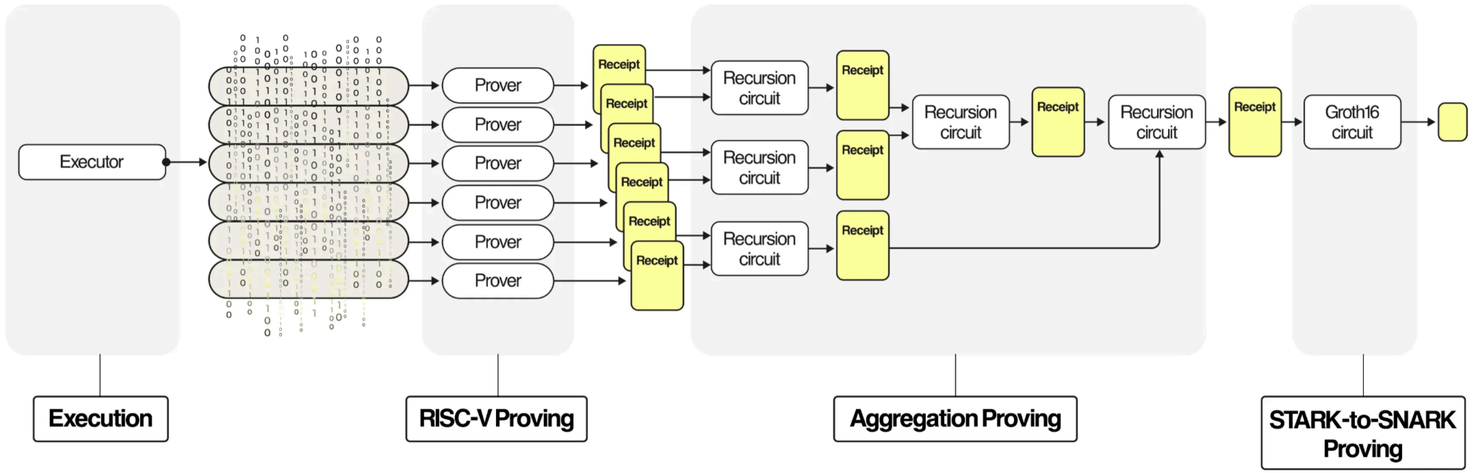


[Image credit: RiscZero]

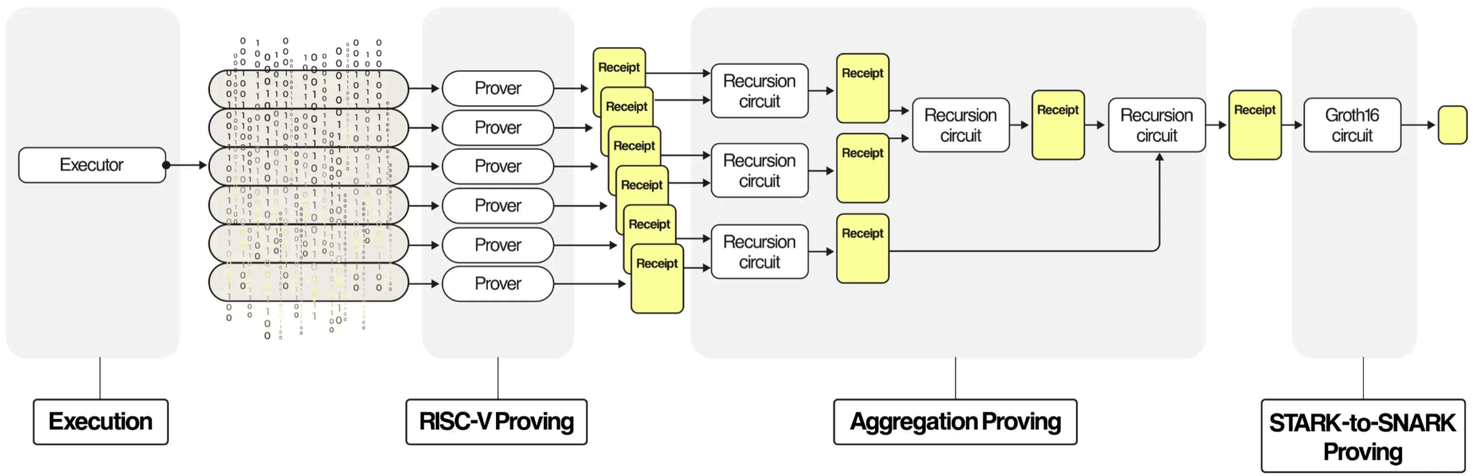
Drawback: Complexity



Drawback: Complexity



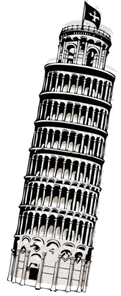
Drawback: Complexity



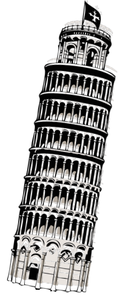
Very complex tech stack.

Hard to analyze, maintain, audit.

Drawback: Extra Security Concerns

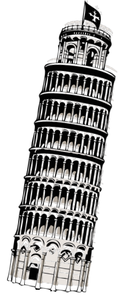


Drawback: Extra Security Concerns



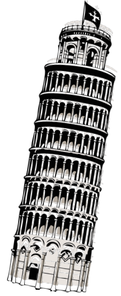
- **Security concern 1: complexity itself**

Drawback: Extra Security Concerns



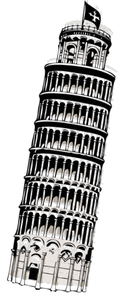
- **Security concern 1: complexity itself**
  - More building blocks and layers → More bugs that are harder to spot

Drawback: Extra Security Concerns



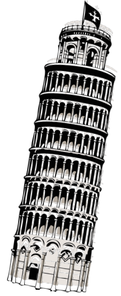
- **Security concern 1: complexity itself**
  - More building blocks and layers → More bugs that are harder to spot
- **Security concern 2: cryptographic “hygiene” and assumptions**

Drawback: Extra Security Concerns



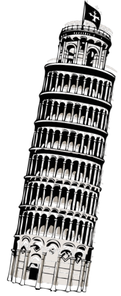
- **Security concern 1: complexity itself**
  - More building blocks and layers → More bugs that are harder to spot
- **Security concern 2: cryptographic “hygiene” and assumptions**
  - Recursion depth?

Drawback: Extra Security Concerns



- **Security concern 1: complexity itself**
  - More building blocks and layers → More bugs that are harder to spot
- **Security concern 2: cryptographic “hygiene” and assumptions**
  - Recursion depth?
  - Conjectures?

Drawback: Extra Security Concerns



- **Security concern 1: complexity itself**
  - More building blocks and layers → More bugs that are harder to spot
- **Security concern 2: cryptographic “hygiene” and assumptions**
  - Recursion depth?
  - Conjectures?
  - Random Oracle “as circuit”?

Final Drawback:

Final Drawback: 

Final Drawback: 

Is this the right way of “shaving” away the problem of Verifiable SQL?

General-purpose solutions—Summary

General-purpose solutions—Summary

- Extremely expressive ✓

General-purpose solutions—Summary

- Extremely expressive ✓
- Can be very efficient ✓

General-purpose solutions—Summary

- Extremely expressive ✓
- Can be very efficient ✓
  - fast proving time (with the right number of GPU and investment)

General-purpose solutions—Summary

- Extremely expressive ✓
- Can be very efficient ✓
  - fast proving time (with the right number of GPU and investment)
  - short proof size / small verification cost

General-purpose solutions—Summary

- Extremely expressive ✓
- Can be very efficient ✓
  - fast proving time (with the right number of GPU and investment)
  - short proof size / small verification cost

- Sledgehammer approach to verifiable SQL ❌

General-purpose solutions—Summary

- Extremely expressive ✓
- Can be very efficient ✓
  - fast proving time (with the right number of GPU and investment)
  - short proof size / small verification cost

- Sledgehammer approach to verifiable SQL ❌
- Extremely complex tech stack ❌

General-purpose solutions—Summary

- Extremely expressive ✓
- Can be very efficient ✓
  - fast proving time (with the right number of GPU and investment)
  - short proof size / small verification cost

- Sledgehammer approach to verifiable SQL ❌
- Extremely complex tech stack ❌
- Hard to analyze and audit ❌

General-purpose solutions—Summary

- Extremely expressive ✓
- Can be very efficient ✓
  - fast proving time (with the right number of GPU and investment)
  - short proof size / small verification cost

- Sledgehammer approach to verifiable SQL ❌
- Extremely complex tech stack ❌
- Hard to analyze and audit ❌
- Suboptimal developer experience (especially if requires writing circuits) ❌

General-purpose solutions—Summary

- Extremely expressive ✓
- Can be very efficient ✓
  - fast proving time (with the right number of GPU and investment)
  - short proof size / small verification cost

- Sledgehammer approach to verifiable SQL ❌
- Extremely complex tech stack ❌
- Hard to analyze and audit ❌
- Suboptimal developer experience (especially if requires writing circuits) ❌
- Additional security risks (both from complexity and cryptographic heuristics) ❌

General-purpose solutions—Summary

- Extremely expressive ✓
- Can be very efficient ✓
  - fast proving time (with the right number of GPU and investment)
  - short proof size / small verification cost

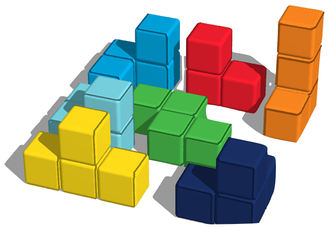
- Sledgehammer approach to verifiable SQL ❌
- Extremely complex tech stack ❌
- Hard to analyze and audit ❌
- Suboptimal developer experience (especially if requires writing circuits) ❌
- Additional security risks (both from complexity and cryptographic heuristics) ❌

My claim: we may want to explore alternative approaches for verifiable SQL.

Verifiable DBs from Authenticated Data Structures

Let’s talk about 

prior work

to then get to 

qedb (this work)

Quick Slide on Advancements in ADS

Quick Slide on Advancements in ADS

Recall:

Quick Slide on Advancements in ADS

Recall:

- **accumulators**:
  can prove $y \in S$

Quick Slide on Advancements in ADS

Recall:

- **accumulators**:
  can prove $y \in S$
  - at times, can also prove set relations $S \subseteq T, R \cup S = T, \ldots$

Quick Slide on Advancements in ADS

Recall:

- **accumulators**:
  can prove $y \in S$
  - at times, can also prove set relations $S \subseteq T, R \cup S = T, \ldots$
- **vector commitments:**
  can prove $(y_1, y_2, \ldots, y_\ell) = \left(\vec{v}[j]\right)_{j \in J}$

Quick Slide on Advancements in ADS

Recall:

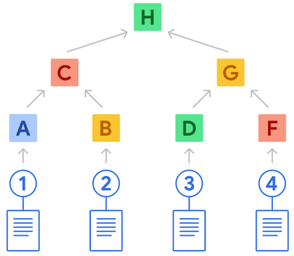
- **accumulators**:
  can prove $y \in S$
  - at times, can also prove set relations $S \subseteq T, R \cup S = T, \ldots$
- **vector commitments:**
  can prove $(y_1, y_2, \ldots, y_\ell) = \left(\vec{v}[j]\right)_{j \in J}$
- **polynomial commitments:**
  can prove $f(x) = y$

Quick Slide on Advancements in ADS

Recall:

- **accumulators**:
  can prove $y \in S$
  - at times, can also prove set relations $S \subseteq T, R \cup S = T, \ldots$
- **vector commitments:**
  can prove $(y_1, y_2, \ldots, y_\ell) = \left(\vec{v}[j]\right)_{j \in J}$
- **polynomial commitments:**
  can prove $f(x) = y$

Standard construction:
Merkle Trees (from hashing)



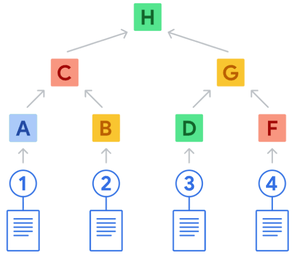
has logarithmic-sized proof

Quick Slide on Advancements in ADS

Recall:

- **accumulators**:
  can prove $y \in S$
  - at times, can also prove set relations $S \subseteq T, R \cup S = T, \ldots$
- **vector commitments:**
  can prove $(y_1, y_2, \ldots, y_\ell) = \left(\vec{v}[j]\right)_{j \in J}$
- **polynomial commitments:**
  can prove $f(x) = y$

Standard construction:
Merkle Trees (from hashing)



has logarithmic-sized proof

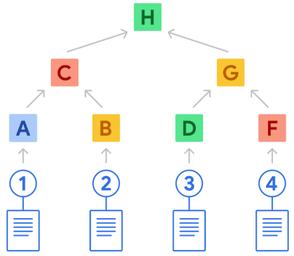
Many advancements (2010s)
from elliptic curves [pairings]

Quick Slide on Advancements in ADS

Recall:

- **accumulators**:
  can prove $y \in S$
  - at times, can also prove set relations $S \subseteq T, R \cup S = T, \ldots$
- **vector commitments:**
  can prove $(y_1, y_2, \ldots, y_\ell) = \left(\vec{v}[j]\right)_{j \in J}$
- **polynomial commitments:**
  can prove $f(x) = y$

Standard construction:
Merkle Trees (from hashing)



has logarithmic-sized proof

Many advancements (2010s)
from elliptic curves [pairings]

accumulators, vector and polynomial commitments with O(1) sized proofs.

Verifiable DBs from ADS (state of the art)

IntegriDB (CCS 2015)

Verifiable DBs from ADS (state of the art)

IntegriDB (CCS 2015)

Improves on the state of the art on ADS-based verifiable DBs:

- combines simple hash-based auth. interval trees with modern accumulators (from elliptic curves)

Verifiable DBs from ADS (state of the art)

IntegriDB (CCS 2015)

Improves on the state of the art on ADS-based verifiable DBs:

- combines simple hash-based auth. interval trees with modern accumulators (from elliptic curves)

	Join	Multidim range	Functions	Nested queries	Update
Tree-based [YPPK09]	✘	✘	✘	✘	✓
Signature-based [PZM09]	✘	✘	✘	✘	✘
Multi-range [PPT14]	✘	✓	✘	✘	✘
IntegriDB	✓	✓	✓	✓	✓

Table by Yupeng Zhang (from IntegriDB presentation @ ACM CCS 2015).

Verifiable DBs from ADS (state of the art)

IntegriDB (CCS 2015)

Improves on the state of the art on ADS-based verifiable DBs:

- combines simple hash-based auth. interval trees with modern accumulators (from elliptic curves)

	Join	Multidim range	Functions
Tree-based [YPPK09]	✘	✘	✘
Signature-based [PZM09]	✘	✘	✘
Multi-range [PPT14]	✘	✓	✘
IntegriDB	✓	✓	✓

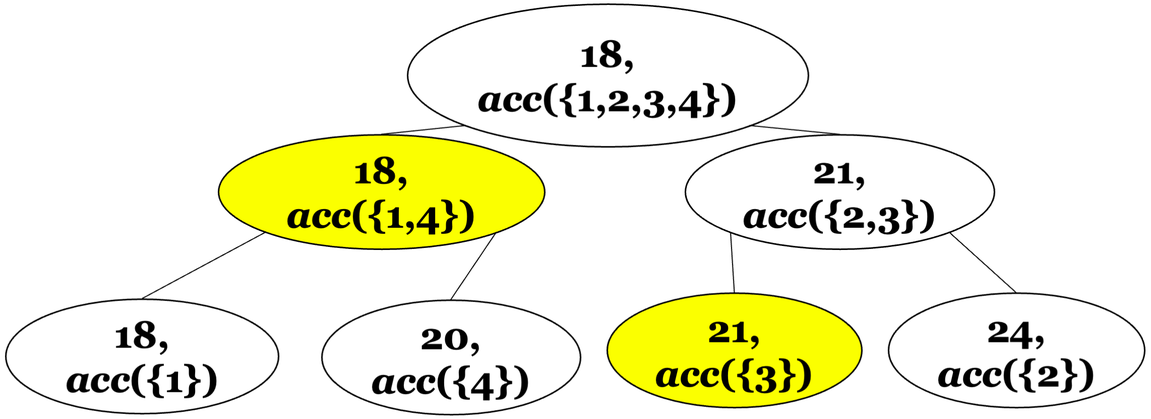
```
1. SELECT SUM(l_extendedprice* (1 - l_discount))
2. AS revenue
3. FROM lineitem, part
4. WHERE
5. (    p_partkey = l_partkey
6.    AND p_brand = 'Brand#41'
7.    AND p_container IN ('SM CASE', 'SM BOX', 'SM
          PACK', 'SM PKG')
8.    AND l_quantity >= 7 AND l_quantity <= 7 + 10
9.    AND p_size BETWEEN 1 AND 5
10.   AND l_shipmode IN ('AIR', 'AIR REG')
11.   AND l_shipinstruct = 'DELIVER IN PERSON' )
12. OR
13. (   p_partkey = l_partkey
14.   AND p_brand = 'Brand#14'
15.   AND p_container IN ('MED BAG', 'MED BOX',
          'MED PKG', 'MED PACK')
16.   AND l_quantity >= 14 AND l_quantity <= 14 + 10
17.   AND p_size BETWEEN 1 AND 10
18.   AND l_shipmode IN ('AIR', 'AIR REG')
19.   AND l_shipinstruct = 'DELIVER IN PERSON' )
20. OR
21. (   p_partkey = l_partkey
22.   AND p_brand = 'Brand#23'
23.   AND p_container IN ('LG CASE', 'LG BOX', 'LG
          PACK', 'LG PKG')
24.   AND l_quantity >= 25 AND l_quantity <= 25 + 10
25.   AND p_size BETWEEN 1 AND 15
26.   AND l_shipmode IN ('AIR', 'AIR REG')
27.   AND l_shipinstruct = 'DELIVER IN PERSON' );
```

Figure 6: Query #19 of the TPC-H benchmark.

Table by Yupeng Zhang (from IntegriDB presentation @ ACM CCS 2015).

Verifiable DBs from ADS (state of the art)

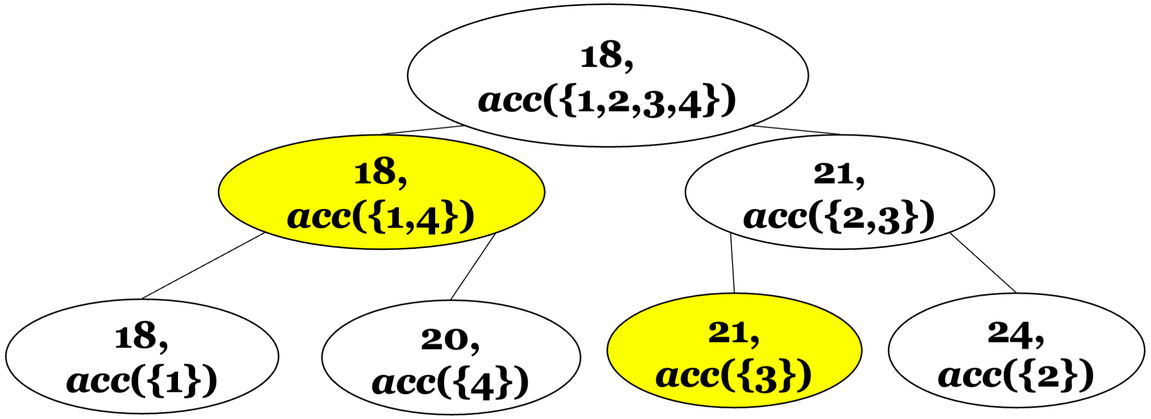
IntegriDB (CCS 2015)



Verifiable DBs from ADS (state of the art)

IntegriDB (CCS 2015)

Techniques in a nutshell:

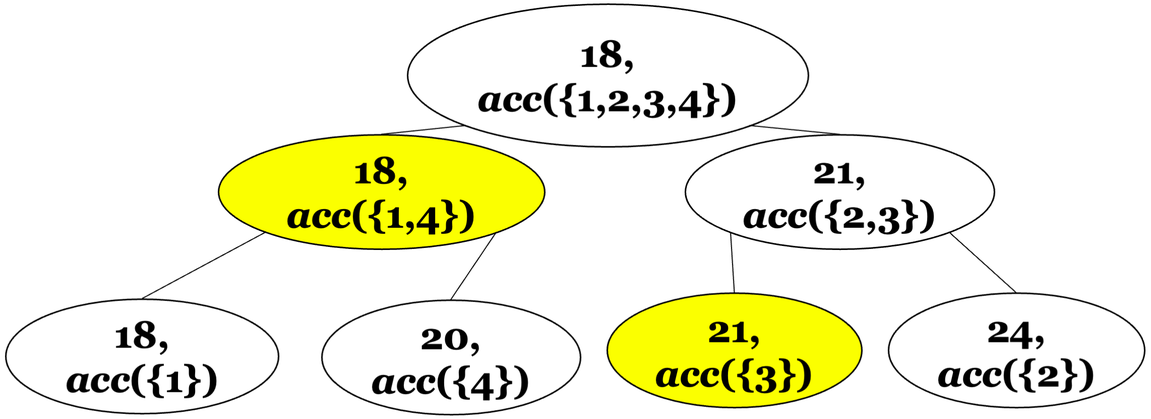


Verifiable DBs from ADS (state of the art)

IntegriDB (CCS 2015)

Techniques in a nutshell:

- “collapses” sets of rows with specific intervals properties (via accumulators)
- combines accumulators and authenticated interval trees
- proves OR/AND via set relation proofs



Pain Points of the State of the Art

(pain points of IntegriDB)

Pain Points of the State of the Art

(pain points of IntegriDB)

Proof Size is large

Pain Points of the State of the Art

(pain points of IntegriDB)

Proof Size is large

Can easily get to hundreds of KB.
Grows with DB size.

Pain Points of the State of the Art

(pain points of IntegriDB)

Proof Size **is large**

Can easily get to hundreds of KB.
Grows with DB size.

High computational requirements
(proving and preprocessing)

Pain Points of the State of the Art

(pain points of IntegriDB)

Proof Size **is large**

Can easily get to hundreds of KB.
Grows with DB size.

Very memory intensive

High computational requirements
(proving and preprocessing)

Pain Points of the State of the Art

(pain points of IntegriDB)

Proof Size **is large**

Can easily get to hundreds of KB.
Grows with DB size.

Very memory intensive

High computational requirements
(proving and preprocessing)

Their techniques inherently entail a **quadratic** overhead in order to support JOINs

Pain Points of the State of the Art

(pain points of IntegriDB)

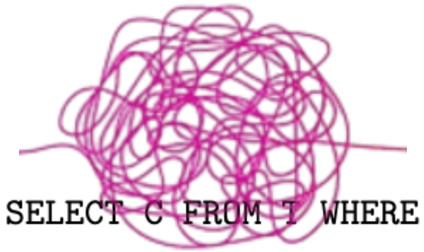
Proof Size **is large**

Can easily get to hundreds of KB.
Grows with DB size.

Very memory intensive

High computational requirements
(proving and preprocessing)

Their techniques inherently entail a **quadratic** overhead in order to support JOINs



Constrained expressivity and succinctness

Pain Points of the State of the Art

(pain points of IntegriDB)

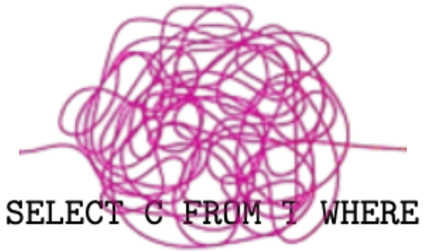
Proof Size is large

Can easily get to hundreds of KB.
Grows with DB size.

High computational requirements
(proving and preprocessing)

Very memory intensive

Their techniques inherently entail a **quadratic** overhead in order to support JOINs



Constrained expressivity and succinctness

E.g., doesn't support comparison among columns

Pain Points of the State of the Art

(pain points of IntegriDB)

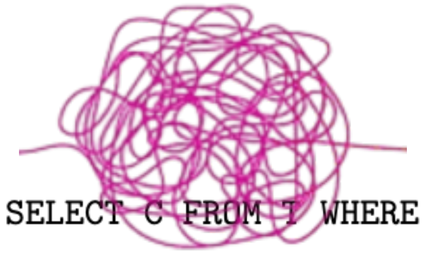
Proof Size is large

Can easily get to hundreds of KB.
Grows with DB size.

High computational requirements
(proving and preprocessing)

Very memory intensive

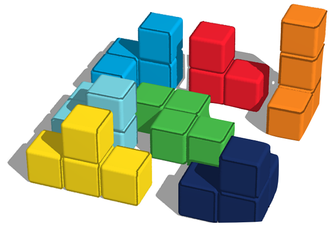
Their techniques inherently entail a **quadratic** overhead in order to support JOINs



Constrained expressivity and succinctness

E.g., doesn't support comparison among columns

Proof size/verification grows with # of duplicated elements in response



This work (qedb) addresses many of these limitations

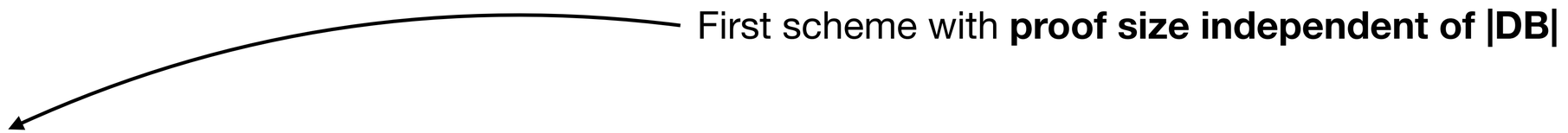
Core Thesis of qedb

It is possible to design Verifiable DBs that are:

- highly **efficient**
- highly **expressive**
- from **simple** building blocks

Core Thesis of qedb

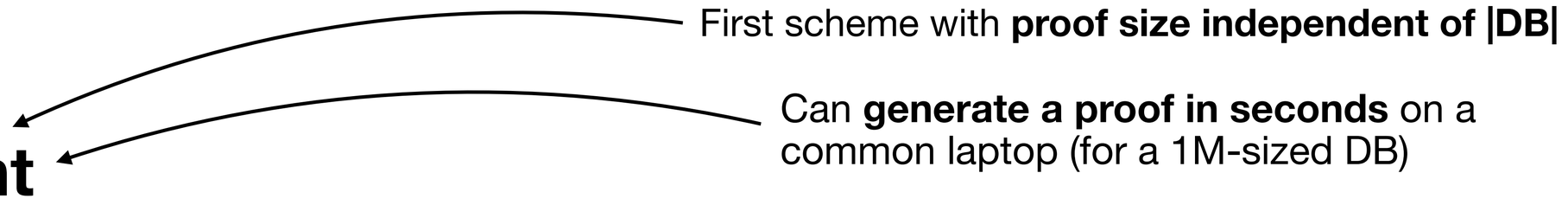
It is possible to design Verifiable DBs that are:



- highly **efficient**
- highly **expressive**
- from **simple** building blocks

Core Thesis of qedb

It is possible to design Verifiable DBs that are:

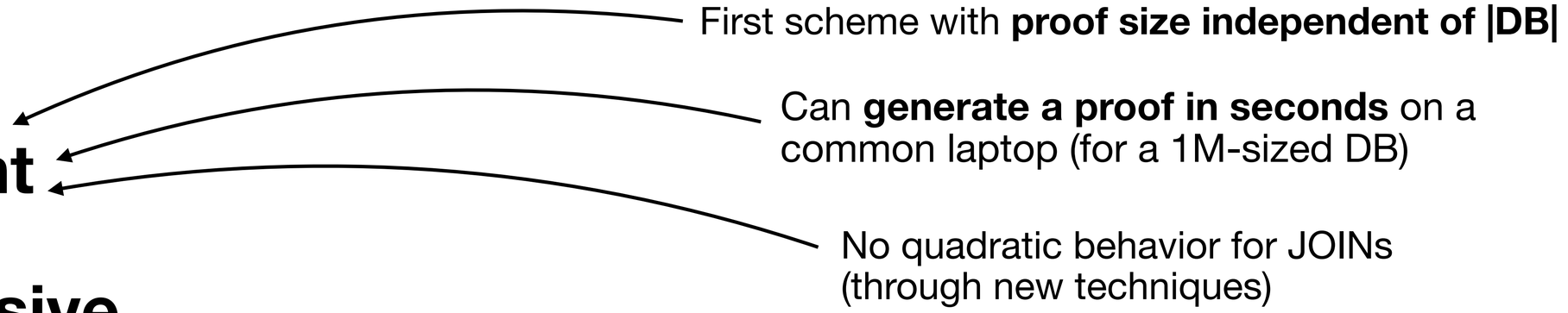


- highly **efficient**
- highly **expressive**
- from **simple** building blocks

Core Thesis of qedb

It is possible to design Verifiable DBs that are:

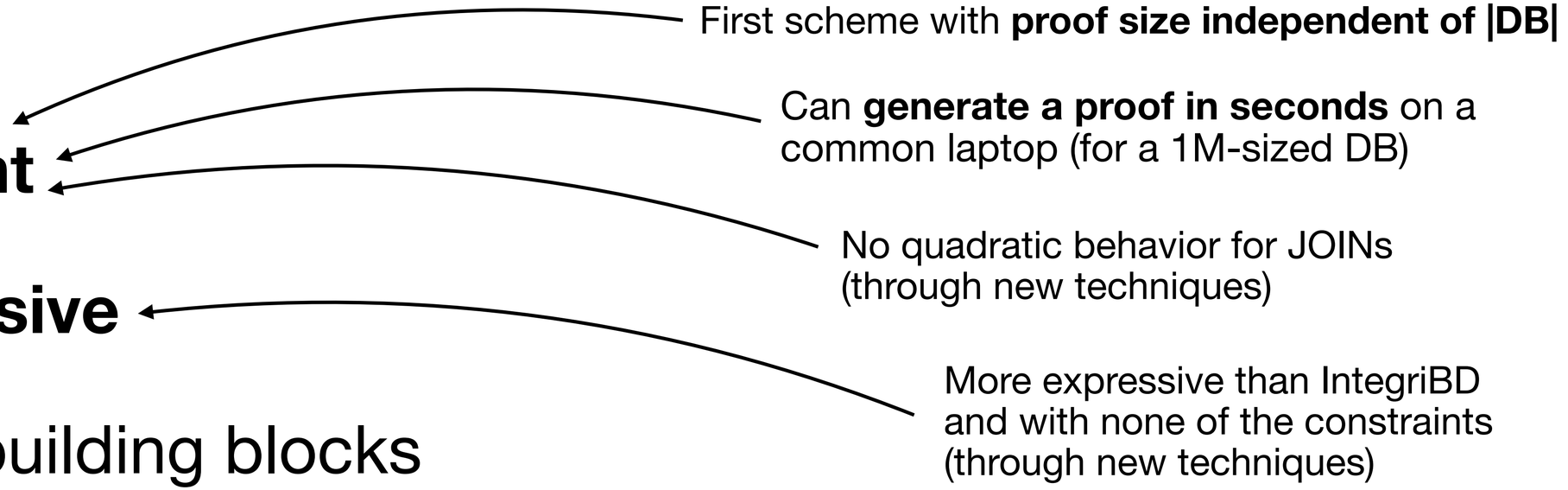
- highly **efficient**
- highly **expressive**
- from **simple** building blocks



Core Thesis of qedb

It is possible to design Verifiable DBs that are:

- highly **efficient**
- highly **expressive**
- from **simple** building blocks



Core Thesis of qedb

It is possible to design Verifiable DBs that are:

- highly **efficient**
- highly **expressive**
- from **simple** building blocks

First scheme with **proof size independent of |DB|**

Can **generate a proof in seconds** on a common laptop (for a 1M-sized DB)

No quadratic behavior for JOINs (through new techniques)

More expressive than IntegriBD and with none of the constraints (through new techniques)

No general-purpose SNARKs. Instead: specialized vector commitments and accumulators

Zooming in on Efficiency

Asymptotics

Scheme
IntegriDB [85]
vSQL [84]
This work

(NB: commonly $|\mathsf{resp}| \ll |\mathsf{column}| \ll |\mathsf{db}|$; for aggregate queries, $|\mathsf{qry}| \approx |\mathsf{resp}|$, else $|\mathsf{qry}| \ll |\mathsf{resp}|$).

Zooming in on Efficiency

Asymptotics

Scheme	Overhead in $\lvert\pi\rvert$, $V_{\mathbf{time}}$ (queries w/o JOINs)
IntegriDB [85]	log(|column|)
vSQL [84]	polylog|db|
This work	|qry|

(NB: commonly $\lvert\mathsf{resp}\rvert \ll \lvert\mathsf{column}\rvert \ll \lvert\mathsf{db}\rvert$; for aggregate queries, $\lvert\mathsf{qry}\rvert \approx \lvert\mathsf{resp}\rvert$, else $\lvert\mathsf{qry}\rvert \ll \lvert\mathsf{resp}\rvert$).

Zooming in on Efficiency

Asymptotics

Scheme	Overhead in $\|\pi\|$, V_{time} (queries w/o JOINs)	Overhead in $\|\pi\|$, V_{time} (JOINs)
IntegriDB [85]	$\log(\|\mathsf{column}\|)$	$\|\mathsf{resp}\| \cdot \log \|\mathsf{column}\|$
vSQL [84]	$\text{polylog}\|\mathsf{db}\|$	$\text{polylog}\|\mathsf{db}\|$
This work	$\|\mathsf{qry}\|$	$\|\mathsf{resp}\|$

(NB: commonly $|\mathsf{resp}| \ll |\mathsf{column}| \ll |\mathsf{db}|$; for aggregate queries, $|\mathsf{qry}| \approx |\mathsf{resp}|$, else $|\mathsf{qry}| \ll |\mathsf{resp}|$).

Zooming in on Efficiency

Asymptotics

Scheme	Overhead in $\|\pi\|$, V_{time} (queries w/o JOINs)	Overhead in $\|\pi\|$, V_{time} (JOINs)	Preprocessing & server storage
IntegriDB [85]	$\log(\|\mathsf{column}\|)$	$\|\mathsf{resp}\| \cdot \log \|\mathsf{column}\|$	$\|\mathsf{db}\| + n_{\text{cols}}^2$
vSQL [84]	$\text{polylog}\|\mathsf{db}\|$	$\text{polylog}\|\mathsf{db}\|$	$\|\mathsf{db}\|$
This work	$\|\mathsf{qry}\|$	$\|\mathsf{resp}\|$	$\|\mathsf{db}\|$

(NB: commonly $|\mathsf{resp}| \ll |\mathsf{column}| \ll |\mathsf{db}|$; for aggregate queries, $|\mathsf{qry}| \approx |\mathsf{resp}|$, else $|\mathsf{qry}| \ll |\mathsf{resp}|$).

Zooming in on Efficiency

Preliminary Experimental Evaluation (DB with 100K rows)

Q_{Tot} ●

```
SELECT SUM(price) FROM Transaction
WHERE account_id = '5938' AND trade_date = '2025-01-01'
```

↳ Computes total price of transactions executed by an account on a given date

Q_{CntTx} ▲

```
SELECT COUNT(*) FROM Transaction
WHERE trade_date BETWEEN '2025-01-01' AND '2025-03-31'
```

↳ Computes the number of transactions executed within the first quarter

Q_{MatchExp} ■

```
SELECT tx_id, price, expected_price, price = expected_price
FROM Transaction WHERE trade_date = '2025-04-05'
```

↳ Retrieves the transactions whose executed price equals their expected price

Zooming in on Efficiency

Preliminary Experimental Evaluation (DB with 100K rows)

Query	Prover Time	Verifier Time	Proof Size
Q_{Tot} ●	1.21 s	13.00 ms	0.66 KB
Q_{CntTx} ▲	15.59 s	21.81 ms	5.13 KB
Q_{MatchExp} ■	6.15 s	25.17 ms	0.98 KB

Q_{Tot} ●

```
SELECT SUM(price) FROM Transaction
WHERE account_id = '5938' AND trade_date = '2025-01-01'
```

↳ Computes total price of transactions executed by an account on a given date

Q_{CntTx} ▲

```
SELECT COUNT(*) FROM Transaction
WHERE trade_date BETWEEN '2025-01-01' AND '2025-03-31'
```

↳ Computes the number of transactions executed within the first quarter

Q_{MatchExp} ■

```
SELECT tx_id, price, expected_price, price = expected_price
FROM Transaction WHERE trade_date = '2025-04-05'
```

↳ Retrieves the transactions whose executed price equals their expected price

Zooming in on Efficiency

Preliminary Experimental Evaluation (DB with 100K rows)

Query	Prover Time	Verifier Time	Proof Size
Q_{Tot} ●	1.21 s	13.00 ms	0.66 KB
Q_{CntTx} ▲	15.59 s	21.81 ms	5.13 KB
Q_{MatchExp} ■	6.15 s	25.17 ms	0.98 KB

Q_{Tot} ●

```
SELECT SUM(price) FROM Transaction
WHERE account_id = '5938' AND trade_date = '2025-01-01'
```

↳ Computes total price of transactions executed by an account on a given date

Q_{CntTx} ▲

```
SELECT COUNT(*) FROM Transaction
WHERE trade_date BETWEEN '2025-01-01' AND '2025-03-31'
```

↳ Computes the number of transactions executed within the first quarter

Q_{MatchExp} ■

```
SELECT tx_id, price, expected_price, price = expected_price
FROM Transaction WHERE trade_date = '2025-04-05'
```

↳ Retrieves the transactions whose executed price equals their expected price

5x smaller than IntegriDB's

Zooming in on Efficiency

Preliminary Experimental Evaluation (DB with 100K rows)

One order of magnitude smaller than IntegriDB's

Query	Prover Time	Verifier Time	Proof Size
Q_{Tot} ●	1.21 s	13.00 ms	0.66 KB
Q_{CntTx} ▲	15.59 s	21.81 ms	5.13 KB
Q_{MatchExp} ■	6.15 s	25.17 ms	0.98 KB

5x smaller than IntegriDB's

Q_{Tot} ●

```
SELECT SUM(price) FROM Transaction
WHERE account_id = '5938' AND trade_date = '2025-01-01'
```

↳ Computes total price of transactions executed by an account on a given date

Q_{CntTx} ▲

```
SELECT COUNT(*) FROM Transaction
WHERE trade_date BETWEEN '2025-01-01' AND '2025-03-31'
```

↳ Computes the number of transactions executed within the first quarter

Q_{MatchExp} ■

```
SELECT tx_id, price, expected_price, price = expected_price
FROM Transaction WHERE trade_date = '2025-04-05'
```

↳ Retrieves the transactions whose executed price equals their expected price

Zooming in on Efficiency

Preliminary Experimental Evaluation (DB with 100K rows)

One order of magnitude smaller than IntegriDB's

Query	Prover Time	Verifier Time	Proof Size
Q_{Tot} ●	1.21 s	13.00 ms	0.66 KB
Q_{CntTx} ▲	15.59 s	21.81 ms	5.13 KB
Q_{MatchExp} ■	6.15 s	25.17 ms	0.98 KB

Large time / proof size due to naive implementation of a range proof subprotocol

5x smaller than IntegriDB's

Q_{Tot} ●

```
SELECT SUM(price) FROM Transaction
WHERE account_id = '5938' AND trade_date = '2025-01-01'
```

↳ Computes total price of transactions executed by an account on a given date

Q_{CntTx} ▲

```
SELECT COUNT(*) FROM Transaction
WHERE trade_date BETWEEN '2025-01-01' AND '2025-03-31'
```

↳ Computes the number of transactions executed within the first quarter

Q_{MatchExp} ■

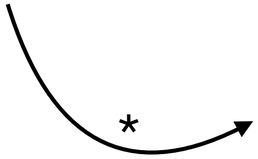
```
SELECT tx_id, price, expected_price, price = expected_price
FROM Transaction WHERE trade_date = '2025-04-05'
```

↳ Retrieves the transactions whose executed price equals their expected price

Zooming in on Expressivity

Zooming in on Expressivity

IntegriDB supports:



Predicate types: Multi-dim. range queries, list membership, any AND/OR.
Aggregate queries: MAX, MIN, COUNT, SUM, AVG.
JOINs: Equality-based joins over columns, possibly with duplicates.

* Not always supported succinctly: loses succinct proof in case of duplicate elements.

Zooming in on Expressivity

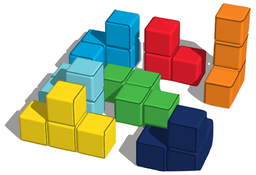
IntegriDB supports:

*

Predicate types: Multi-dim. range queries, list membership, any AND/OR.

Aggregate queries: MAX, MIN, COUNT, SUM, AVG.

JOINs: Equality-based joins over columns, possibly with duplicates.



qedb supports:

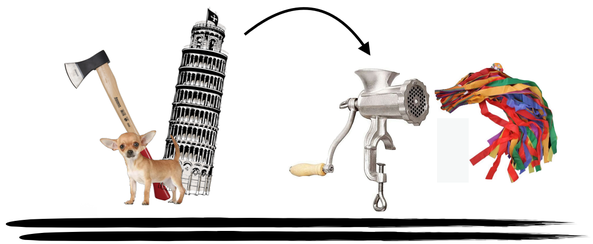
Comparison between columns: Predicates involving more than one column (e.g. "[...] WHERE $c_1 \geq 2c_2 + c_3$").

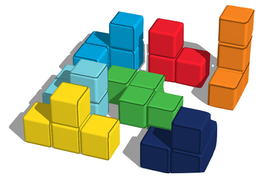
Aggregation among columns: Expressions involving more than one column in the SELECT clause (e.g. "SELECT $c_1 + 2c_2$ FROM [...]").

* Not always supported succinctly: loses succinct proof in case of duplicate elements.

Zooming in on Simplicity

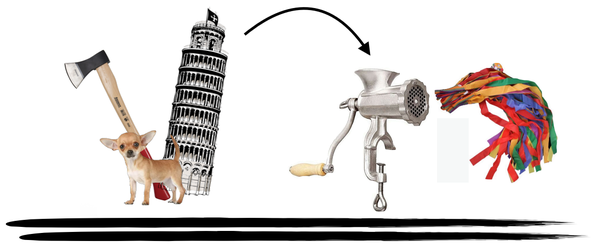
A First Lens—Tech Stacks

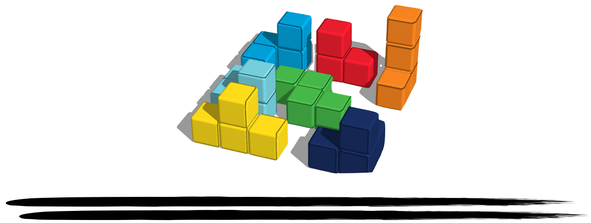




Zooming in on Simplicity

A First Lens—Tech Stacks

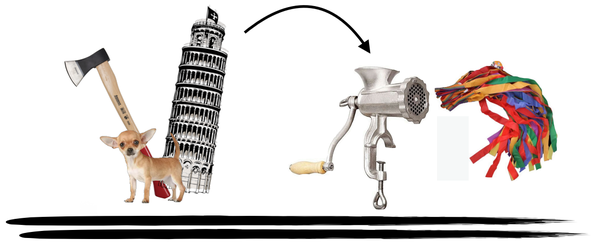


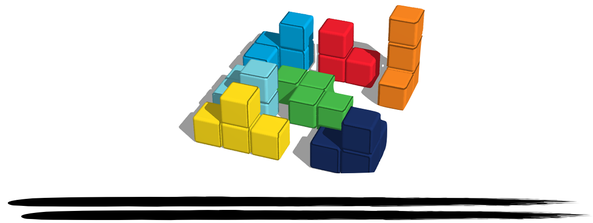


“If you are stuck on a desert island and all you have is a KZG setup, then you should still be able to deploy a Verifiable DB in a few hours”

Zooming in on Simplicity

A First Lens—Tech Stacks



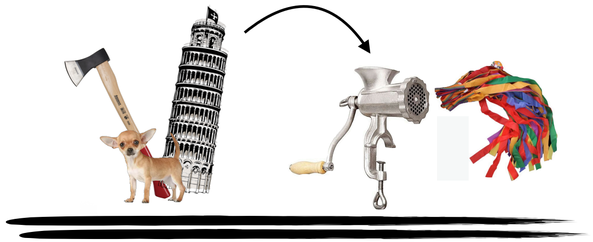


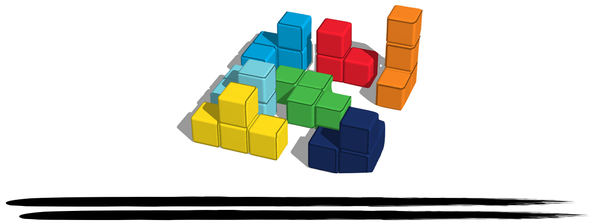
“If you are stuck on a desert island and all you have is a KZG setup, then you should still be able to deploy a Verifiable DB in a few hours”

[For non cryptographers, KZG is arguably the “most popular” polynomial commitment, common in deployed solutions also]

Zooming in on Simplicity

A First Lens—Tech Stacks





A good B.Sc. student can code this easily

Protocol “Atoms” and Their Composition

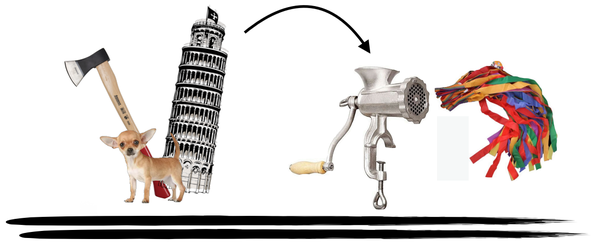
KZG

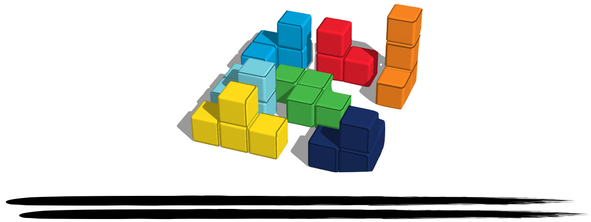
“If you are stuck on a desert island and all you have is a KZG setup, then you should still be able to deploy a Verifiable DB in a few hours”

[For non cryptographers, KZG is arguably the “most popular” polynomial commitment, common in deployed solutions also]

Zooming in on Simplicity

A First Lens—Tech Stacks





A good B.Sc. student can code this easily

Protocol “Atoms” and Their Composition

KZG

Groth16

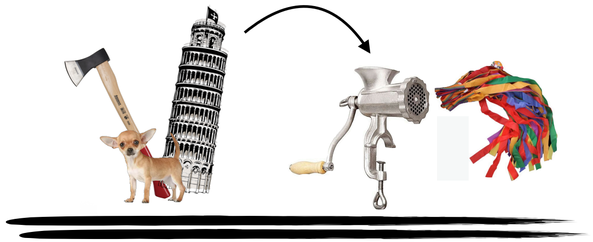
FRI & STARKs

“If you are stuck on a desert island and all you have is a KZG setup, then you should still be able to deploy a Verifiable DB in a few hours”

[For non cryptographers, KZG is arguably the “most popular” polynomial commitment, common in deployed solutions also]

Zooming in on Simplicity

A First Lens—Tech Stacks

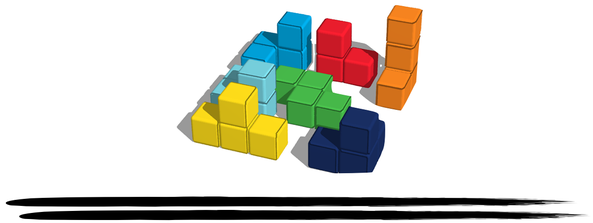


Circuits* for SQL

Circuits* for STARK recursion

Groth16

FRI & STARKs



A good B.Sc. student can code this easily

Protocol “Atoms” and Their Composition

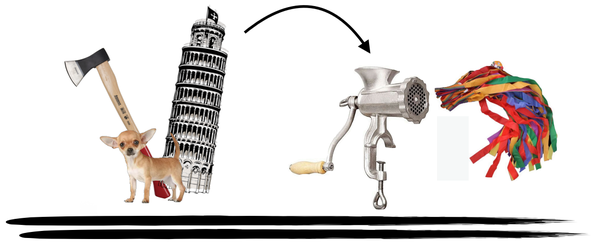
KZG

“If you are stuck on a desert island and all you have is a KZG setup, then you should still be able to deploy a Verifiable DB in a few hours”

[For non cryptographers, KZG is arguably the “most popular” polynomial commitment, common in deployed solutions also]

Zooming in on Simplicity

A First Lens—Tech Stacks

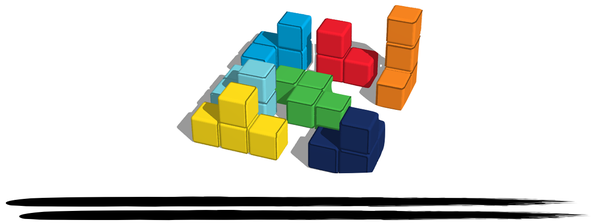


Circuits* for SQL

Circuits* for STARK recursion

Groth16

FRI & STARKs



A good B.Sc. student can code this easily

Protocol “Atoms” and Their Composition

KZG

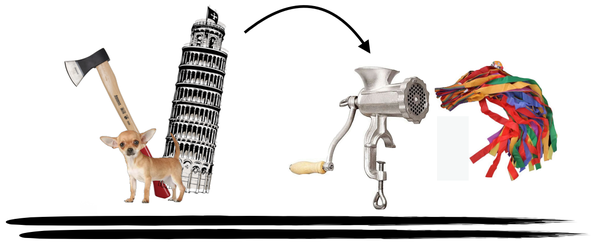
“If you are stuck on a desert island and all you have is a KZG setup, then you should still be able to deploy a Verifiable DB in a few hours”

* or other representation (constraints or “ZK”-VM port)

[For non cryptographers, KZG is arguably the “most popular” polynomial commitment, common in deployed solutions also]

Zooming in on Simplicity

A First Lens—Tech Stacks

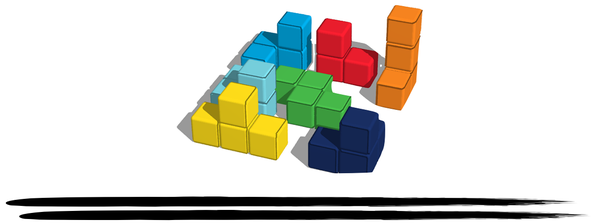


Circuits* for SQL

Circuits* for STARK recursion

Groth16

FRI & STARKs



A good B.Sc. student can code this easily

Protocol “Atoms” and Their Composition

KZG

“If you are stuck on a desert island and all you have is a KZG setup, then you should still be able to deploy a Verifiable DB in a few hours”

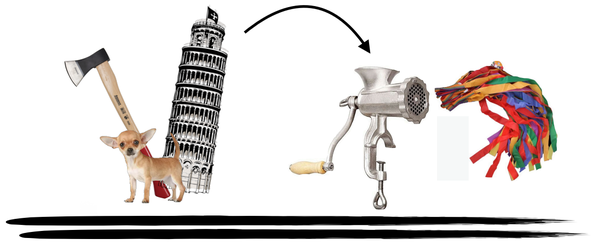
* or other representation (constraints or “ZK”-VM port)

** if applicable

[For non cryptographers, KZG is arguably the “most popular” polynomial commitment, common in deployed solutions also]

Zooming in on Simplicity

A First Lens—Tech Stacks



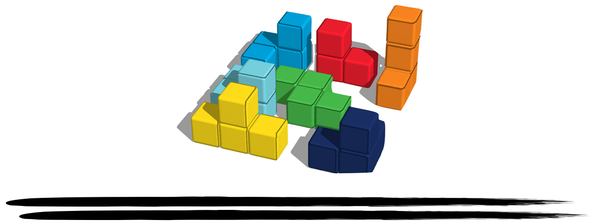
Circuits* for SQL

Circuits* for STARK recursion

Groth16

FRI & STARKs

Segmenting Computations & Recursion Tree Logic



A good B.Sc. student can code this easily

Protocol “Atoms” and Their Composition

KZG

“If you are stuck on a desert island and all you have is a KZG setup, then you should still be able to deploy a Verifiable DB in a few hours”

* or other representation (constraints or “ZK”-VM port)

** if applicable

[For non cryptographers, KZG is arguably the “most popular” polynomial commitment, common in deployed solutions also]

Zooming in on Simplicity

A Second Lens—Modularity

Zooming in on Simplicity

A Second Lens—Modularity

Designing powerful protocols is good.

Zooming in on Simplicity

A Second Lens—Modularity

Designing powerful protocols is good. Doing that by glueing simple protocols is best.

Zooming in on Simplicity

A Second Lens—Modularity

Designing powerful protocols is good. Doing that by glueing simple protocols is best.

Thompson and Ritchie, creators of UNIX

An apocryphal quote from the holy scriptures (i.e., left as a comment just before a macro definitions in one of the early UNIX implementations)

Zooming in on Simplicity

A Second Lens—Modularity

Designing powerful protocols is good. Doing that by glueing simple protocols is best.

An apocryphal quote from the holy scriptures (i.e., left as a comment just before a macro definitions in one of the early UNIX implementations)

Thompson and Ritchie, creators of UNIX

The qedb vision for Verifiable DBs (VDB) design:

```
echo “SELECT * FROM T WHERE block_number = 0x123” | vdb_prover_no_crypto | vdb_compile_to_crypto
```

Zooming in on Simplicity

A Second Lens—Modularity

Designing powerful protocols is good. Doing that by glueing simple protocols is best.

An apocryphal quote from the holy scriptures (i.e., left as a comment just before a macro definitions in one of the early UNIX implementations)

Thompson and Ritchie, creators of UNIX

The qedb vision for Verifiable DBs (VDB) design:

```
echo “SELECT * FROM T WHERE block_number = 0x123” | vdb_prover_no_crypto | vdb_compile_to_crypto
```

Separate **algorithmic** (or information-theoretic) concerns from the **cryptographic** ones

Zooming in on Simplicity

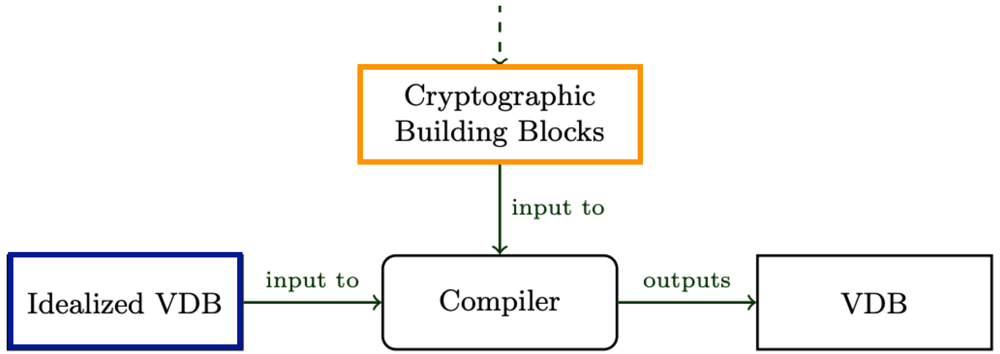
A Second Lens — Modularity (continued)

```
echo “SELECT * FROM T WHERE block_number = 0x123” | vdb_prover_no_crypto | vdb_compile_to_crypto
```

Zooming in on Simplicity

A Second Lens—Modularity (continued)

```
echo “SELECT * FROM T WHERE block_number = 0x123” | vdb_prover_no_crypto | vdb_compile_to_crypto
```

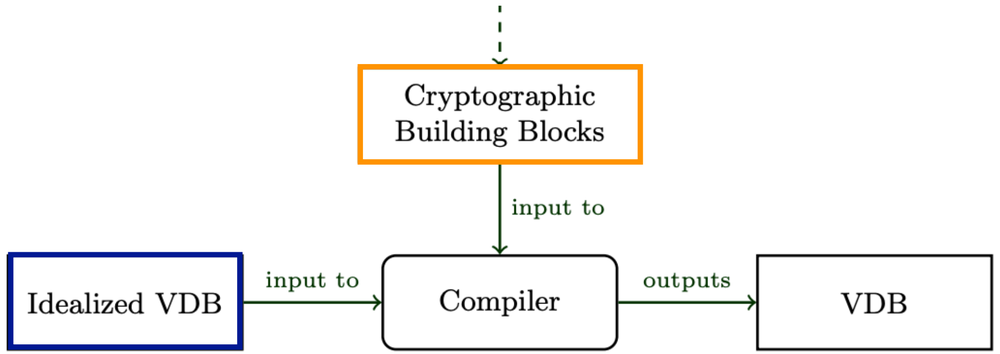


Zooming in on Simplicity

A Second Lens—Modularity (continued)

```
echo “SELECT * FROM T WHERE block_number = 0x123” | vdb_prover_no_crypto | vdb_compile_to_crypto
```

The resulting design flow:



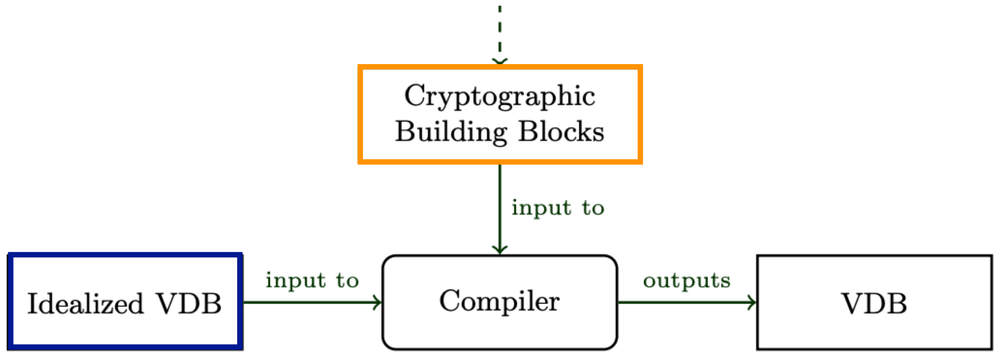
Zooming in on Simplicity

A Second Lens—Modularity (continued)

```
echo “SELECT * FROM T WHERE block_number = 0x123” | vdb_prover_no_crypto | vdb_compile_to_crypto
```

The resulting design flow:

1. Design an “idealized” VDB
   (don’t think about cryptography)



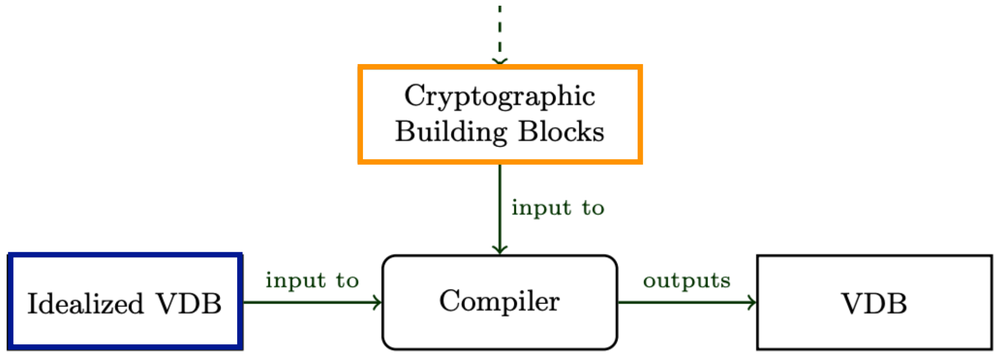
Zooming in on Simplicity

A Second Lens—Modularity (continued)

```
echo "SELECT * FROM T WHERE block_number = 0x123" | vdb_prover_no_crypto | vdb_compile_to_crypto
```

The resulting design flow:

1. Design an "idealized" VDB
   (don't think about cryptography)
2. Prove its security in its own idealized model
   (this is often *very* easy)



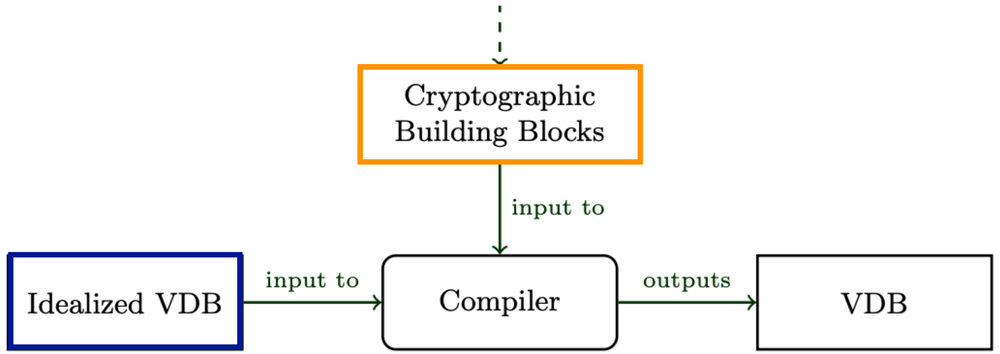
Zooming in on Simplicity

A Second Lens—Modularity (continued)

```
echo "SELECT * FROM T WHERE block_number = 0x123" | vdb_prover_no_crypto | vdb_compile_to_crypto
```

The resulting design flow:

1. Design an "idealized" VDB
   (don't think about cryptography)
2. Prove its security in its own idealized model
   (this is often *very* easy)
3. Use the cryptographic compiler
   → get security of final VDB for free



Zooming in on Simplicity

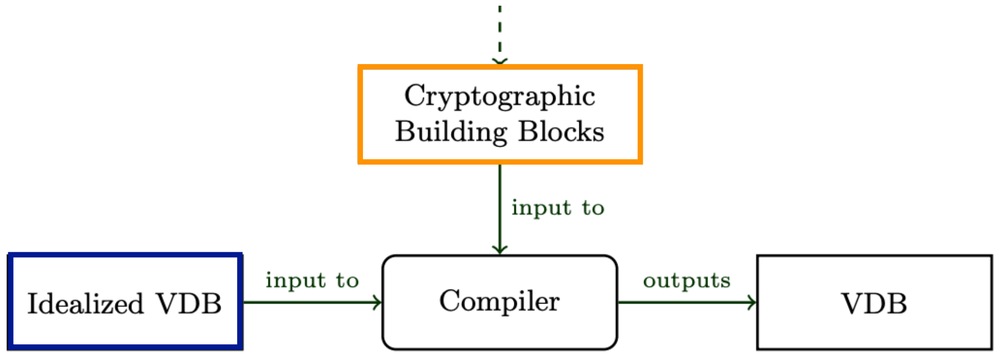
A Second Lens—Modularity (continued)

```
echo "SELECT * FROM T WHERE block_number = 0x123" | vdb_prover_no_crypto | vdb_compile_to_crypto
```

The resulting design flow:

1. Design an "idealized" VDB
   (don't think about cryptography)
2. Prove its security in its own idealized model
   (this is often *very* easy)
3. Use the cryptographic compiler
   → get security of final VDB for free

Implication 1: Want to improve the design of a VDB? Improve its building blocks OR the idealized blueprint. Afterwards, *no need to reprove security from scratch*.



Zooming in on Simplicity

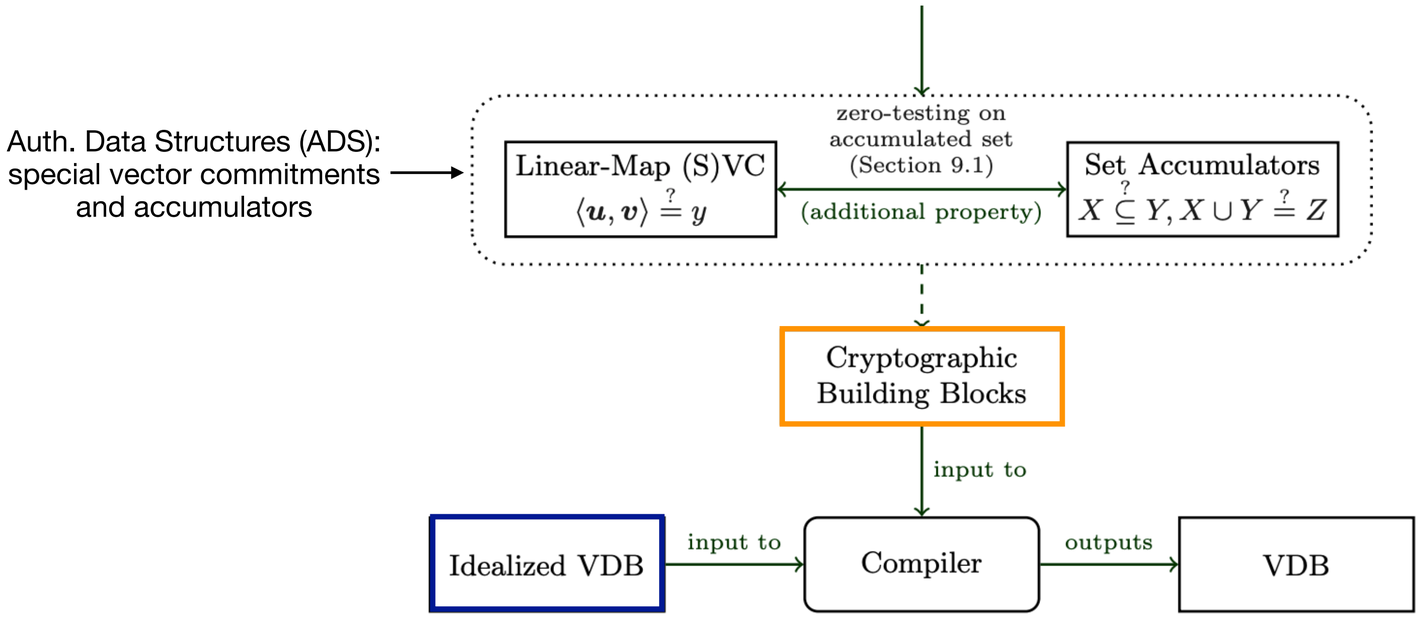
A Second Lens—Modularity (continued)

`echo "SELECT * FROM T WHERE block_number = 0x123"` | vdb_prover_no_crypto | vdb_compile_to_crypto

The resulting design flow:

1. Design an "idealized" VDB
   (don't think about cryptography)
2. Prove its security in its own idealized model
   (this is often *very* easy)
3. Use the cryptographic compiler
   → get security of final VDB for free

Implication 1: Want to improve the design of a VDB? Improve its building blocks OR the idealized blueprint. Afterwards, *no need to reprove security from scratch*.



Zooming in on Simplicity

A Second Lens—Modularity (continued)

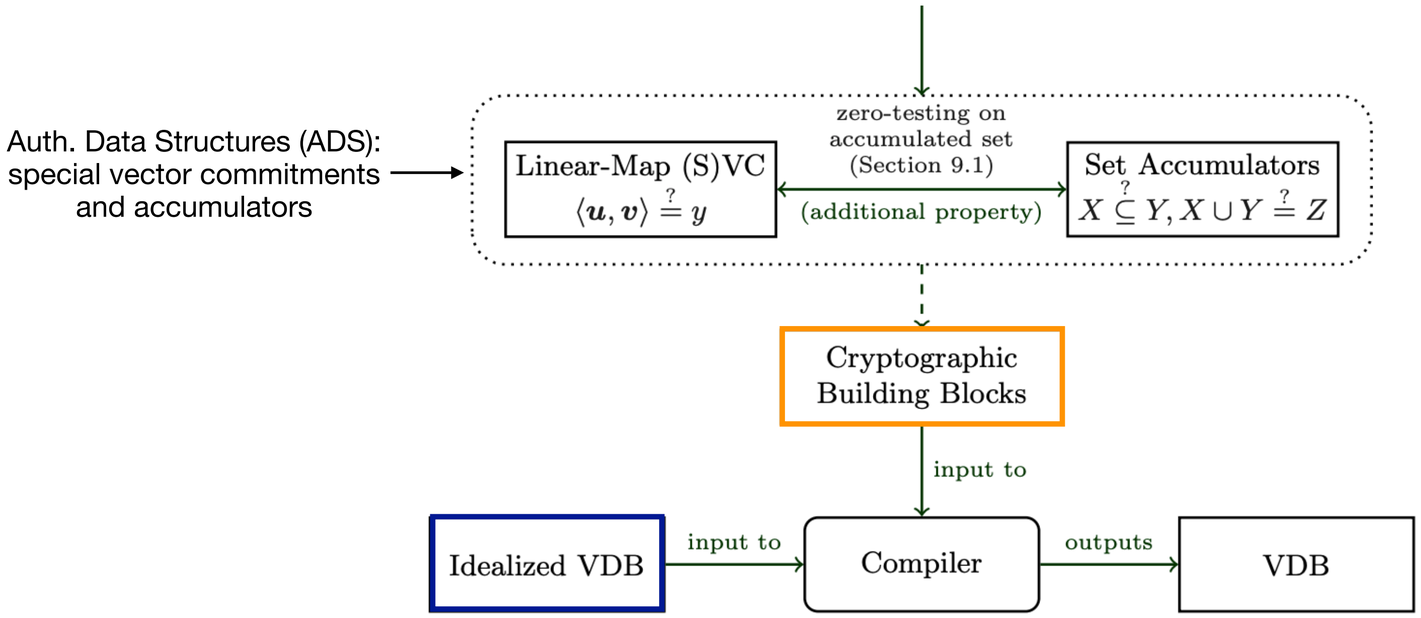
`echo "SELECT * FROM T WHERE block_number = 0x123"` | vdb_prover_no_crypto | vdb_compile_to_crypto

The resulting design flow:

1. Design an "idealized" VDB
   (don't think about cryptography)
2. Prove its security in its own idealized model
   (this is often *very* easy)
3. Use the cryptographic compiler
   → get security of final VDB for free

Implication 1: Want to improve the design of a VDB? Improve its building blocks OR the idealized blueprint. Afterwards, *no need to reprove security from scratch*.

But we can push modularity further!



Zooming in on Simplicity

A Second Lens—Modularity (continued)

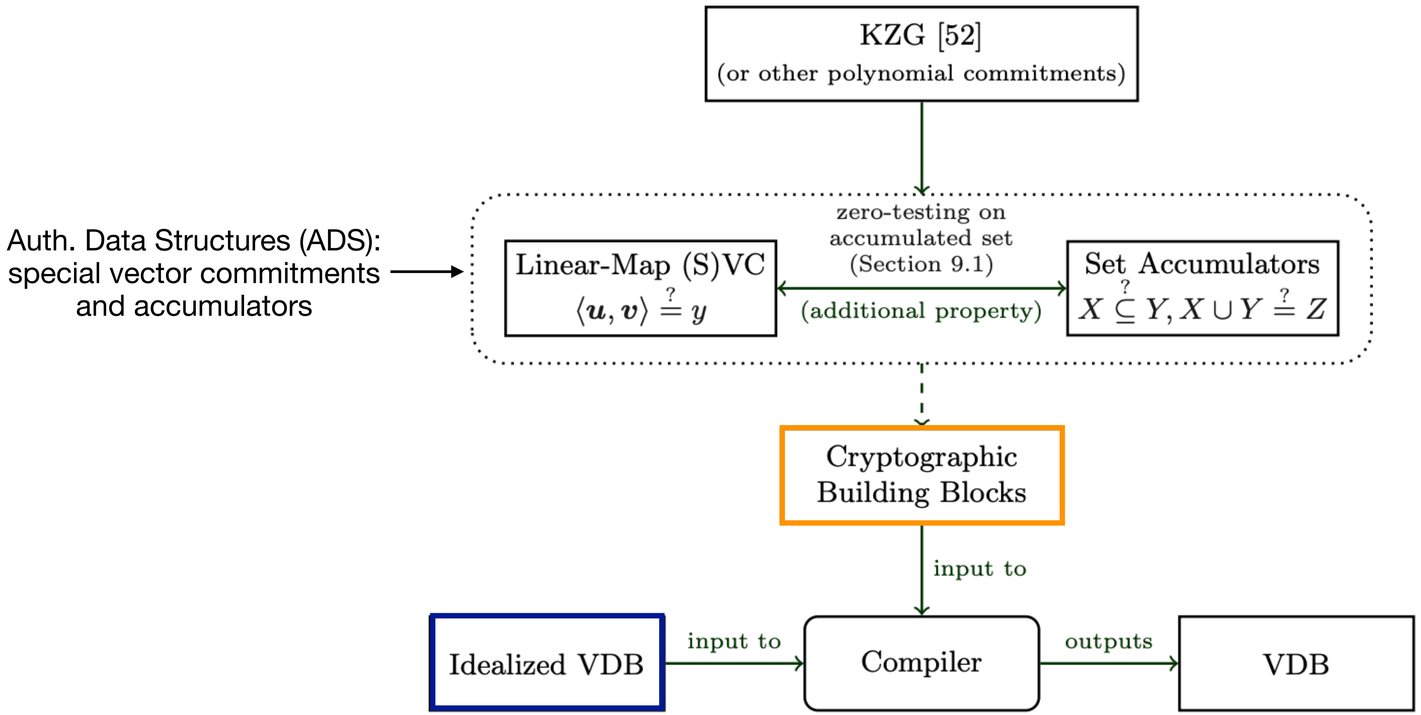
echo “SELECT * FROM T WHERE block_number = 0x123” | vdb_prover_no_crypto | vdb_compile_to_crypto

The resulting design flow:

1. Design an “idealized” VDB
   (don’t think about cryptography)
2. Prove its security in its own idealized model
   (this is often *very* easy)
3. Use the cryptographic compiler
   → get security of final VDB for free

Implication 1: Want to improve the design of a VDB? Improve its building blocks OR the idealized blueprint. Afterwards, *no need to reprove security from scratch*.

But we can push modularity further!



Zooming in on Simplicity

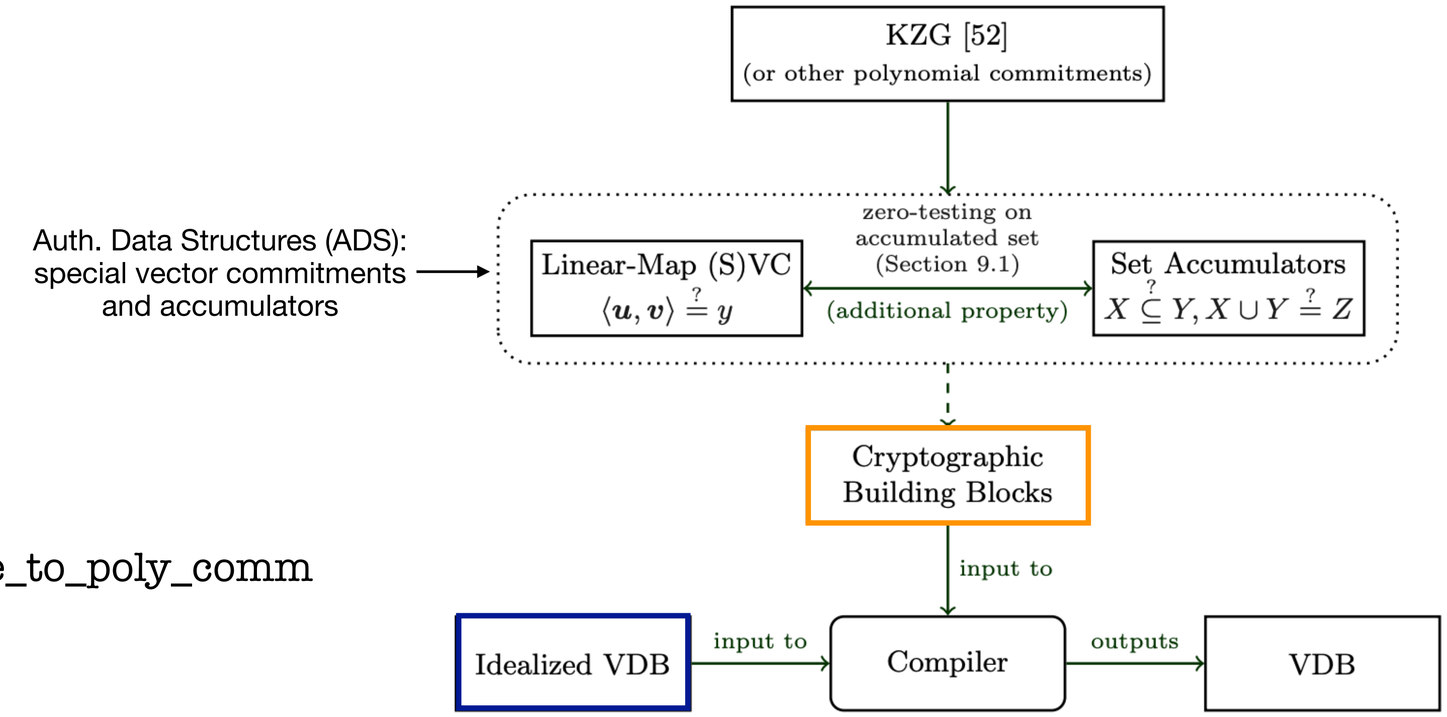
A Second Lens—Modularity (continued)

echo “SELECT * FROM T WHERE block_number = 0x123” | vdb_prover_no_crypto | vdb_compile_to_crypto

The resulting design flow:

1. Design an “idealized” VDB
   (don’t think about cryptography)
2. Prove its security in its own idealized model
   (this is often *very* easy)
3. Use the cryptographic compiler
   → get security of final VDB for free

Implication 1: Want to improve the design of a VDB? Improve its building blocks OR the idealized blueprint. Afterwards, *no need to reprove security from scratch*.



But we can push modularity further!

vdb_compile_to_crypto = compile_to_ADS | compile_to_poly_comm

Zooming in on Simplicity

A Second Lens—Modularity (continued)

echo “SELECT * FROM T WHERE block_number = 0x123” | vdb_prover_no_crypto | vdb_compile_to_crypto

The resulting design flow:

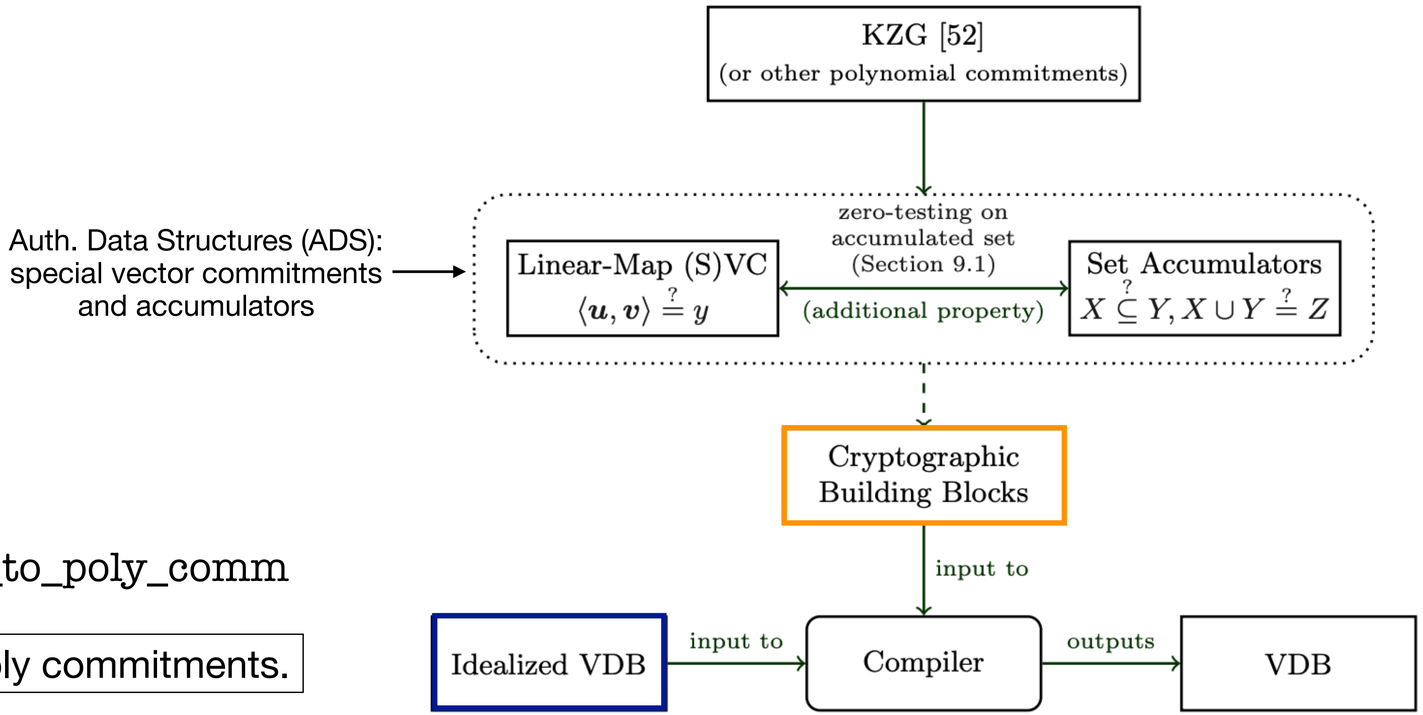
1. Design an “idealized” VDB
   (don’t think about cryptography)
2. Prove its security in its own idealized model
   (this is often *very* easy)
3. Use the cryptographic compiler
   → get security of final VDB for free

Implication 1: Want to improve the design of a VDB? Improve its building blocks OR the idealized blueprint. Afterwards, *no need to reprove security from scratch*.

But we can push modularity further!

vdb_compile_to_crypto = compile_to_ADS | compile_to_poly_comm

Implication 2: Want better building blocks? Just improve poly commitments.



Zooming in on Simplicity

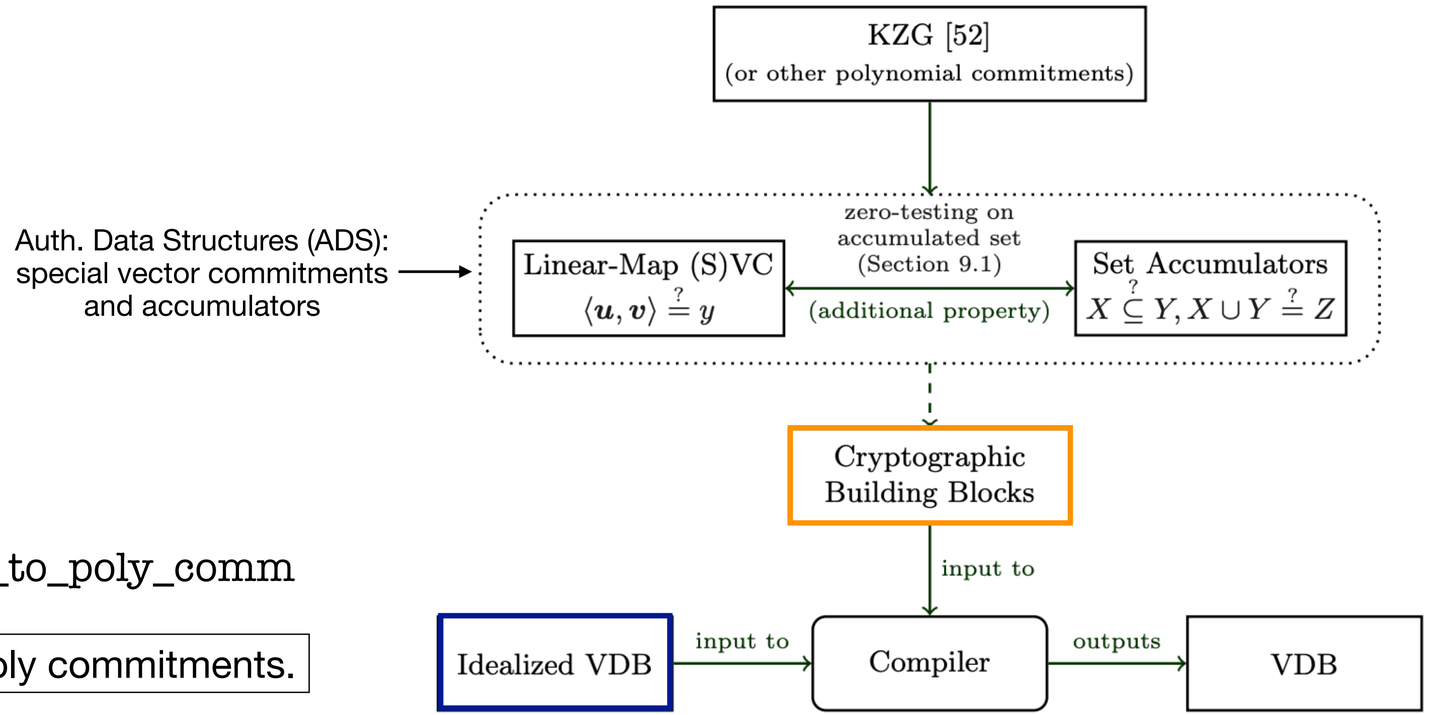
A Second Lens—Modularity (continued)

echo “SELECT * FROM T WHERE block_number = 0x123” | vdb_prover_no_crypto | vdb_compile_to_crypto

The resulting design flow:

1. Design an “idealized” VDB
   (don’t think about cryptography)
2. Prove its security in its own idealized model
   (this is often *very* easy)
3. Use the cryptographic compiler
   → get security of final VDB for free

Implication 1: Want to improve the design of a VDB? Improve its building blocks OR the idealized blueprint. Afterwards, *no need to reprove security from scratch*.



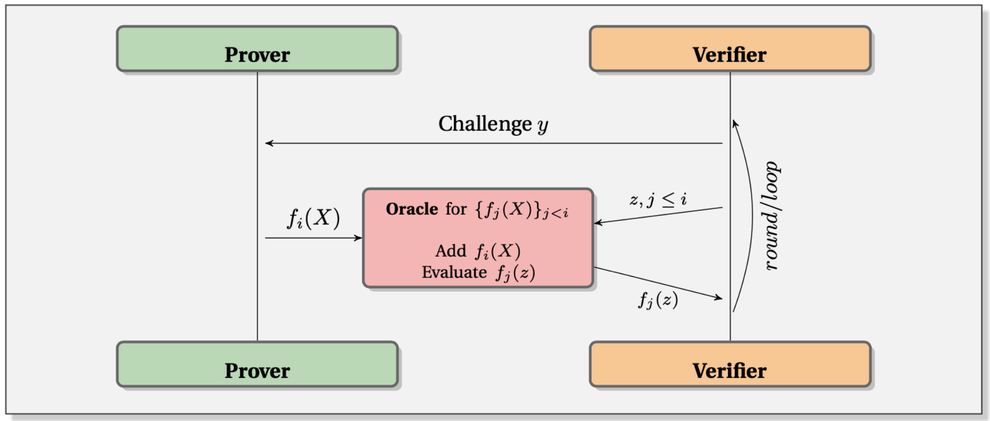
But we can push modularity further!

vdb_compile_to_crypto = compile_to_ADS | compile_to_poly_comm

Implication 2: Want better building blocks? Just improve poly commitments.

Implication 3: *Get post-quantum VDB for free!* (from lattice-based poly commitments)

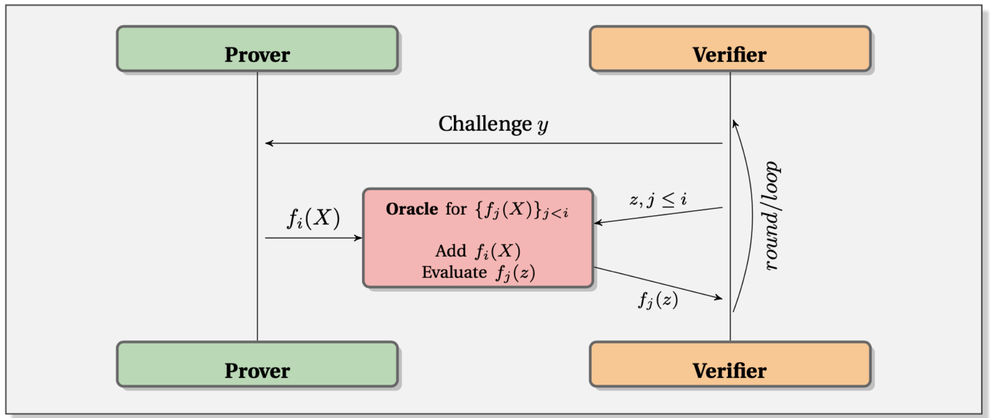
Quick Digression: Idealized Models in SNARKs



Idealized model for SNARKs (Polynomial Interactive Oracle Proofs, or PIOP)
[Source: Anca Nitulescu's *zkSNARKs: A Gentle Introduction*]

Quick Digression: Idealized Models in SNARKs

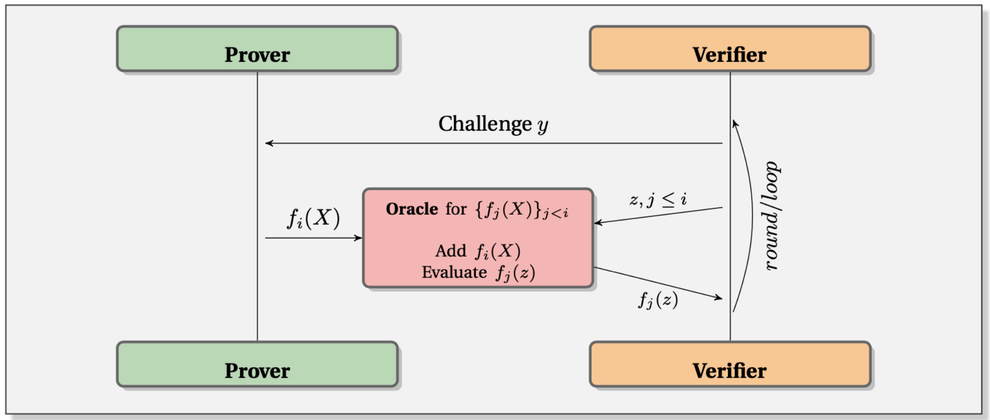
- Splitting “idealized” and cryptographic parts is not uncommon in SNARKs



Idealized model for SNARKs (Polynomial Interactive Oracle Proofs, or PIOP)
[Source: Anca Nitulescu’s *zkSNARKs: A Gentle Introduction*]

Quick Digression: Idealized Models in SNARKs

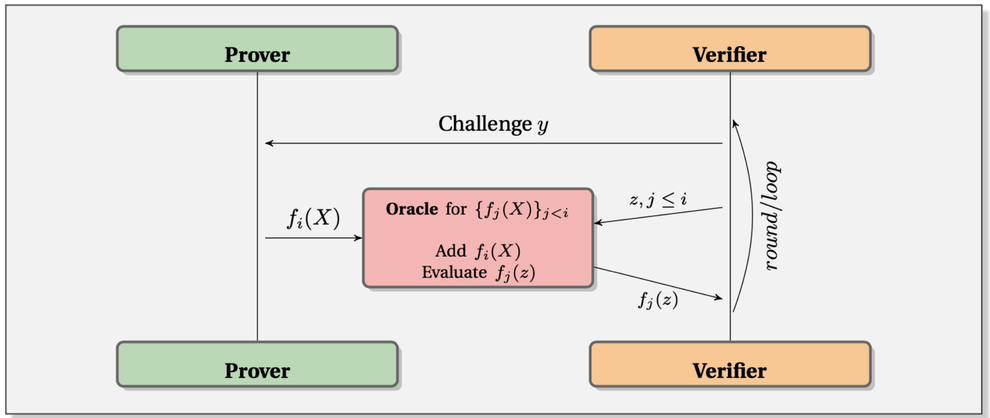
- Splitting “idealized” and cryptographic parts is not uncommon in SNARKs
- **So what’s new here?**



Idealized model for SNARKs (Polynomial Interactive Oracle Proofs, or PIOP)
[Source: Anca Nitulescu’s *zkSNARKs: A Gentle Introduction*]

Quick Digression: Idealized Models in SNARKs

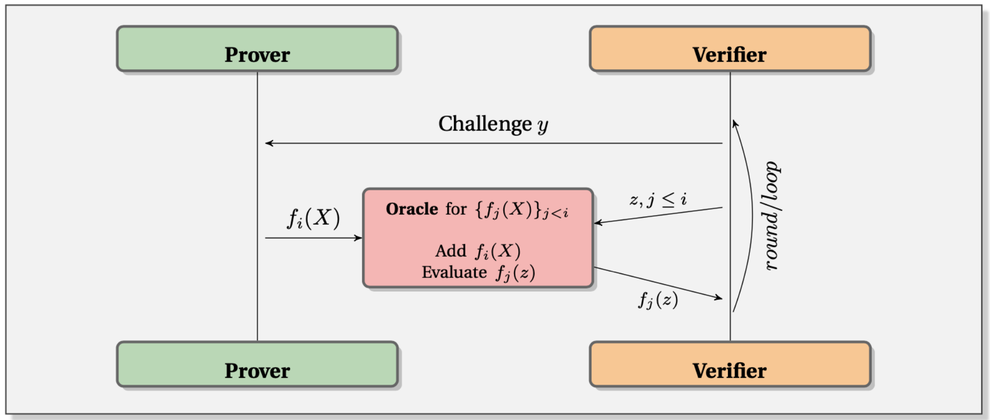
- Splitting “idealized” and cryptographic parts is not uncommon in SNARKs
- **So what’s new here?**
  - This was not a design pattern among VDBs from ADS



Idealized model for SNARKs (Polynomial Interactive Oracle Proofs, or PIOP)
[Source: Anca Nitulescu’s *zkSNARKs: A Gentle Introduction*]

Quick Digression: Idealized Models in SNARKs

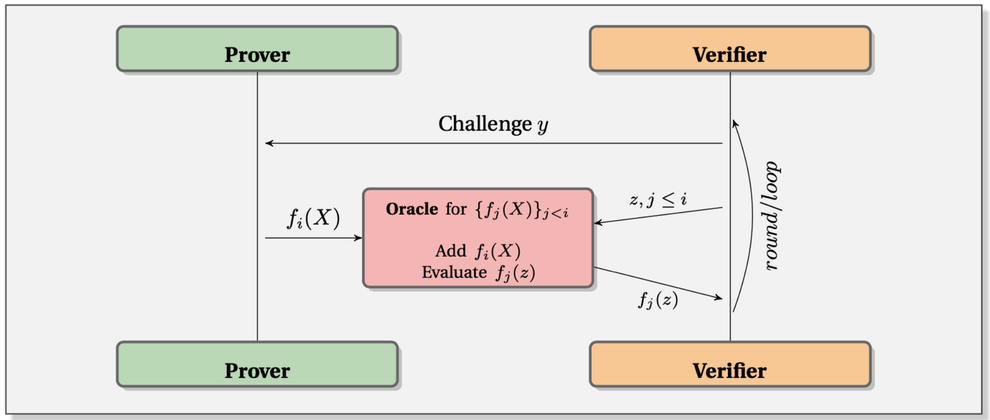
- Splitting “idealized” and cryptographic parts is not uncommon in SNARKs
- **So what’s new here?**
  - This was not a design pattern among VDBs from ADS
  - Our idealized models are *tailored* to the DB setting



Idealized model for SNARKs (Polynomial Interactive Oracle Proofs, or PIOP)
[Source: Anca Nitulescu’s *zkSNARKs: A Gentle Introduction*]

Quick Digression: Idealized Models in SNARKs

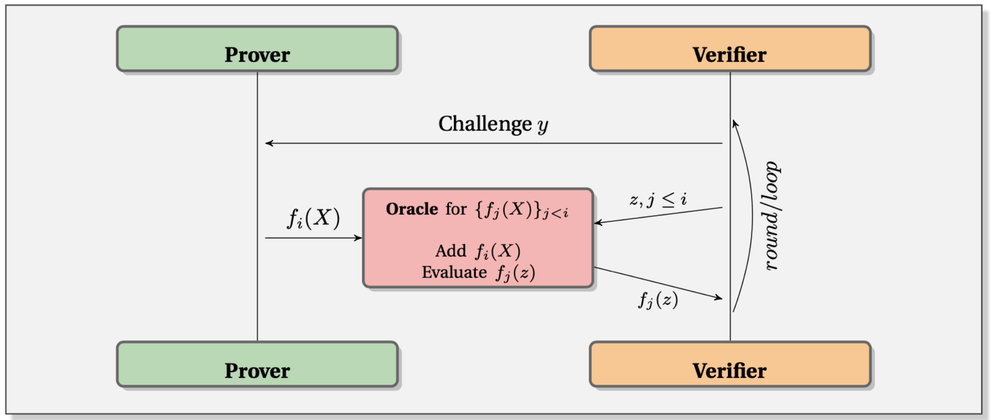
- Splitting “idealized” and cryptographic parts is not uncommon in SNARKs
- **So what’s new here?**
  - This was not a design pattern among VDBs from ADS
  - Our idealized models are *tailored* to the DB setting
    - different “oracles” from those in SNARKs



Idealized model for SNARKs (Polynomial Interactive Oracle Proofs, or PIOP)
[Source: Anca Nitulescu’s *zkSNARKs: A Gentle Introduction*]

Quick Digression: Idealized Models in SNARKs

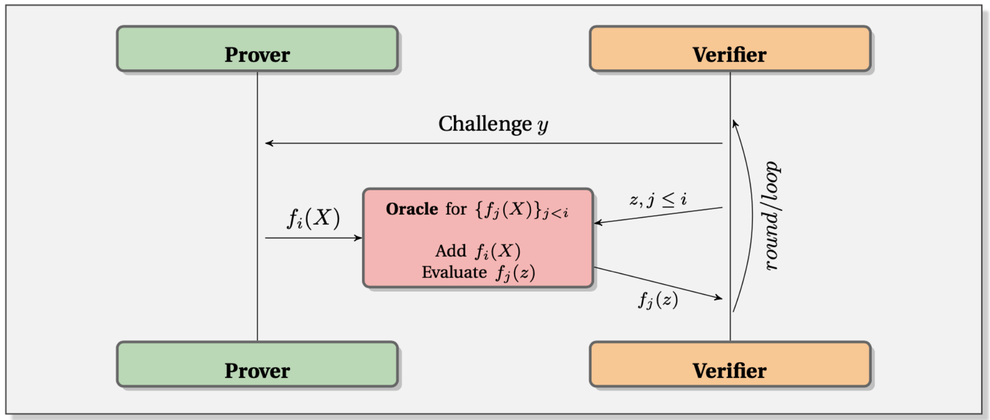
- Splitting “idealized” and cryptographic parts is not uncommon in SNARKs
- **So what’s new here?**
  - This was not a design pattern among VDBs from ADS
  - Our idealized models are *tailored* to the DB setting
    - different “oracles” from those in SNARKs
      - SNARKs: polynomial evaluation



Idealized model for SNARKs (Polynomial Interactive Oracle Proofs, or PIOP)
[Source: Anca Nitulescu’s *zkSNARKs: A Gentle Introduction*]

Quick Digression: Idealized Models in SNARKs

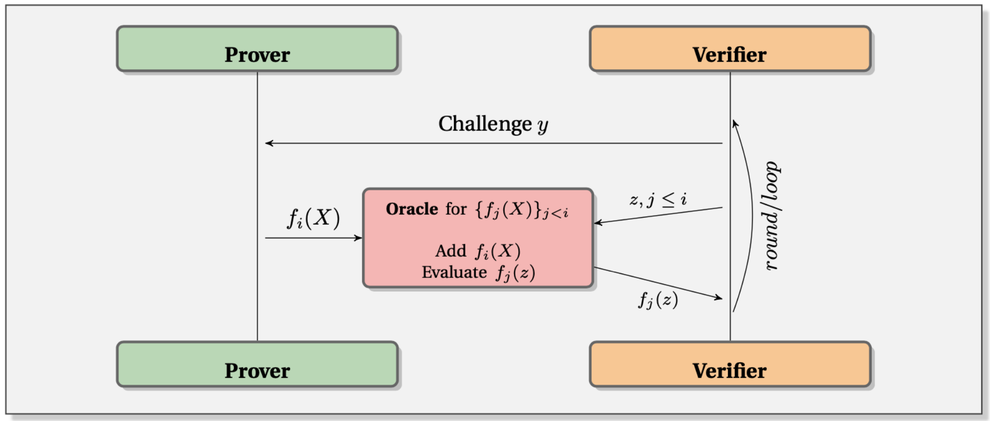
- Splitting “idealized” and cryptographic parts is not uncommon in SNARKs
- **So what’s new here?**
  - This was not a design pattern among VDBs from ADS
  - Our idealized models are *tailored* to the DB setting
    - different “oracles” from those in SNARKs
      - SNARKs: polynomial evaluation
      - ours: vectors/sets-based (next slides)



Idealized model for SNARKs (Polynomial Interactive Oracle Proofs, or PIOP)
[Source: Anca Nitulescu’s *zkSNARKs: A Gentle Introduction*]

Quick Digression: Idealized Models in SNARKs

- Splitting “idealized” and cryptographic parts is not uncommon in SNARKs
- **So what’s new here?**
  - This was not a design pattern among VDBs from ADS
  - Our idealized models are *tailored* to the DB setting
    - different “oracles” from those in SNARKs
      - SNARKs: polynomial evaluation
      - ours: vectors/sets-based (next slides)
    - **Implication:** no algebra; simpler model to reason about

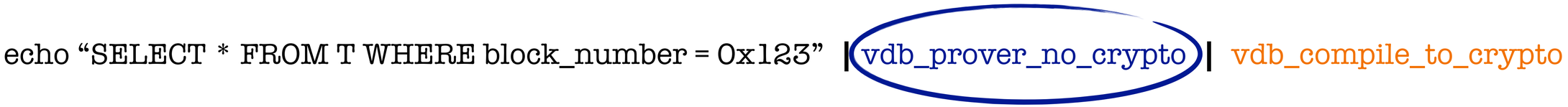


Idealized model for SNARKs (Polynomial Interactive Oracle Proofs, or PIOP)
[Source: Anca Nitulescu’s *zkSNARKs: A Gentle Introduction*]

So Far

- Landscape of prior VDB design and limitations
- Quick overview of efficiency and design philosophy of qedb
- **NEXT:** more on idealized protocols for VDBs and intuitions about how qedb works

Idealized VDBs



A Warm-up Example for Idealized VDBs

Consider a “query template” such as this:

```
SELECT C FROM T WHERE SomeCondition
```

A Warm-up Example for Idealized VDBs

Consider a “query template” such as this:

```
SELECT C FROM T WHERE SomeCondition
```

example for **`SomeCondition:`**

`C1 > 4 AND C2 = “OCL”`

A Warm-up Example for Idealized VDBs

Consider a “query template” such as this:

```
SELECT C FROM T WHERE SomeCondition
```

example for **`SomeCondition:`**

`C1 > 4 AND C2 = “OCL”`

Client will receive a response Resp like this:

$$\text{Resp} = \left(X, (y_r)_{r \in X}\right)$$

A Warm-up Example for Idealized VDBs

Consider a “query template” such as this:

```
SELECT C FROM T WHERE SomeCondition
```

example for **`SomeCondition:`**

`C1 > 4 AND C2 = “OCL”`

Client will receive a response Resp like this:

$$\text{Resp} = \left(X, \left(y_r\right)_{r \in X}\right)$$

(*Claimed*) Set of relevant rows from column C

A Warm-up Example for Idealized VDBs

Consider a “query template” such as this:

```
SELECT C FROM T WHERE SomeCondition
```

example for **`SomeCondition:`**

`C1 > 4 AND C2 = “OCL”`

Client will receive a response Resp like this:

$$\text{Resp} = \left(X, \left(y_r\right)_{r \in X}\right)$$

(*Claimed*) Set of relevant rows from column `C`

Values of column at rows X (also *claimed*)

A Warm-up Example for Idealized VDBs

Consider a “query template” such as this:

```
SELECT C FROM T WHERE SomeCondition
```

example for **`SomeCondition:`**

`C1 > 4 AND C2 = “OCL”`

Client will receive a response Resp like this:

$$\text{Resp} = \left(X, \left(y_r\right)_{r \in X}\right)$$

(*Claimed*) Set of relevant rows from column C

Values of column at rows X (also *claimed*)

What could the client check to be persuaded that Resp is correct?

A Warm-up Example for Idealized VDBs

Consider a “query template” such as this:

```
SELECT C FROM T WHERE SomeCondition
```

example for **`SomeCondition:`**

`C1 > 4 AND C2 = “OCL”`

Client will receive a response Resp like this:

$$\text{Resp} = \left(X, \left(y_r\right)_{r \in X}\right)$$

(*Claimed*) Set of relevant rows from column C

Values of column at rows X (also *claimed*)

What could the client check to be persuaded that Resp is correct?

1. $\text{SomeCondition}(r) = \top \iff r \in X$ (*right rows?)*
2. $\forall r \in X \quad y_r = \text{C}[r]$ (*right values?)*

A Warm-up Example for Idealized VDBs
(continued)

```
SELECT C FROM T WHERE SomeCondition
```

We want to check:

1. SomeCondition$(r) = \top \iff r \in X$ (*right rows?)*
2. $\forall r \in X \quad y_r = \mathsf{C}[r]$ (*right values?)*

$$\text{Resp} = \left(X, (y_r)_{r \in X}\right)$$

(*Claimed*) Set of relevant rows from column **C**

Values of column at rows X (also, *claimed*)

A Warm-up Example for Idealized VDBs
(continued)

```
SELECT C FROM T WHERE SomeCondition
```

$$\text{Resp} = \left(X, \left(y_r\right)_{r \in X}\right)$$

(*Claimed*) Set of relevant rows from column **C**

Values of column at rows X (also, *claimed*)

We want to check:

1. $\text{SomeCondition}(r) = \top \iff r \in X$ (*right rows?)*
2. $\forall r \in X \quad y_r = \text{C}[r]$ (*right values?)*

Let's endow server/client with "special powers":

A Warm-up Example for Idealized VDBs
(continued)

```
SELECT C FROM T WHERE SomeCondition
```

$$\text{Resp} = \left(X, \left(y_r\right)_{r \in X}\right)$$

(*Claimed*) Set of relevant rows from column **C**

Values of column at rows X (also, *claimed*)

We want to check:

1. $\text{SomeCondition}(r) = \top \iff r \in X$ (*right rows?)*
2. $\forall r \in X \quad y_r = \text{C}[r]$ (*right values?)*

Let's endow server/client with "special powers":

Prover can send "pointers" (*handles)* to sets and vectors.

A Warm-up Example for Idealized VDBs
(continued)

```
SELECT C FROM T WHERE SomeCondition
```

$$\text{Resp} = \left(X, \left(y_r\right)_{r \in X}\right)$$

(*Claimed*) Set of relevant rows from column **C**

Values of column at rows X (also, *claimed*)

We want to check:

1. $\text{SomeCondition}(r) = \top \iff r \in X$ (*right rows?)*
2. $\forall r \in X \quad y_r = \text{C}[r]$ (*right values?)*

Let's endow server/client with "special powers":

Prover can send "pointers" (*handles)* to sets and vectors.

X handle to a set

$\boldsymbol{v}$ handle to a vector

A Warm-up Example for Idealized VDBs
(continued)

```
SELECT C FROM T WHERE SomeCondition
```

$$\text{Resp} = \left(X, \left(y_r\right)_{r \in X}\right)$$

(*Claimed*) Set of relevant rows from column **C**

Values of column at rows X (also, *claimed*)

We want to check:

1. $\text{SomeCondition}(r) = \top \iff r \in X$ (*right rows?)*
2. $\forall r \in X \quad y_r = \text{C}[r]$ (*right values?)*

Let's endow server/client with "special powers":

Prover can send "pointers" (*handles)* to sets and vectors.

Client can perform special checks on handles.

X handle to a set

$\boldsymbol{v}$ handle to a vector

A Warm-up Example for Idealized VDBs
(continued)

```
SELECT C FROM T WHERE SomeCondition
```

$$\text{Resp} = \left(X, \left(y_r\right)_{r \in X}\right)$$

(*Claimed*) Set of relevant rows from column **C**

Values of column at rows X (also, *claimed*)

We want to check:

1. $\text{SomeCondition}(r) = \top \iff r \in X$ (*right rows?)*
2. $\forall r \in X \quad y_r = \text{C}[r]$ (*right values?)*

Let's endow server/client with "special powers":

Prover can send "pointers" (*handles)* to sets and vectors.

Client can perform special checks on handles.

X handle to a set

$\boldsymbol{v}$ handle to a vector

Example: $\mathbf{read?}(\boldsymbol{u}, X, \boldsymbol{v})$

checks whether $\boldsymbol{u}_X = \boldsymbol{v}$

("If I read from *u* at positions *X* do I get *v?")*

A Warm-up Example for Idealized VDBs
(continued)

```
SELECT C FROM T WHERE SomeCondition
```

We want to check:

1. SomeCondition$(r) = \top \iff r \in X$ (*right rows?)*
2. $\forall r \in X \quad y_r = \mathsf{C}[r]$ (*right values?)*

$$\text{Resp} = \left(X, (y_r)_{r \in X}\right)$$

(*Claimed*) Set of relevant rows from column **C**

Values of column at rows X (also, *claimed*)

Let's endow server/client with "special powers":

Prover can send "pointers" (*handles)* to sets and vectors.

Client can perform special checks on handles.

X handle to a set

$\boldsymbol{v}$ handle to a vector

Example: $\mathbf{read?}(\boldsymbol{u}, X, \boldsymbol{v})$

checks whether $\boldsymbol{u}_X = \boldsymbol{v}$

("If I read from *u* at positions *X* do I get *v?")*

Toy example:

P **V**

A Warm-up Example for Idealized VDBs
(continued)

```
SELECT C FROM T WHERE SomeCondition
```

$$\text{Resp} = \left(X, \left(y_r\right)_{r \in X}\right)$$

(*Claimed*) Set of relevant rows from column **C**

Values of column at rows X (also, *claimed*)

We want to check:

1. $\text{SomeCondition}(r) = \top \iff r \in X$ (*right rows?*)
2. $\forall r \in X \quad y_r = \text{C}[r]$ (*right values?*)

Let's endow server/client with "special powers":

Prover can send "pointers" (*handles)* to sets and vectors.

Client can perform special checks on handles.

X handle to a set

$\boldsymbol{v}$ handle to a vector

Example: $\mathbf{read?}(\boldsymbol{u}, X, \boldsymbol{v})$

checks whether $\boldsymbol{u}_X = \boldsymbol{v}$

("If I read from *u* at positions *X* do I get *v?")*

Toy example:

P

V

$\boldsymbol{v} := (9, 16, 25, 36, 49)$

A Warm-up Example for Idealized VDBs
(continued)

```
SELECT C FROM T WHERE SomeCondition
```

$$\text{Resp} = \left(X, (y_r)_{r \in X}\right)$$

(*Claimed*) Set of relevant rows from column **C**

Values of column at rows X (also, *claimed*)

We want to check:

1. $\text{SomeCondition}(r) = \top \iff r \in X$ (*right rows?*)
2. $\forall r \in X \quad y_r = \text{C}[r]$ (*right values?*)

Let's endow server/client with "special powers":

Prover can send "pointers" (*handles)* to sets and vectors.

Client can perform special checks on handles.

X handle to a set

$\boldsymbol{v}$ handle to a vector

Example: $\mathbf{read?}(\boldsymbol{u}, X, \boldsymbol{v})$

checks whether $\boldsymbol{u}_X = \boldsymbol{v}$

("If I read from *u* at positions *X* do I get *v?")*

Toy example:

P

$\boldsymbol{u} := (1^2, 2^2, \ldots, 99^2, 100^2)$

$X := \{3, 4, 5, 6, 7\}$

V

$\boldsymbol{v} := (9, 16, 25, 36, 49)$

A Warm-up Example for Idealized VDBs
(continued)

```
SELECT C FROM T WHERE SomeCondition
```

$$\text{Resp} = \left(X, \left(y_r\right)_{r \in X}\right)$$

(*Claimed*) Set of relevant rows from column **C**

Values of column at rows X (also, *claimed*)

We want to check:

1. $\text{SomeCondition}(r) = \top \iff r \in X$ (*right rows?*)
2. $\forall r \in X \quad y_r = \text{C}[r]$ (*right values?*)

Let's endow server/client with "special powers":

Prover can send "pointers" (*handles)* to sets and vectors.

Client can perform special checks on handles.

X handle to a set

$\boldsymbol{v}$ handle to a vector

Example: $\mathbf{read?}(\boldsymbol{u}, X, \boldsymbol{v})$

checks whether $\boldsymbol{u}_X = \boldsymbol{v}$

("If I read from *u* at positions *X* do I get *v?")*

Toy example:

P

$\boldsymbol{u} := (1^2, 2^2, \ldots, 99^2, 100^2)$

$X := \{3, 4, 5, 6, 7\}$

Send $\boldsymbol{u}, X$ →

V

$\boldsymbol{v} := (9, 16, 25, 36, 49)$

A Warm-up Example for Idealized VDBs
(continued)

```
SELECT C FROM T WHERE SomeCondition
```

$$\text{Resp} = \left(X, \left(y_r\right)_{r \in X}\right)$$

(*Claimed*) Set of relevant rows from column **C**

Values of column at rows X (also, *claimed*)

We want to check:

1. $\text{SomeCondition}(r) = \top \iff r \in X$ (*right rows?*)
2. $\forall r \in X \quad y_r = \text{C}[r]$ (*right values?*)

Let's endow server/client with "special powers":

Prover can send "pointers" (*handles)* to sets and vectors.

Client can perform special checks on handles.

X handle to a set

$\boldsymbol{v}$ handle to a vector

Example: $\mathbf{read?}(\boldsymbol{u}, X, \boldsymbol{v})$

checks whether $\boldsymbol{u}_X = \boldsymbol{v}$

("If I read from *u* at positions *X* do I get *v?")*

Toy example:

P

$\boldsymbol{u} := (1^2, 2^2, \ldots, 99^2, 100^2)$

$X := \{3, 4, 5, 6, 7\}$

Send $\boldsymbol{u}, X$ →

V

$\boldsymbol{v} := (9, 16, 25, 36, 49)$

Assert $\mathbf{read?}(\boldsymbol{u}, X, \boldsymbol{v})$

A Warm-up Example for Idealized VDBs
(continued)

```
SELECT C FROM T WHERE SomeCondition
```

$$\text{Resp} = \left(X, \left(y_r\right)_{r \in X}\right)$$

(*Claimed*) Set of relevant rows from column **C**

Values of column at rows X (also, *claimed*)

We want to check:

1. $\text{SomeCondition}(r) = \top \iff r \in X$ (*right rows?*)
2. $\forall r \in X \quad y_r = \text{C}[r]$ (*right values?*)

Special handles/"powers"

X handle to a set

v handle to a vector

$\mathbf{read?}(u, X, v)$ checks whether $u_X = v$

[plus more checks, that are not relevant at the moment]

A Warm-up Example for Idealized VDBs
(continued)

```
SELECT C FROM T WHERE SomeCondition
```

$$\text{Resp} = \left(X, (y_r)_{r \in X}\right)$$

(*Claimed*) Set of relevant rows from column **C**

Values of column at rows X (also, *claimed*)

We want to check:

1. $\text{SomeCondition}(r) = \top \iff r \in X$ (*right rows?*)
2. $\forall r \in X \quad y_r = \text{C}[r]$ (*right values?*)

Special handles/"powers"

X handle to a set

v handle to a vector

$\mathbf{read?}(u, X, v)$ checks whether $u_X = v$

[plus more checks, that are not relevant at the moment]

A protocol sketch for the query above:

P `SELECT C FROM T WHERE SomeCondition` **V**

A Warm-up Example for Idealized VDBs
(continued)

```
SELECT C FROM T WHERE SomeCondition
```

$$\text{Resp} = \left(X, (y_r)_{r \in X}\right)$$

(*Claimed*) Set of relevant rows from column C

Values of column at rows X (also, *claimed*)

We want to check:

1. $\text{SomeCondition}(r) = \top \iff r \in X$ (*right rows?*)
2. $\forall r \in X \quad y_r = \text{C}[r]$ (*right values?*)

Special handles/"powers"

X handle to a set

v handle to a vector

$\mathbf{read?}(u, X, v)$ checks whether $u_X = v$

[plus more checks, that are not relevant at the moment]

A protocol sketch for the query above:

P `SELECT C FROM T WHERE SomeCondition` **V**

Preprocessing: V holds a handle v_C to the values in column C from an **offline stage**

A Warm-up Example for Idealized VDBs (continued)

```
SELECT C FROM T WHERE SomeCondition
```

$$\text{Resp} = \left(X, (y_r)_{r \in X}\right)$$

(*Claimed*) Set of relevant rows from column C

Values of column at rows X (also, *claimed*)

We want to check:

1. $\text{SomeCondition}(r) = \top \iff r \in X$ (*right rows?*)
2. $\forall r \in X \quad y_r = \text{C}[r]$ (*right values?*)

Special handles/"powers"

X handle to a set

$\boldsymbol{v}$ handle to a vector

$\mathbf{read?}(\boldsymbol{u}, X, \boldsymbol{v})$ checks whether $\boldsymbol{u}_X = \boldsymbol{v}$

[plus more checks, that are not relevant at the moment]

A protocol sketch for the query above:

P

```
SELECT C FROM T WHERE SomeCondition
```

$\text{Resp} := \left(X, (y_r)_{r \in X}\right)$

V

Preprocessing: V holds a handle $\boldsymbol{v}_{\text{C}}$ to the values in column C from an **offline stage**

A Warm-up Example for Idealized VDBs
(continued)

```
SELECT C FROM T WHERE SomeCondition
```

$$\text{Resp} = \left(X, (y_r)_{r \in X}\right)$$

(*Claimed*) Set of relevant rows from column C

Values of column at rows X (also, *claimed*)

We want to check:

1. $\text{SomeCondition}(r) = \top \iff r \in X$ (*right rows?*)
2. $\forall r \in X \quad y_r = \text{C}[r]$ (*right values?*)

Special handles/"powers"

X handle to a set

v handle to a vector

$\mathbf{read?}(u, X, v)$ checks whether $u_X = v$

[plus more checks, that are not relevant at the moment]

A protocol sketch for the query above:

P — **V**

P: `SELECT C FROM T WHERE SomeCondition` →

P: $\text{Resp} := \left(X, (y_r)_{r \in X}\right)$

P: Send X, y →

Preprocessing: V holds a handle v_C to the values in column C from an **offline stage**

A Warm-up Example for Idealized VDBs
(continued)

```
SELECT C FROM T WHERE SomeCondition
```

$$\text{Resp} = \left(X, (y_r)_{r \in X}\right)$$

(*Claimed*) Set of relevant rows from column C

Values of column at rows X (also, *claimed*)

We want to check:

1. $\text{SomeCondition}(r) = \top \iff r \in X$ (*right rows?*)
2. $\forall r \in X \quad y_r = \text{C}[r]$ (*right values?*)

Special handles/"powers"

X handle to a set

v handle to a vector

$\mathbf{read?}(u, X, v)$ checks whether $u_X = v$

[plus more checks, that are not relevant at the moment]

A protocol sketch for the query above:

P

```
SELECT C FROM T WHERE SomeCondition
```

Resp := $\left(X, (y_r)_{r \in X}\right)$

Send X, y →

V

Preprocessing: V holds a handle v_{C} to the values in column C from an **offline stage**

Assert $\mathbf{read?}(v_{\text{C}}, X, y)$ *// checks condition 2 above*

A Warm-up Example for Idealized VDBs (continued)

```
SELECT C FROM T WHERE SomeCondition
```

$\text{Resp} = \left(X, (y_r)_{r \in X}\right)$

(*Claimed*) Set of relevant rows from column C

Values of column at rows X (also, *claimed*)

We want to check:

1. $\text{SomeCondition}(r) = \top \iff r \in X$ (*right rows?*)
2. $\forall r \in X \quad y_r = \text{C}[r]$ (*right values?*)

Special handles/"powers"

X handle to a set

v handle to a vector

$\mathbf{read?}(u, X, v)$ checks whether $u_X = v$

[plus more checks, that are not relevant at the moment]

A protocol sketch for the query above:

P

```
SELECT C FROM T WHERE SomeCondition
```

$\text{Resp} := \left(X, (y_r)_{r \in X}\right)$

Send X, y →

V

Preprocessing: V holds a handle v_C to the values in column C from an **offline stage**

Assert $\mathbf{read?}(v_C, X, y)$ *// checks condition 2 above*

A Warm-up Example for Idealized VDBs (continued)

```
SELECT C FROM T WHERE SomeCondition
```

$$\text{Resp} = \left(X, (y_r)_{r \in X}\right)$$

(*Claimed*) Set of relevant rows from column C

Values of column at rows X (also, *claimed*)

We want to check:

1. $\text{SomeCondition}(r) = \top \iff r \in X$ (*right rows?*)
2. $\forall r \in X \quad y_r = \text{C}[r]$ (*right values?*)

Special handles/"powers"

X handle to a set

v handle to a vector

$\mathbf{read?}(u, X, v)$ checks whether $u_X = v$

[plus more checks, that are not relevant at the moment]

A protocol sketch for the query above:

P

```
SELECT C FROM T WHERE SomeCondition
```

Resp := $\left(X, (y_r)_{r \in X}\right)$

Send X, y →

V

Preprocessing: V holds a handle v_C to the values in column C from an **offline stage**

Assert $\mathbf{read?}(v_C, X, y)$ // *checks condition 2 above*

Run subprotocol for checking $\texttt{SomeCondition}(r) = \top \iff r \in X$

A Warm-up Example for Idealized VDBs
(continued)

```
SELECT C FROM T WHERE SomeCondition
```

$$\text{Resp} = \left(X, (y_r)_{r \in X}\right)$$

(*Claimed*) Set of relevant rows from column C

Values of column at rows X (also, *claimed*)

We want to check:

1. $\text{SomeCondition}(r) = \top \iff r \in X$ (*right rows?*)
2. $\forall r \in X \quad y_r = \text{C}[r]$ (*right values?*)

Special handles/"powers"

X handle to a set

v handle to a vector

$\mathbf{read?}(u, X, v)$ checks whether $u_X = v$

[plus more checks, that are not relevant at the moment]

A protocol sketch for the query above:

P

```
SELECT C FROM T WHERE SomeCondition
```

$\text{Resp} := \left(X, (y_r)_{r \in X}\right)$

Send X, y →

V

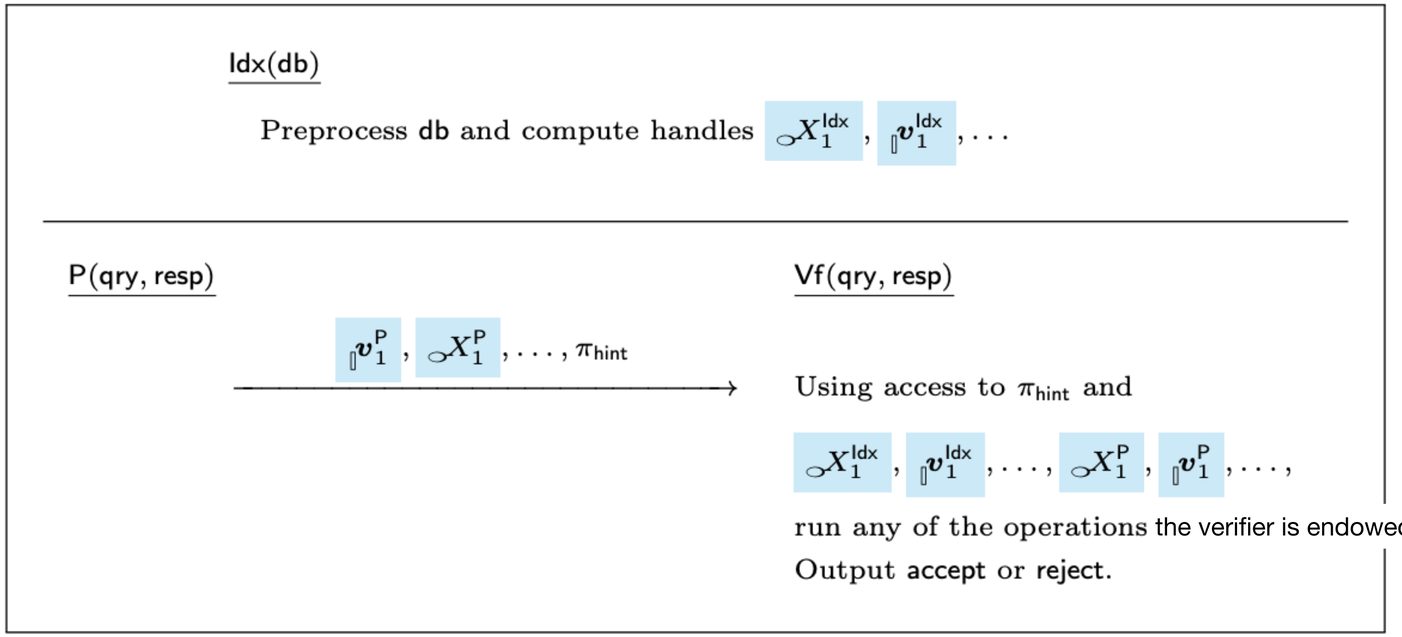
Preprocessing: V holds a handle v_C to the values in column C from an **offline stage**

Assert $\mathbf{read?}(v_C, X, y)$ *// checks condition 2 above*

Run subprotocol for checking $\texttt{SomeCondition}(r) = \top \iff r \in X$

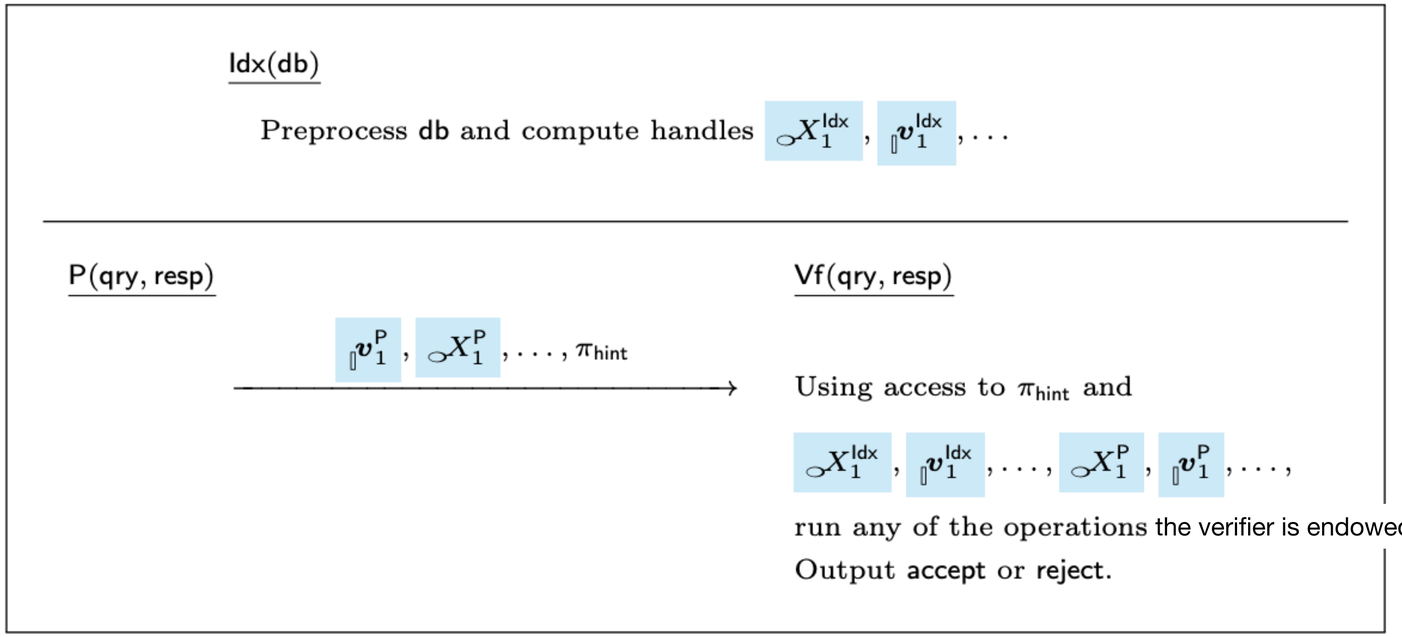
The Flow of an Idealized Protocols

The Flow of an Idealized Protocols



General flow in an idealized protocol

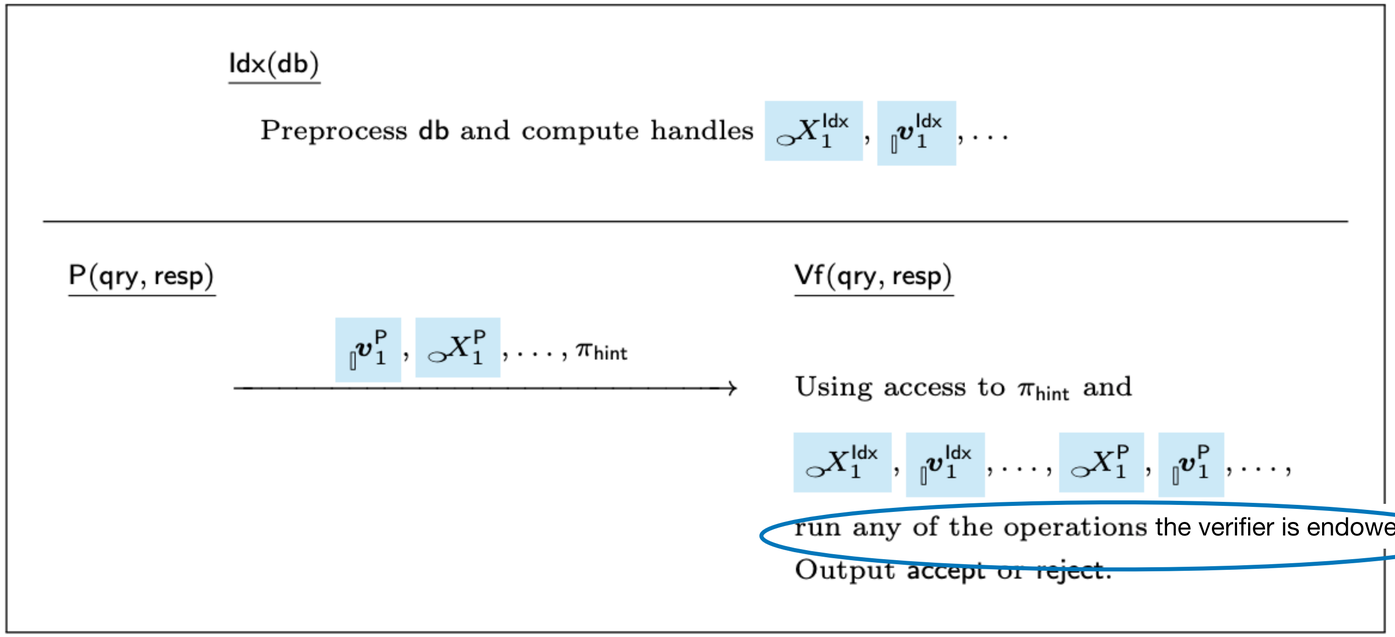
The Flow of an Idealized Protocols



General flow in an idealized protocol

Key role of handles (and their ops):
providing expressivity, but also succinctness.

The Flow of an Idealized Protocols



General flow in an idealized protocol

Key role of handles (and their ops):
providing expressivity, but also succinctness.

“Atomic” Operations in an Idealized VDB

“Atomic” Operations in an Idealized VDB

$$\mathbf{read?}(\boldsymbol{u}, X, \boldsymbol{v}) \ \textit{(read)}$$

“Atomic” Operations in an Idealized VDB

$$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \overset{?}{\subseteq} Y \qquad Z \overset{?}{=} X \cup Y \qquad Z \overset{?}{=} X \cap Y \quad \textit{(set ops)}$$

“Atomic” Operations in an Idealized VDB

$$\mathbf{read?}(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \overset{?}{\subseteq} Y \qquad Z \overset{?}{=} X \cup Y \qquad Z \overset{?}{=} X \cap Y \quad \textit{(set ops)}$$

$$\boldsymbol{u} \overset{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)}$$

“Atomic” Operations in an Idealized VDB

$$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \overset{?}{\subseteq} Y \qquad Z \overset{?}{=} X \cup Y \qquad Z \overset{?}{=} X \cap Y \quad \textit{(set ops)}$$

$$\boldsymbol{u} \overset{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)} \qquad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \overset{?}{=} y \quad \textit{(inner product)}$$

“Atomic” Operations in an Idealized VDB

$$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \overset{?}{\subseteq} Y \qquad Z \overset{?}{=} X \cup Y \qquad Z \overset{?}{=} X \cap Y \quad \textit{(set ops)}$$

$$\boldsymbol{u} \overset{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)} \qquad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \overset{?}{=} y \quad \textit{(inner product)}$$

$$\boldsymbol{v} \left[X \right] \overset{?}{=} \mathbf{0} \quad \textit{(zero test)}$$

“Atomic” Operations in an Idealized VDB

$$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \overset{?}{\subseteq} Y \qquad Z \overset{?}{=} X \cup Y \qquad Z \overset{?}{=} X \cap Y \quad \textit{(set ops)}$$

$$\boldsymbol{u} \overset{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)} \qquad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \overset{?}{=} y \quad \textit{(inner product)}$$

$$\boldsymbol{v} \left[X \right] \overset{?}{=} \mathbf{0} \quad \textit{(zero test)}$$

Q1: Are these operations expressive enough?

“Atomic” Operations in an Idealized VDB

$$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \stackrel{?}{\subseteq} Y \qquad Z \stackrel{?}{=} X \cup Y \qquad Z \stackrel{?}{=} X \cap Y \quad \textit{(set ops)}$$

$$\boldsymbol{u} \stackrel{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)} \qquad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \stackrel{?}{=} y \quad \textit{(inner product)}$$

$$\boldsymbol{v} \left[X \right] \stackrel{?}{=} \mathbf{0} \quad \textit{(zero test)}$$

Q1: Are these operations expressive enough?

Q2: Can we compile them through simple cryptographic building blocks?

Idealized Protocols are Very Expressive

From these:

$$\mathbf{read?}(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \overset{?}{\subseteq} Y \qquad Z \overset{?}{=} X \cup Y \qquad Z \overset{?}{=} X \cap Y \quad \textit{(set ops)}$$

$$\boldsymbol{u} \overset{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)} \qquad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \overset{?}{=} y \quad \textit{(inner product)}$$

$$\boldsymbol{v} \left[X \right] \overset{?}{=} \mathbf{0} \quad \textit{(zero test)}$$

Idealized Protocols are Very Expressive

From these:

$$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \stackrel{?}{\subseteq} Y \qquad Z \stackrel{?}{=} X \cup Y \qquad Z \stackrel{?}{=} X \cap Y \quad \textit{(set ops)}$$

$$\boldsymbol{u} \stackrel{?}{=} \alpha\,\boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)} \qquad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \stackrel{?}{=} y \quad \textit{(inner product)}$$

$$\boldsymbol{v}\left[X\right] \stackrel{?}{=} \mathbf{0} \quad \textit{(zero test)}$$

We can get these (and more):

Idealized Protocols are Very Expressive

From these:

$$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \overset{?}{\subseteq} Y \qquad Z \overset{?}{=} X \cup Y \qquad Z \overset{?}{=} X \cap Y \quad \textit{(set ops)}$$

$$\boldsymbol{u} \overset{?}{=} \alpha\,\boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)} \qquad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \overset{?}{=} y \quad \textit{(inner product)}$$

$$\boldsymbol{v}\left[X\right] \overset{?}{=} \mathbf{0} \quad \textit{(zero test)}$$

We can get these (and more):

$$\alpha \overset{?}{\leq} \boldsymbol{v} \overset{?}{\leq} \beta \quad \textit{(range check)}$$

Idealized Protocols are Very Expressive

From these:

$$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \overset{?}{\subseteq} Y \qquad Z \overset{?}{=} X \cup Y \qquad Z \overset{?}{=} X \cap Y \quad \textit{(set ops)}$$

$$\boldsymbol{u} \overset{?}{=} \alpha\,\boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)} \qquad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \overset{?}{=} y \quad \textit{(inner product)}$$

$$\boldsymbol{v}\left[X\right] \overset{?}{=} \mathbf{0} \quad \textit{(zero test)}$$

We can get these (and more):

$$\alpha \overset{?}{\leq} \boldsymbol{v} \overset{?}{\leq} \beta \quad \textit{(range check)}$$

$$\sum_{j \in X} v_j \overset{?}{=} y \quad \textit{(sum check in target subset)}$$

Idealized Protocols are Very Expressive

From these:

$$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \overset{?}{\subseteq} Y \qquad Z \overset{?}{=} X \cup Y \qquad Z \overset{?}{=} X \cap Y \quad \textit{(set ops)}$$

$$\boldsymbol{u} \overset{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)} \qquad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \overset{?}{=} y \quad \textit{(inner product)}$$

$$\boldsymbol{v} \left[X \right] \overset{?}{=} \mathbf{0} \quad \textit{(zero test)}$$

We can get these (and more):

$$\alpha \overset{?}{\leq} \boldsymbol{v} \overset{?}{\leq} \beta \quad \textit{(range check)}$$

$$\sum_{j \in X} v_j \overset{?}{=} y \quad \textit{(sum check in target subset)}$$

$$X \overset{?}{=} \mathsf{eqSet}\left(\boldsymbol{u}, \boldsymbol{v}\right) \quad \textit{(tests where two slices are equal)}$$

From Idealized to Cryptographic VDB

$$\mathbf{read?}(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \overset{?}{\subseteq} Y \qquad Z \overset{?}{=} X \cup Y \qquad Z \overset{?}{=} X \cap Y \quad \textit{(set ops)}$$

$$\boldsymbol{u} \overset{?}{=} \alpha\,\boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)} \qquad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \overset{?}{=} y \quad \textit{(inner product)}$$

$$\boldsymbol{v}\left[X\right] \overset{?}{=} \mathbf{0} \quad \textit{(zero test)}$$

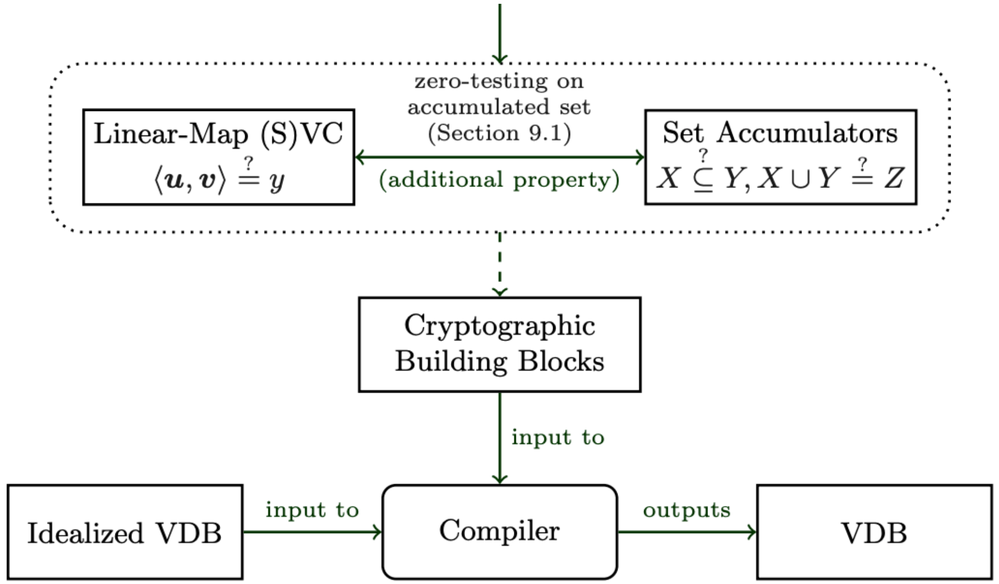
From Idealized to Cryptographic VDB

$$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \overset{?}{\subseteq} Y \qquad Z \overset{?}{=} X \cup Y \qquad Z \overset{?}{=} X \cap Y \quad \textit{(set ops)}$$

$$\boldsymbol{u} \overset{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)} \qquad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \overset{?}{=} y \quad \textit{(inner product)}$$

$$\boldsymbol{v} \left[X \right] \overset{?}{=} \mathbf{0} \quad \textit{(zero test)}$$



From Idealized to Cryptographic VDB

$$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \overset{?}{\subseteq} Y \qquad Z \overset{?}{=} X \cup Y \qquad Z \overset{?}{=} X \cap Y \quad \textit{(set ops)}$$

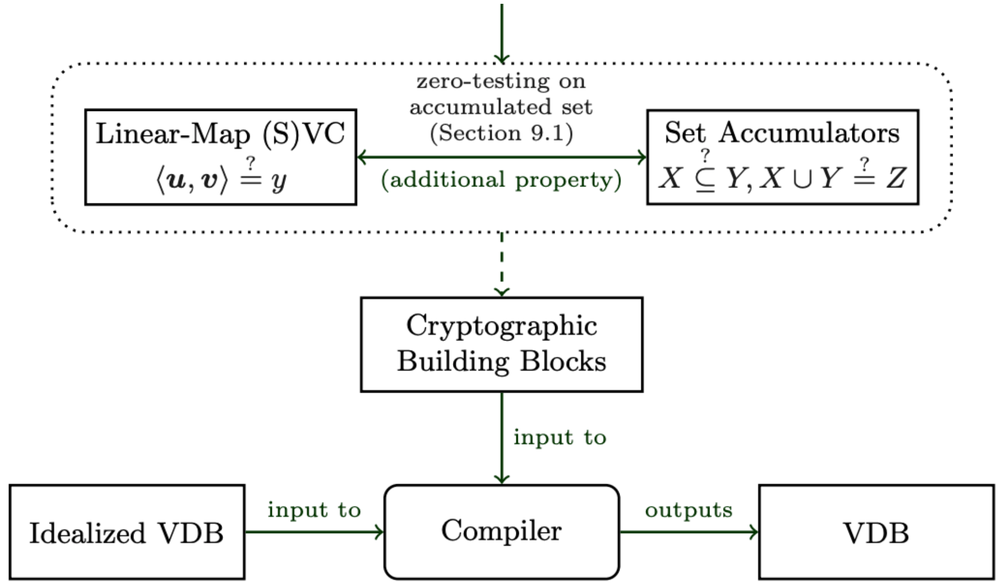
$$\boldsymbol{u} \overset{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)} \qquad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \overset{?}{=} y \quad \textit{(inner product)}$$

$$\boldsymbol{v} \left[X \right] \overset{?}{=} \mathbf{0} \quad \textit{(zero test)}$$

We commit to handles:

X accumulator

$\boldsymbol{v}$ linear-map vector commitment



From Idealized to Cryptographic VDB

$$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \overset{?}{\subseteq} Y \qquad Z \overset{?}{=} X \cup Y \qquad Z \overset{?}{=} X \cap Y \quad \textit{(set ops)}$$

$$\boldsymbol{u} \overset{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)} \qquad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \overset{?}{=} y \quad \textit{(inner product)}$$

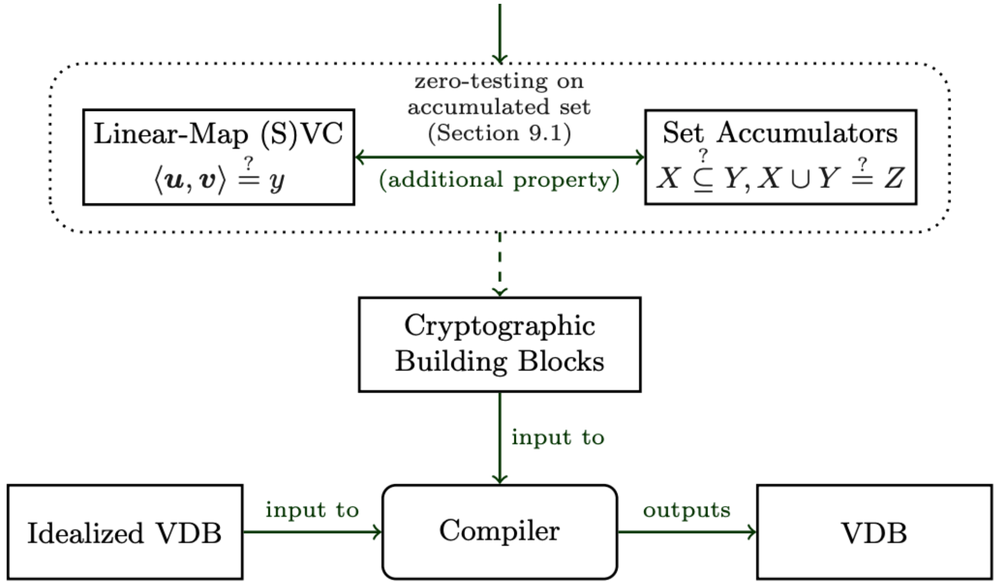
$$\boldsymbol{v} \left[X \right] \overset{?}{=} \mathbf{0} \quad \textit{(zero test)}$$

We commit to handles:

X accumulator

$\boldsymbol{v}$ linear-map vector commitment

Accumulators: $\mathsf{VfySubset}(\mathsf{acc}_X, \mathsf{acc}_Y, \pi_{\text{subset}})$ (checks $X \subseteq Y$)



From Idealized to Cryptographic VDB

$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v})$ *(read)*

$X \stackrel{?}{\subseteq} Y$ $\quad Z \stackrel{?}{=} X \cup Y$ $\quad Z \stackrel{?}{=} X \cap Y$ *(set ops)*

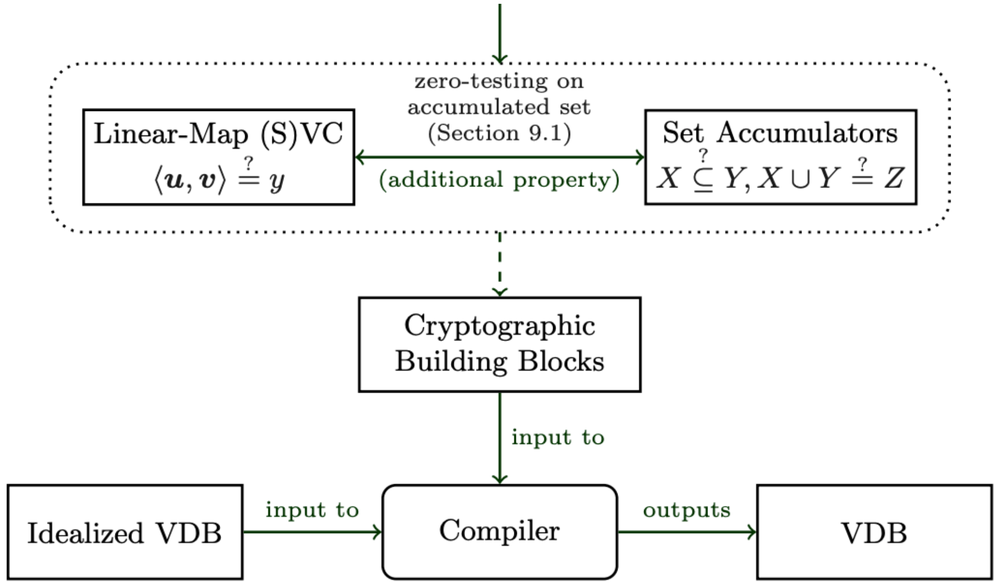
$\boldsymbol{u} \stackrel{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w}$ *(homomorphism)* $\quad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \stackrel{?}{=} y$ *(inner product)*

$\boldsymbol{v} \left[X \right] \stackrel{?}{=} \mathbf{0}$ *(zero test)*

We commit to handles:

X accumulator $\quad \boldsymbol{v}$ linear-map vector commitment

Accumulators: $\mathsf{VfySubset}(\mathsf{acc}_X, \mathsf{acc}_Y, \pi_{\text{subset}})$ (checks $X \subseteq Y$)



From Idealized to Cryptographic VDB

$$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \overset{?}{\subseteq} Y \qquad Z \overset{?}{=} X \cup Y \qquad Z \overset{?}{=} X \cap Y \quad \textit{(set ops)}$$

$$\boldsymbol{u} \overset{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)} \qquad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \overset{?}{=} y \quad \textit{(inner product)}$$

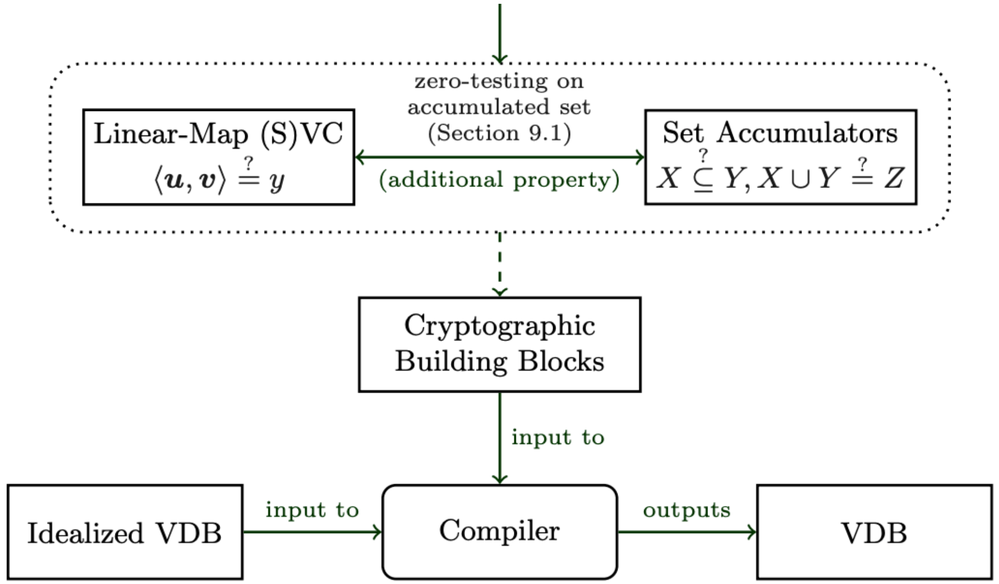
$$\boldsymbol{v} \left[X \right] \overset{?}{=} \mathbf{0} \quad \textit{(zero test)}$$

We commit to handles:

X accumulator

$\boldsymbol{v}$ linear-map vector commitment

Accumulators: $\mathsf{VfySubset}(\mathsf{acc}_X, \mathsf{acc}_Y, \pi_{\text{subset}})$ (checks $X \subseteq Y$)



From Idealized to Cryptographic VDB

$$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \overset{?}{\subseteq} Y \qquad Z \overset{?}{=} X \cup Y \qquad Z \overset{?}{=} X \cap Y \quad \textit{(set ops)}$$

$$\boldsymbol{u} \overset{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)} \qquad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \overset{?}{=} y \quad \textit{(inner product)}$$

$$\boldsymbol{v} \left[X \right] \overset{?}{=} \mathbf{0} \quad \textit{(zero test)}$$

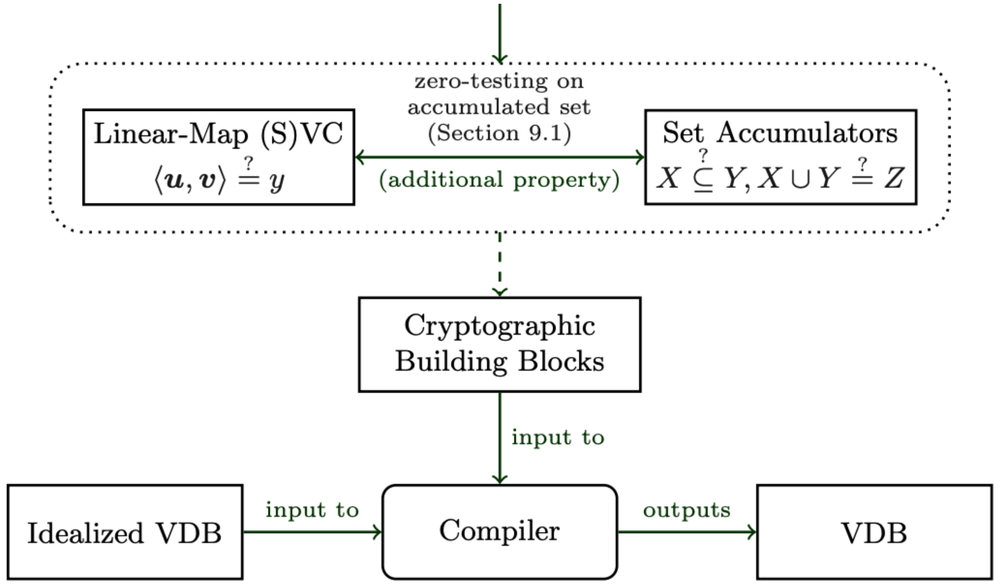
We commit to handles:

X accumulator

$\boldsymbol{v}$ linear-map vector commitment

Accumulators: $\mathsf{VfySubset}(\mathsf{acc}_X, \mathsf{acc}_Y, \pi_{\text{subset}})$ (checks $X \subseteq Y$)

Vector Commitments: $\mathsf{VfySubvec}(\mathsf{cm}_{\boldsymbol{v}}, X, \boldsymbol{y}, \pi_{\text{subvec}})$ (checks $\boldsymbol{v}_X = \boldsymbol{y}$)



From Idealized to Cryptographic VDB

$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v})$ *(read)*

$X \stackrel{?}{\subseteq} Y$ $\quad Z \stackrel{?}{=} X \cup Y$ $\quad Z \stackrel{?}{=} X \cap Y$ *(set ops)*

$\boldsymbol{u} \stackrel{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w}$ *(homomorphism)* $\quad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \stackrel{?}{=} y$ *(inner product)*

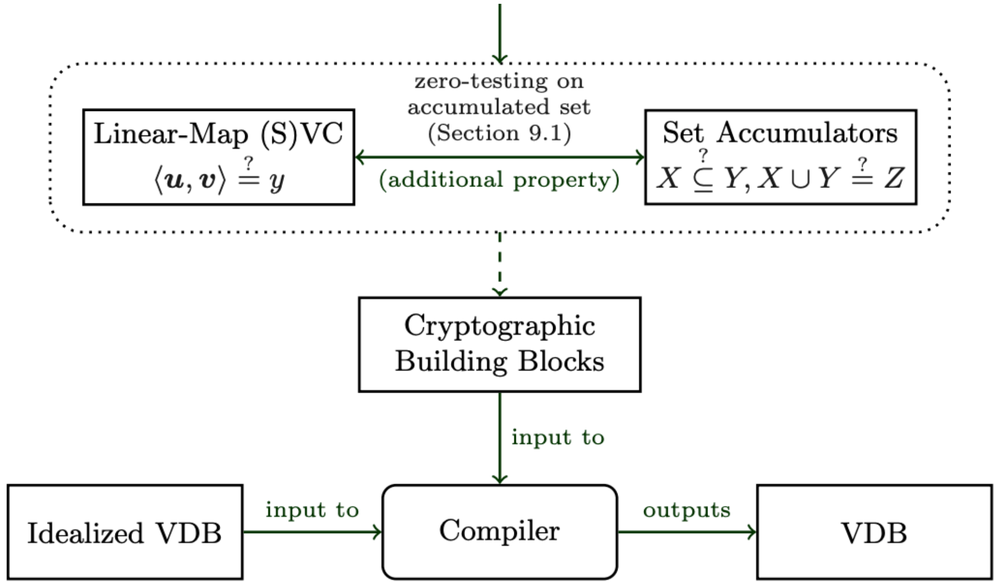
$\boldsymbol{v} \left[X \right] \stackrel{?}{=} \mathbf{0}$ *(zero test)*

We commit to handles:

X accumulator $\quad \boldsymbol{v}$ linear-map vector commitment

Accumulators: $\mathsf{VfySubset}(\mathsf{acc}_X, \mathsf{acc}_Y, \pi_{\text{subset}})$ (checks $X \subseteq Y$)

Vector Commitments: $\mathsf{VfySubvec}(\mathsf{cm}_{\boldsymbol{v}}, X, \boldsymbol{y}, \pi_{\text{subvec}})$ (checks $\boldsymbol{v}_X = \boldsymbol{y}$)



From Idealized to Cryptographic VDB

$$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \overset{?}{\subseteq} Y \qquad Z \overset{?}{=} X \cup Y \qquad Z \overset{?}{=} X \cap Y \quad \textit{(set ops)}$$

$$\boldsymbol{u} \overset{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)} \qquad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \overset{?}{=} y \quad \textit{(inner product)}$$

$$\boldsymbol{v} \left[X \right] \overset{?}{=} \mathbf{0} \quad \textit{(zero test)}$$

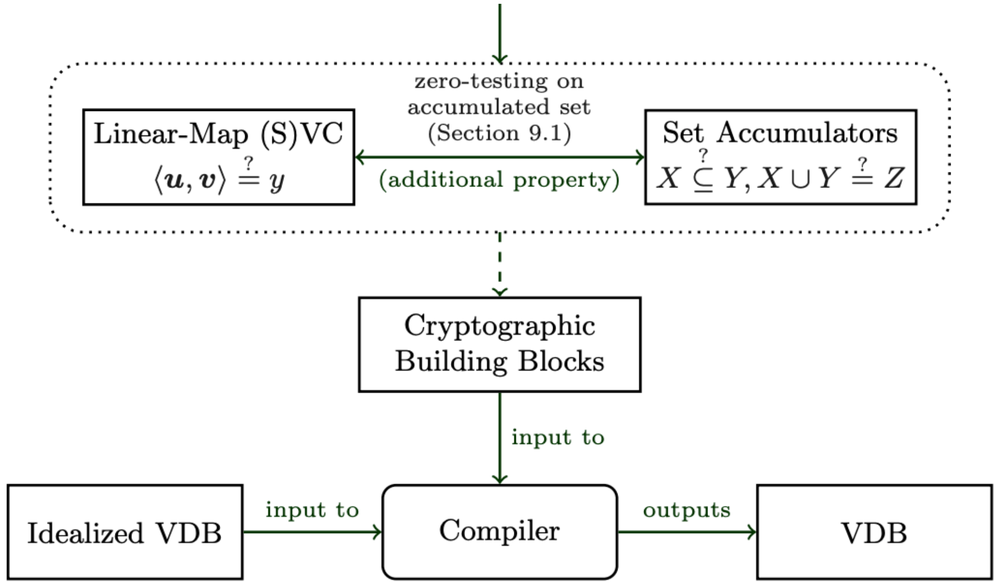
We commit to handles:

X accumulator

$\boldsymbol{v}$ linear-map vector commitment

Accumulators: $\mathsf{VfySubset}(\mathsf{acc}_X, \mathsf{acc}_Y, \pi_{\text{subset}})$ (checks $X \subseteq Y$)

Vector Commitments: $\mathsf{VfySubvec}(\mathsf{cm}_{\boldsymbol{v}}, X, \boldsymbol{y}, \pi_{\text{subvec}})$ (checks $\boldsymbol{v}_X = \boldsymbol{y}$)



From Idealized to Cryptographic VDB

$$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \overset{?}{\subseteq} Y \qquad Z \overset{?}{=} X \cup Y \qquad Z \overset{?}{=} X \cap Y \quad \textit{(set ops)}$$

$$\boldsymbol{u} \overset{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)} \qquad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \overset{?}{=} y \quad \textit{(inner product)}$$

$$\boldsymbol{v} \left[X \right] \overset{?}{=} \mathbf{0} \quad \textit{(zero test)}$$

We commit to handles:

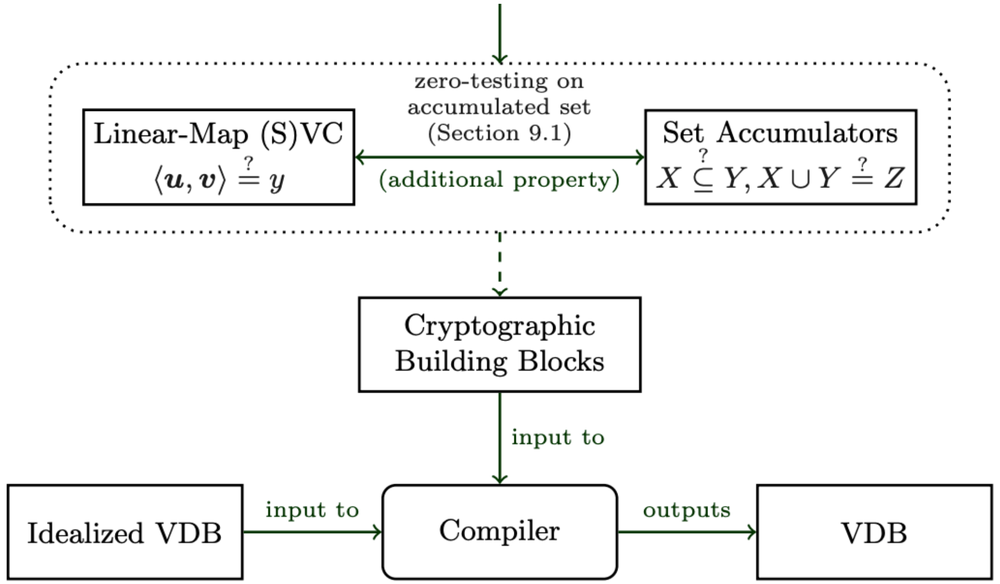
X accumulator

$\boldsymbol{v}$ linear-map vector commitment

Accumulators: $\mathsf{VfySubset}(\mathsf{acc}_X, \mathsf{acc}_Y, \pi_{\text{subset}})$ (checks $X \subseteq Y$)

Vector Commitments: $\mathsf{VfySubvec}(\mathsf{cm}_{\boldsymbol{v}}, X, \boldsymbol{y}, \pi_{\text{subvec}})$ (checks $\boldsymbol{v}_X = \boldsymbol{y}$)

+ linear-map opening: $\mathsf{VfyLin}(\mathsf{cm}_{\boldsymbol{u}}, \mathsf{cm}_{\boldsymbol{v}}, y, \pi_{\text{lin}})$ (checks $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = y$)



From Idealized to Cryptographic VDB

$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v})$ *(read)*

$X \overset{?}{\subseteq} Y$ $\quad Z \overset{?}{=} X \cup Y$ $\quad Z \overset{?}{=} X \cap Y$ *(set ops)*

$\boldsymbol{u} \overset{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w}$ *(homomorphism)* $\quad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \overset{?}{=} y$ *(inner product)*

$\boldsymbol{v} \left[X \right] \overset{?}{=} \mathbf{0}$ *(zero test)*

We

Accu

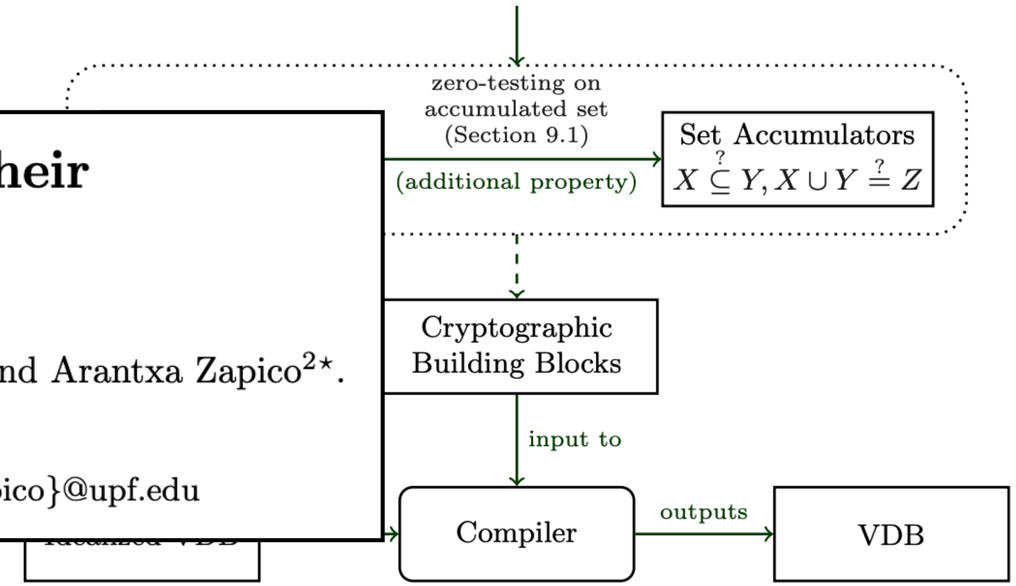
Vector Commitments: $\mathsf{VfySubvec}(\mathsf{cm}_{\boldsymbol{v}}, X, \boldsymbol{y}, \pi_{\mathrm{subvec}})$ (checks $\boldsymbol{v}_X = \boldsymbol{y}$)

+ linear-map opening: $\mathsf{VfyLin}(\mathsf{cm}_{\boldsymbol{u}}, \mathsf{cm}_{\boldsymbol{v}}, y, \pi_{\mathrm{lin}})$ (checks $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = y$)

Linear-map Vector Commitments and their Practical Applications

Matteo Campanelli[1], Anca Nitulescu[1], Carla Ràfols[2], Alexandros Zacharakis[2], and Arantxa Zapico[2⋆].

[1] Protocol Labs {matteo, anca}@protocol.ai
[2] Universitat Pompeu Fabra {carla.rafols, alexandros.zacharakis, arantxa.zapico}@upf.edu



From Idealized to Cryptographic VDB

$$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \overset{?}{\subseteq} Y \qquad Z \overset{?}{=} X \cup Y \qquad Z \overset{?}{=} X \cap Y \quad \textit{(set ops)}$$

$$\boldsymbol{u} \overset{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)} \qquad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \overset{?}{=} y \quad \textit{(inner product)}$$

$$\boldsymbol{v} \left[X \right] \overset{?}{=} \mathbf{0} \quad \textit{(zero test)}$$

We commit to handles:

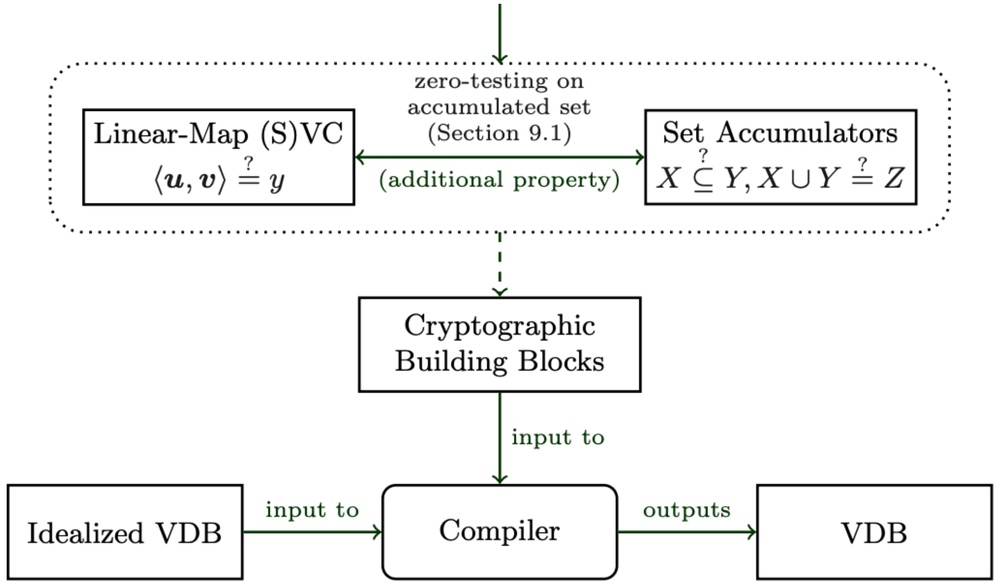
X accumulator

$\boldsymbol{v}$ linear-map vector commitment

Accumulators: $\mathsf{VfySubset}(\mathsf{acc}_X, \mathsf{acc}_Y, \pi_{\text{subset}})$ (checks $X \subseteq Y$)

Vector Commitments: $\mathsf{VfySubvec}(\mathsf{cm}_{\boldsymbol{v}}, X, \boldsymbol{y}, \pi_{\text{subvec}})$ (checks $\boldsymbol{v}_X = \boldsymbol{y}$)

+ linear-map opening: $\mathsf{VfyLin}(\mathsf{cm}_{\boldsymbol{u}}, \mathsf{cm}_{\boldsymbol{v}}, y, \pi_{\text{lin}})$ (checks $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = y$)



From Idealized to Cryptographic VDB

$$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \overset{?}{\subseteq} Y \qquad Z \overset{?}{=} X \cup Y \qquad Z \overset{?}{=} X \cap Y \quad \textit{(set ops)}$$

$$\boldsymbol{u} \overset{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)} \qquad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \overset{?}{=} y \quad \textit{(inner product)}$$

$$\boldsymbol{v} \left[X \right] \overset{?}{=} \mathbf{0} \quad \textit{(zero test)}$$

We commit to handles:

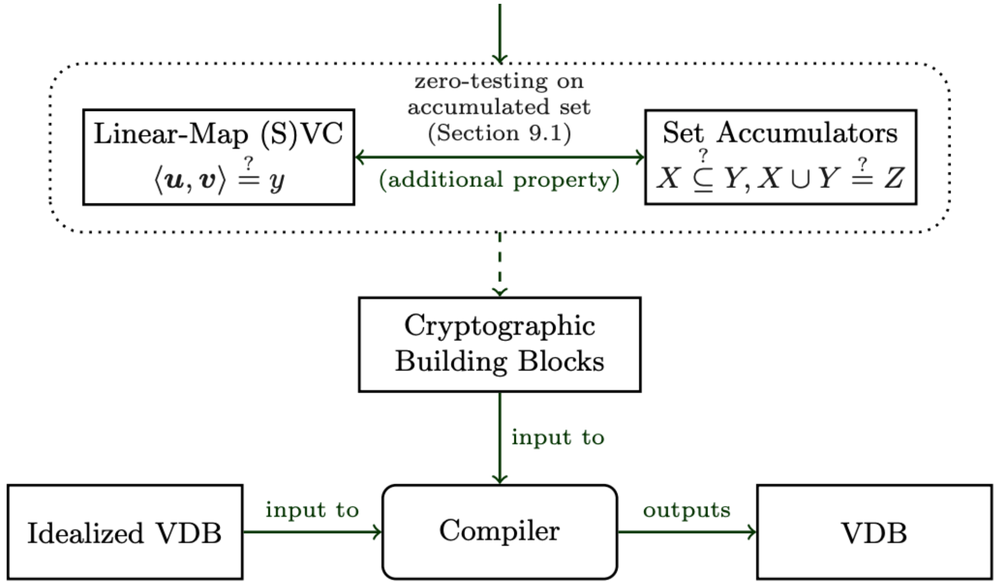
X accumulator

$\boldsymbol{v}$ linear-map vector commitment

Accumulators: $\mathsf{VfySubset}(\mathsf{acc}_X, \mathsf{acc}_Y, \pi_{\text{subset}})$ (checks $X \subseteq Y$)

Vector Commitments: $\mathsf{VfySubvec}(\mathsf{cm}_{\boldsymbol{v}}, X, \boldsymbol{y}, \pi_{\text{subvec}})$ (checks $\boldsymbol{v}_X = \boldsymbol{y}$)

+ linear-map opening: $\mathsf{VfyLin}(\mathsf{cm}_{\boldsymbol{u}}, \mathsf{cm}_{\boldsymbol{v}}, y, \pi_{\text{lin}})$ (checks $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = y$)



From Idealized to Cryptographic VDB

$$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \overset{?}{\subseteq} Y \qquad Z \overset{?}{=} X \cup Y \qquad Z \overset{?}{=} X \cap Y \quad \textit{(set ops)}$$

$$\boldsymbol{u} \overset{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)} \qquad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \overset{?}{=} y \quad \textit{(inner product)}$$

$$\boldsymbol{v} \left[X \right] \overset{?}{=} \mathbf{0} \quad \textit{(zero test)}$$

We commit to handles:

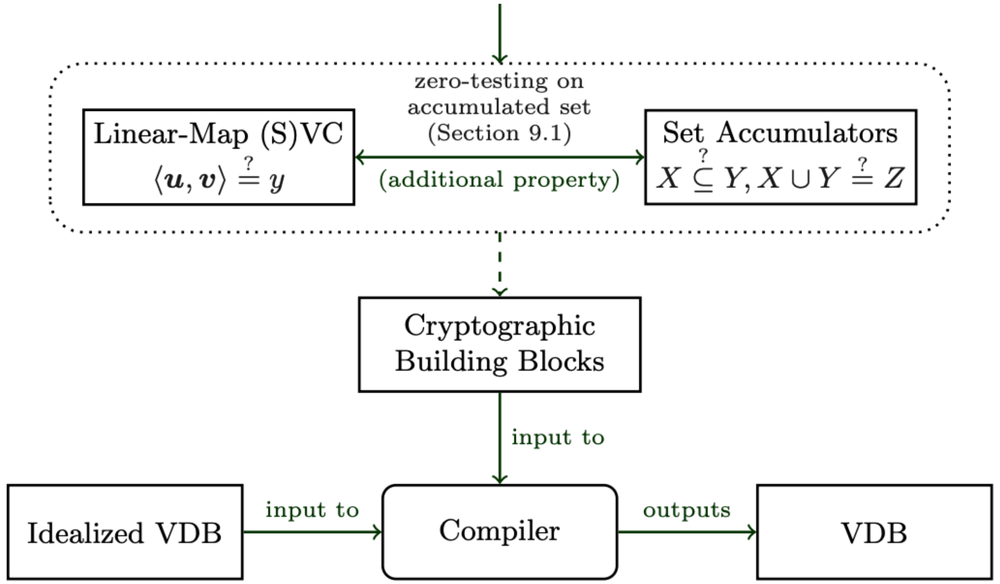
X accumulator

$\boldsymbol{v}$ linear-map vector commitment

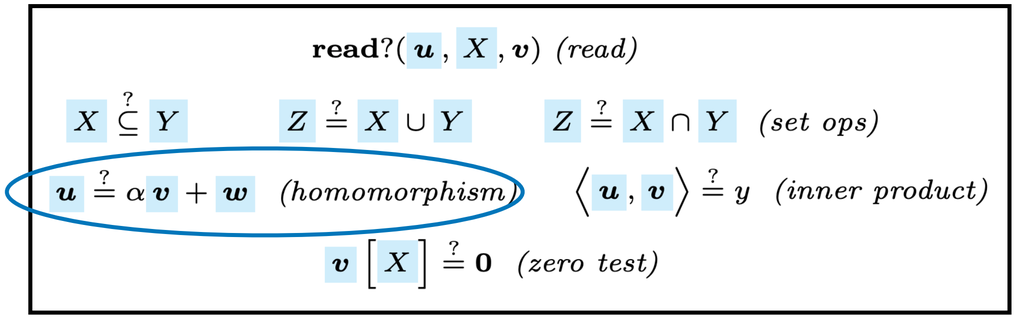
Accumulators: $\mathsf{VfySubset}(\mathsf{acc}_X, \mathsf{acc}_Y, \pi_{\text{subset}})$ (checks $X \subseteq Y$)

Vector Commitments: $\mathsf{VfySubvec}(\mathsf{cm}_{\boldsymbol{v}}, X, \boldsymbol{y}, \pi_{\text{subvec}})$ (checks $\boldsymbol{v}_X = \boldsymbol{y}$)

+ linear-map opening: $\mathsf{VfyLin}(\mathsf{cm}_{\boldsymbol{u}}, \mathsf{cm}_{\boldsymbol{v}}, y, \pi_{\text{lin}})$ (checks $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = y$)



From Idealized to Cryptographic VDB



We commit to handles:

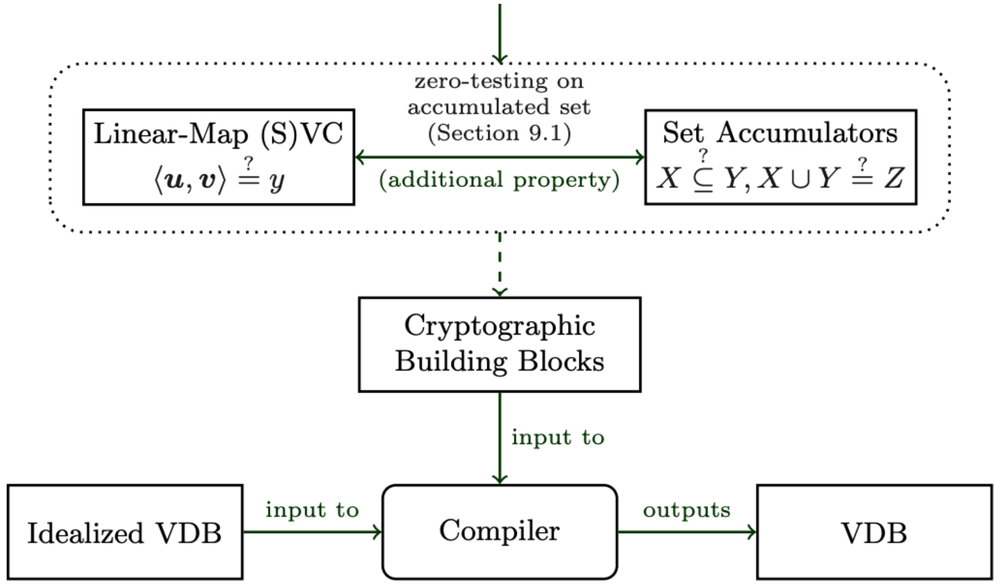
X accumulator

$\boldsymbol{v}$ linear-map vector commitment

Accumulators: $\mathsf{VfySubset}(\mathsf{acc}_X, \mathsf{acc}_Y, \pi_{\text{subset}})$ (checks $X \subseteq Y$)

Vector Commitments: $\mathsf{VfySubvec}(\mathsf{cm}_{\boldsymbol{v}}, X, \boldsymbol{y}, \pi_{\text{subvec}})$ (checks $\boldsymbol{v}_X = \boldsymbol{y}$)

+ linear-map opening: $\mathsf{VfyLin}(\mathsf{cm}_{\boldsymbol{u}}, \mathsf{cm}_{\boldsymbol{v}}, y, \pi_{\text{lin}})$ (checks $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = y$)



From Idealized to Cryptographic VDB

$$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \overset{?}{\subseteq} Y \qquad Z \overset{?}{=} X \cup Y \qquad Z \overset{?}{=} X \cap Y \quad \textit{(set ops)}$$

$$\boldsymbol{u} \overset{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)} \qquad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \overset{?}{=} y \quad \textit{(inner product)}$$

$$\boldsymbol{v} \left[X \right] \overset{?}{=} \mathbf{0} \quad \textit{(zero test)}$$

We commit to handles:

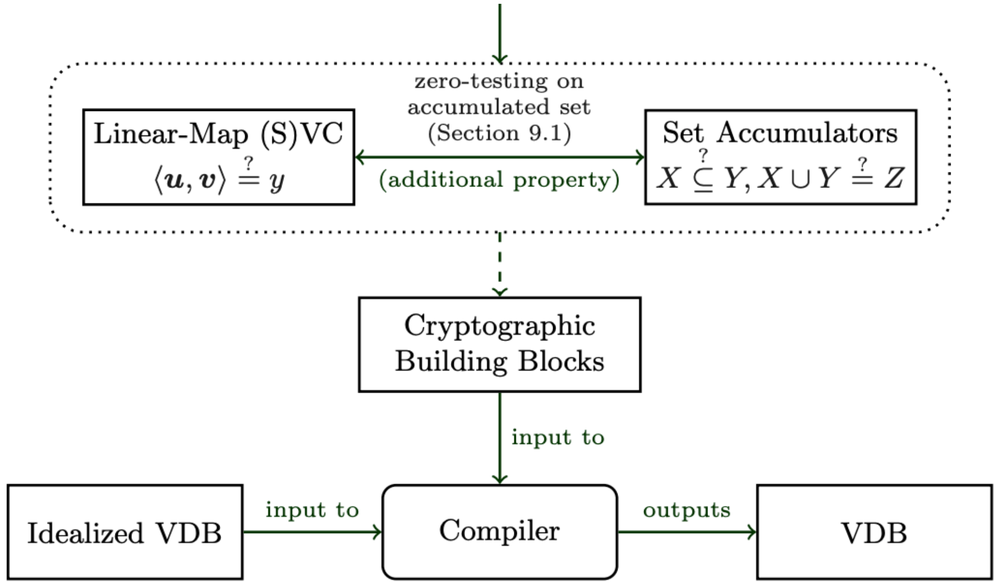
X accumulator

$\boldsymbol{v}$ linear-map vector commitment

Accumulators: $\mathsf{VfySubset}(\mathsf{acc}_X, \mathsf{acc}_Y, \pi_{\text{subset}})$ (checks $X \subseteq Y$)

Vector Commitments: $\mathsf{VfySubvec}(\mathsf{cm}_{\boldsymbol{v}}, X, \boldsymbol{y}, \pi_{\text{subvec}})$ (checks $\boldsymbol{v}_X = \boldsymbol{y}$)

+ linear-map opening: $\mathsf{VfyLin}(\mathsf{cm}_{\boldsymbol{u}}, \mathsf{cm}_{\boldsymbol{v}}, y, \pi_{\text{lin}})$ (checks $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = y$)



From Idealized to Cryptographic VDB

$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v})$ *(read)*

$X \stackrel{?}{\subseteq} Y$ $\quad Z \stackrel{?}{=} X \cup Y$ $\quad Z \stackrel{?}{=} X \cap Y$ *(set ops)*

$\boldsymbol{u} \stackrel{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w}$ *(homomorphism)* $\quad \langle \boldsymbol{u}, \boldsymbol{v} \rangle \stackrel{?}{=} y$ *(inner product)*

$\boldsymbol{v} \left[X \right] \stackrel{?}{=} \mathbf{0}$ *(zero test)*

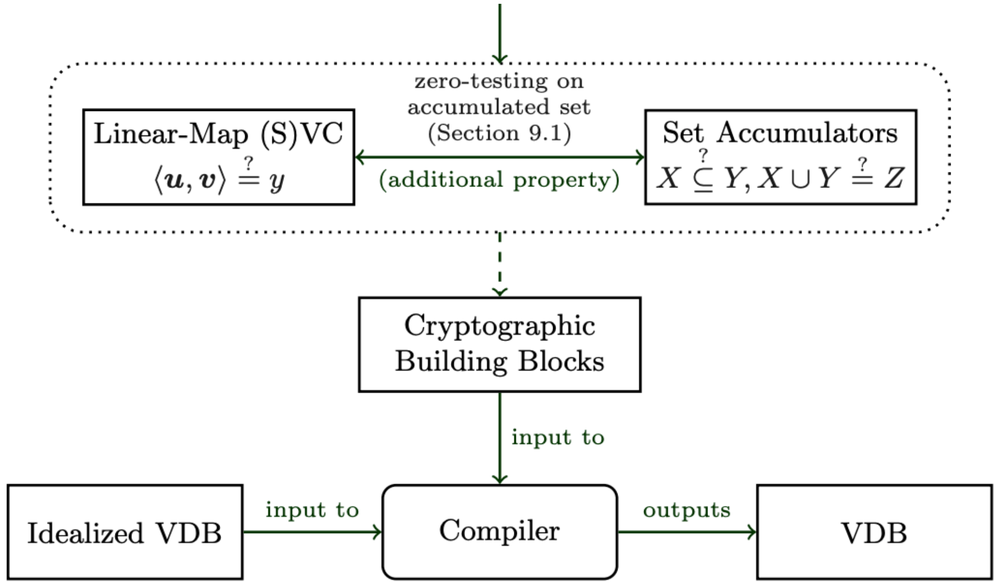
We commit to handles:

X accumulator $\quad \boldsymbol{v}$ linear-map vector commitment

Accumulators: $\mathsf{VfySubset}(\mathsf{acc}_X, \mathsf{acc}_Y, \pi_{\text{subset}})$ (checks $X \subseteq Y$)

Vector Commitments: $\mathsf{VfySubvec}(\mathsf{cm}_{\boldsymbol{v}}, X, \boldsymbol{y}, \pi_{\text{subvec}})$ (checks $\boldsymbol{v}_X = \boldsymbol{y}$)

+ linear-map opening: $\mathsf{VfyLin}(\mathsf{cm}_{\boldsymbol{u}}, \mathsf{cm}_{\boldsymbol{v}}, y, \pi_{\text{lin}})$ (checks $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = y$)



From Idealized to Cryptographic VDB

$$\mathbf{read}?(\boldsymbol{u}, X, \boldsymbol{v}) \quad \textit{(read)}$$

$$X \overset{?}{\subseteq} Y \qquad Z \overset{?}{=} X \cup Y \qquad Z \overset{?}{=} X \cap Y \quad \textit{(set ops)}$$

$$\boldsymbol{u} \overset{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)} \qquad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \overset{?}{=} y \quad \textit{(inner product)}$$

$$\boldsymbol{v} \left[X \right] \overset{?}{=} \mathbf{0} \quad \textit{(zero test)}$$

We commit to handles:

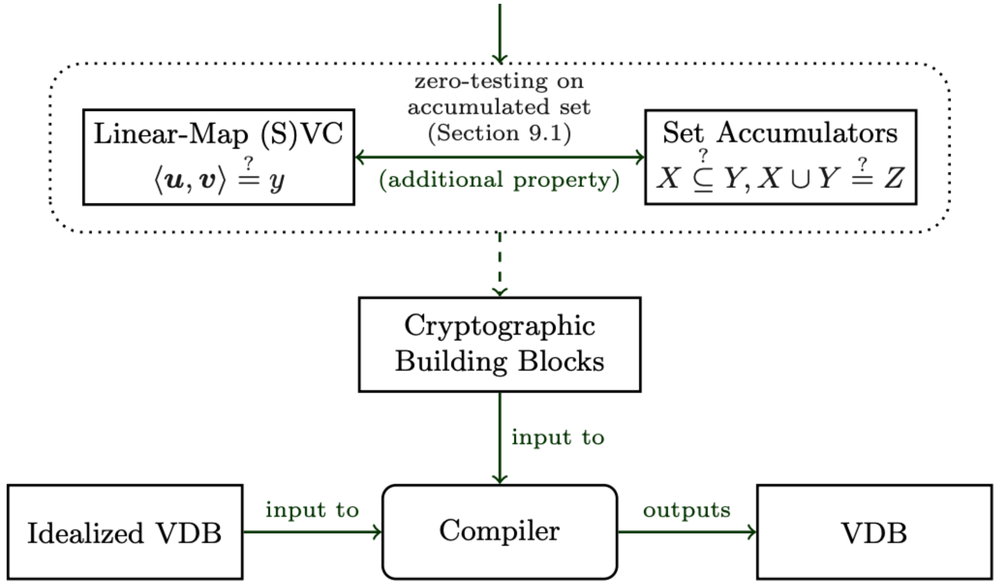
X accumulator

$\boldsymbol{v}$ linear-map vector commitment

Accumulators: $\mathsf{VfySubset}(\mathsf{acc}_X, \mathsf{acc}_Y, \pi_{\text{subset}})$ (checks $X \subseteq Y$)

Vector Commitments: $\mathsf{VfySubvec}(\mathsf{cm}_{\boldsymbol{v}}, X, \boldsymbol{y}, \pi_{\text{subvec}})$ (checks $\boldsymbol{v}_X = \boldsymbol{y}$)

+ linear-map opening: $\mathsf{VfyLin}(\mathsf{cm}_{\boldsymbol{u}}, \mathsf{cm}_{\boldsymbol{v}}, y, \pi_{\text{lin}})$ (checks $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = y$)



From Idealized to Cryptographic VDB

$\mathbf{read?}(\boldsymbol{u}, X, \boldsymbol{v})$ *(read)*

$X \overset{?}{\subseteq} Y$ $\quad Z \overset{?}{=} X \cup Y$ $\quad Z \overset{?}{=} X \cap Y$ *(set ops)*

$\boldsymbol{u} \overset{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w}$ *(homomorphism)* $\quad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \overset{?}{=} y$ *(inner product)*

$\boldsymbol{v} \left[X \right] \overset{?}{=} \mathbf{0}$ *(zero test)*

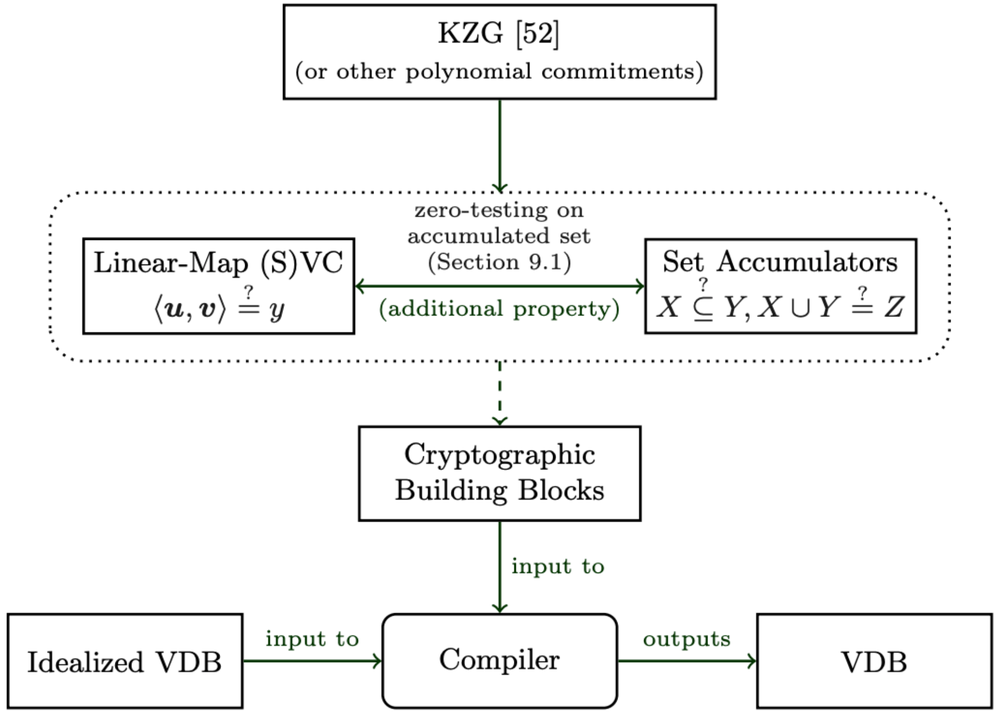
We commit to handles:

X accumulator $\quad \boldsymbol{v}$ linear-map vector commitment

Accumulators: $\mathsf{VfySubset}(\mathsf{acc}_X, \mathsf{acc}_Y, \pi_{\text{subset}})$ (checks $X \subseteq Y$)

Vector Commitments: $\mathsf{VfySubvec}(\mathsf{cm}_{\boldsymbol{v}}, X, \boldsymbol{y}, \pi_{\text{subvec}})$ (checks $\boldsymbol{v}_X = \boldsymbol{y}$)

+ linear-map opening: $\mathsf{VfyLin}(\mathsf{cm}_{\boldsymbol{u}}, \mathsf{cm}_{\boldsymbol{v}}, y, \pi_{\text{lin}})$ (checks $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = y$)



From Idealized to Cryptographic VDB

$\mathbf{read?}(\boldsymbol{u}, X, \boldsymbol{v})$ *(read)*

$X \overset{?}{\subseteq} Y$ $\quad Z \overset{?}{=} X \cup Y$ $\quad Z \overset{?}{=} X \cap Y$ *(set ops)*

$\boldsymbol{u} \overset{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w}$ *(homomorphism)* $\quad \left\langle \boldsymbol{u}, \boldsymbol{v} \right\rangle \overset{?}{=} y$ *(inner product)*

$\boldsymbol{v} \left[X \right] \overset{?}{=} \mathbf{0}$ *(zero test)*

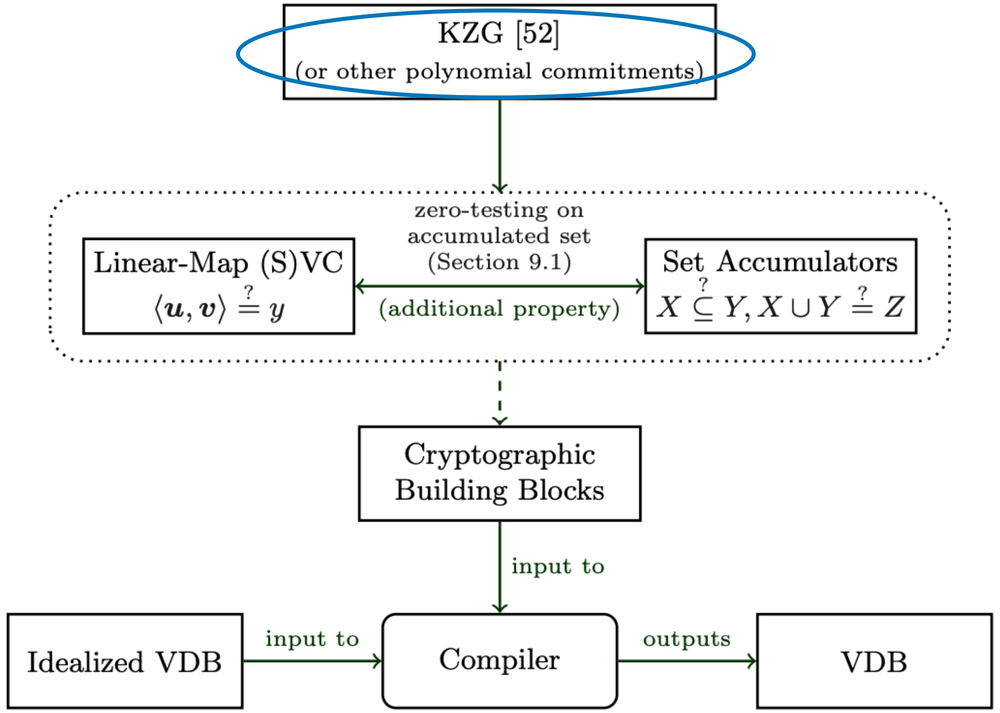
We commit to handles:

X accumulator $\quad \boldsymbol{v}$ linear-map vector commitment

Accumulators: $\mathsf{VfySubset}(\mathsf{acc}_X, \mathsf{acc}_Y, \pi_{\text{subset}})$ (checks $X \subseteq Y$)

Vector Commitments: $\mathsf{VfySubvec}(\mathsf{cm}_{\boldsymbol{v}}, X, \boldsymbol{y}, \pi_{\text{subvec}})$ (checks $\boldsymbol{v}_X = \boldsymbol{y}$)

+ linear-map opening: $\mathsf{VfyLin}(\mathsf{cm}_{\boldsymbol{u}}, \mathsf{cm}_{\boldsymbol{v}}, y, \pi_{\text{lin}})$ (checks $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = y$)



The final construction: qedb

- qedb is simply a specific idealized VDB compiled “through KZG” (with the approach from last slide)

The final construction: qedb

- qedb is simply a specific idealized VDB compiled “through KZG” (with the approach from last slide)

– MIN query: consider the query:

$$\text{Q6}: \quad \text{SELECT MIN}(\mathsf{col}_{tgt}) \text{ FROM } T \tag{7}$$

Pre-processing: we assume that the Verifier has the slice handle $\llbracket tgt \rrbracket$ corresponding to col_{tgt}.

Proof computation: the Prover does as follows:

- compute the set of positions $argmin(tgt)$ in col_{tgt} of the minimum values: $argmin(tgt) = \{j : \mathsf{col}_{tgt}[j] \leq v \quad \text{for all } v \in \mathsf{col}_{tgt}\}$ (here we denote with $\mathsf{col}_{tgt}[j]$ the j-th element in the slice referred by $\llbracket tgt \rrbracket$.
- compute $\bigcirc X_{argmin}$ and sends it to the Verifier

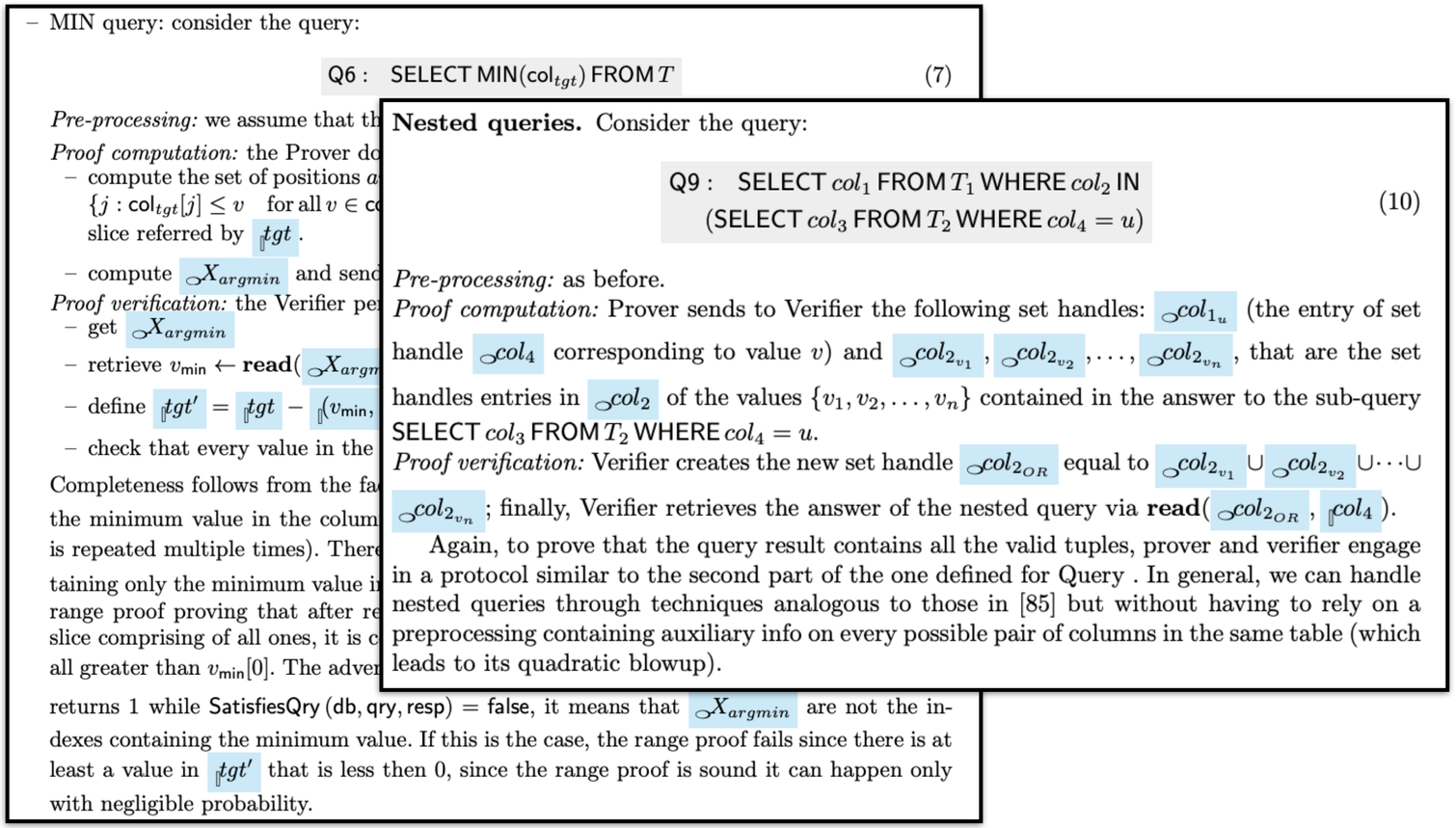
Proof verification: the Verifier performs the following steps:

- get $\bigcirc X_{argmin}$
- retrieve $v_{\mathsf{min}} \leftarrow \mathbf{read}(\bigcirc X_{argmin}, \llbracket tgt \rrbracket)$
- define $\llbracket tgt' \rrbracket = \llbracket tgt \rrbracket - \llbracket (v_{\mathsf{min}}, \ldots, v_{min}) \rrbracket$
- check that every value in the slice handle $\llbracket tgt' \rrbracket$ lies in the interval $[0, 2^l)$

Completeness follows from the fact that $\bigcirc X_{argmin}$ actually contains only the indices with the minimum value in the column (it can contain more than one element if the minimum is repeated multiple times). Therefore $v_{\mathsf{min}} \leftarrow \mathbf{read}(\bigcirc X_{argmin}, \llbracket tgt \rrbracket)$ outputs a vector containing only the minimum value in the column col_{tgt}. The correctness is also enforced by the range proof proving that after removing from col_{tgt} the vector $v_{\mathsf{min}}[0]\boldsymbol{u}_{\mathbb{1}}$, where $\boldsymbol{u}_{\mathbb{1}}$ is the slice comprising of all ones, it is contained in the range $[0, 2^\ell)$ meaning that these values are all greater than $v_{\mathsf{min}}[0]$. The adversarial prover sends $\bigcirc X_{argmin}$, therefore if the verification returns 1 while $\mathsf{SatisfiesQry}(\mathsf{db}, \mathsf{qry}, \mathsf{resp}) = \mathsf{false}$, it means that $\bigcirc X_{argmin}$ are not the indexes containing the minimum value. If this is the case, the range proof fails since there is at least a value in $\llbracket tgt' \rrbracket$ that is less then 0, since the range proof is sound it can happen only with negligible probability.

The final construction: qedb

- qedb is simply a specific idealized VDB compiled “through KZG” (with the approach from last slide)



– MIN query: consider the query:

$$\text{Q6}: \quad \text{SELECT MIN}(\text{col}_{tgt}) \text{ FROM } T \tag{7}$$

Pre-processing: we assume that th

Proof computation: the Prover do
– compute the set of positions a
$\{j : \text{col}_{tgt}[j] \leq v \quad \text{for all } v \in c$
slice referred by $[\![tgt$.
– compute $\subset X_{argmin}$ and send

Proof verification: the Verifier pe
– get $\subset X_{argmin}$
– retrieve $v_{\text{min}} \leftarrow \textbf{read}(\subset X_{argm}$
– define $[\![tgt' = [\![tgt - [\!(v_{\text{min}},$
– check that every value in the

Completeness follows from the fa
the minimum value in the colum
is repeated multiple times). There
taining only the minimum value i
range proof proving that after re
slice comprising of all ones, it is c
all greater than $v_{\text{min}}[0]$. The adver

returns 1 while $\text{SatisfiesQry}(\text{db}, \text{qry}, \text{resp}) = \text{false}$, it means that $\subset X_{argmin}$ are not the indexes containing the minimum value. If this is the case, the range proof fails since there is at least a value in $[\![tgt'$ that is less then 0, since the range proof is sound it can happen only with negligible probability.

Nested queries. Consider the query:

$$\text{Q9}: \quad \text{SELECT } col_1 \text{ FROM } T_1 \text{ WHERE } col_2 \text{ IN } (\text{SELECT } col_3 \text{ FROM } T_2 \text{ WHERE } col_4 = u) \tag{10}$$

Pre-processing: as before.

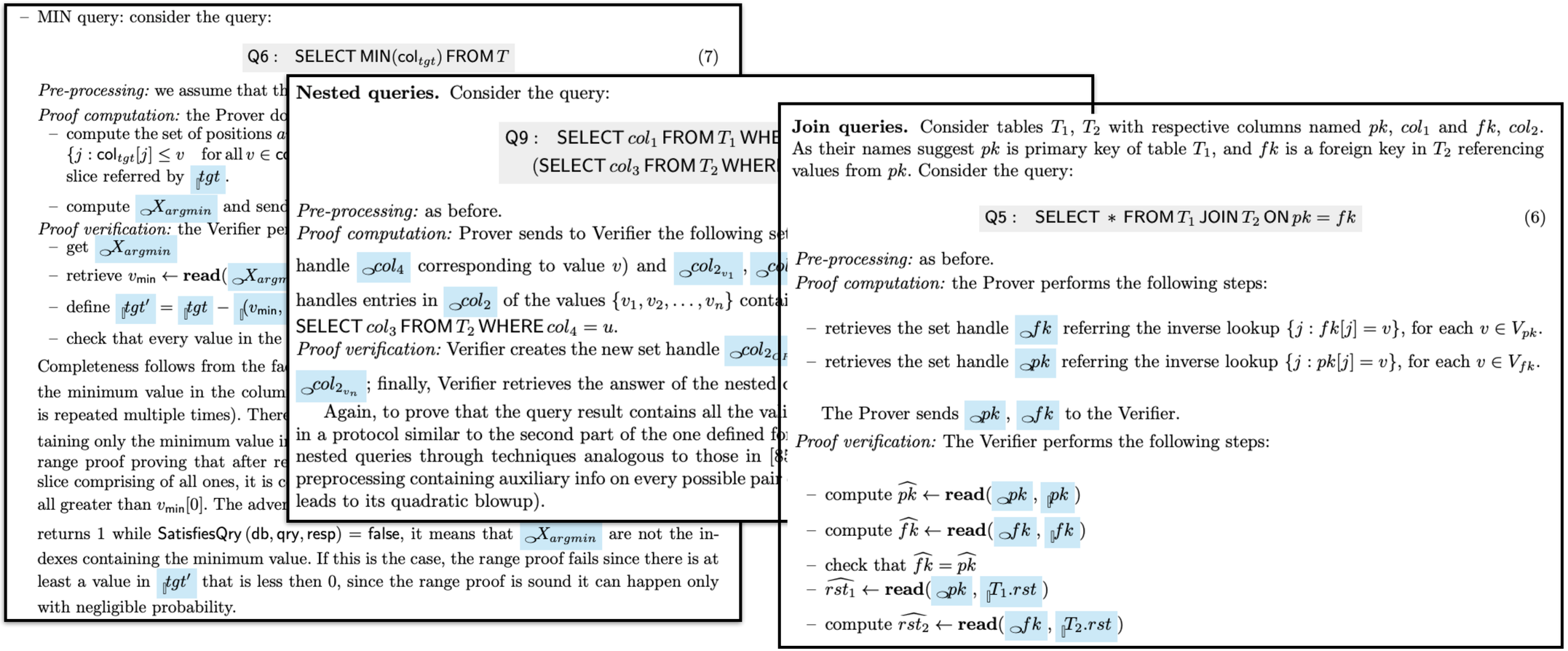
Proof computation: Prover sends to Verifier the following set handles: $\subset col_{1_u}$ (the entry of set handle $\subset col_4$ corresponding to value v) and $\subset col_{2_{v_1}}, \subset col_{2_{v_2}}, \ldots, \subset col_{2_{v_n}}$, that are the set handles entries in $\subset col_2$ of the values $\{v_1, v_2, \ldots, v_n\}$ contained in the answer to the sub-query $\text{SELECT } col_3 \text{ FROM } T_2 \text{ WHERE } col_4 = u$.

Proof verification: Verifier creates the new set handle $\subset col_{2_{OR}}$ equal to $\subset col_{2_{v_1}} \cup \subset col_{2_{v_2}} \cup \cdots \cup \subset col_{2_{v_n}}$; finally, Verifier retrieves the answer of the nested query via $\textbf{read}(\subset col_{2_{OR}}, [\![col_4)$.

Again, to prove that the query result contains all the valid tuples, prover and verifier engage in a protocol similar to the second part of the one defined for Query . In general, we can handle nested queries through techniques analogous to those in [85] but without having to rely on a preprocessing containing auxiliary info on every possible pair of columns in the same table (which leads to its quadratic blowup).

The final construction: qedb

- qedb is simply a specific idealized VDB compiled “through KZG” (with the approach from last slide)



– MIN query: consider the query:

$$\text{Q6}: \quad \text{SELECT MIN}(\text{col}_{tgt}) \text{ FROM } T \tag{7}$$

Pre-processing: we assume that th

Proof computation: the Prover do
– compute the set of positions a
$\{j : \text{col}_{tgt}[j] \leq v \quad \text{for all } v \in c$
slice referred by $\llbracket tgt \rrbracket$.
– compute $\bigcirc X_{argmin}$ and send

Proof verification: the Verifier pe
– get $\bigcirc X_{argmin}$
– retrieve $v_{\min} \leftarrow \textbf{read}(\bigcirc X_{argm}$
– define $\llbracket tgt' \rrbracket = \llbracket tgt \rrbracket - \llbracket (v_{\min},$
– check that every value in the

Completeness follows from the fa
the minimum value in the colum
is repeated multiple times). There
taining only the minimum value i
range proof proving that after re
slice comprising of all ones, it is c
all greater than $v_{\min}[0]$. The adver
returns 1 while $\mathsf{SatisfiesQry}(\mathsf{db}, \mathsf{qry}, \mathsf{resp}) = \mathsf{false}$, it means that $\bigcirc X_{argmin}$ are not the indexes containing the minimum value. If this is the case, the range proof fails since there is at least a value in $\llbracket tgt' \rrbracket$ that is less then 0, since the range proof is sound it can happen only with negligible probability.

Nested queries. Consider the query:

$$\text{Q9}: \quad \text{SELECT } col_1 \text{ FROM } T_1 \text{ WHE}$$
$$(\text{SELECT } col_3 \text{ FROM } T_2 \text{ WHER}$$

Pre-processing: as before.
Proof computation: Prover sends to Verifier the following se
handle $\bigcirc col_4$ corresponding to value v) and $\bigcirc col_{2_{v_1}}$, $\bigcirc co$
handles entries in $\bigcirc col_2$ of the values $\{v_1, v_2, \ldots, v_n\}$ contai
$\text{SELECT } col_3 \text{ FROM } T_2 \text{ WHERE } col_4 = u$.
Proof verification: Verifier creates the new set handle $\bigcirc col_{2_{v_n}}$; finally, Verifier retrieves the answer of the nested

Again, to prove that the query result contains all the va
in a protocol similar to the second part of the one defined fo
nested queries through techniques analogous to those in [8
preprocessing containing auxiliary info on every possible pair
leads to its quadratic blowup).

Join queries. Consider tables T_1, T_2 with respective columns named pk, col_1 and fk, col_2. As their names suggest pk is primary key of table T_1, and fk is a foreign key in T_2 referencing values from pk. Consider the query:

$$\text{Q5}: \quad \text{SELECT } * \text{ FROM } T_1 \text{ JOIN } T_2 \text{ ON } pk = fk \tag{6}$$

Pre-processing: as before.
Proof computation: the Prover performs the following steps:

– retrieves the set handle $\bigcirc fk$ referring the inverse lookup $\{j : fk[j] = v\}$, for each $v \in V_{pk}$.
– retrieves the set handle $\bigcirc pk$ referring the inverse lookup $\{j : pk[j] = v\}$, for each $v \in V_{fk}$.

The Prover sends $\bigcirc pk$, $\bigcirc fk$ to the Verifier.

Proof verification: The Verifier performs the following steps:

– compute $\widehat{pk} \leftarrow \textbf{read}(\bigcirc pk, \llbracket pk \rrbracket)$
– compute $\widehat{fk} \leftarrow \textbf{read}(\bigcirc fk, \llbracket fk \rrbracket)$
– check that $\widehat{fk} = \widehat{pk}$
– $\widehat{rst_1} \leftarrow \textbf{read}(\bigcirc pk, \llbracket T_1.rst \rrbracket)$
– compute $\widehat{rst_2} \leftarrow \textbf{read}(\bigcirc fk, \llbracket T_2.rst \rrbracket)$

Wrapping Up

Wrapping Up

- **Simplicity** is important both for real-world security and for research progress

Wrapping Up

- **Simplicity** is important both for real-world security and for research progress
  - Complicated SNARK-based stacks may often be an overkill and not worth the additional risks

Wrapping Up

- **Simplicity** is important both for real-world security and for research progress
  - Complicated SNARK-based stacks may often be an overkill and not worth the additional risks
- **Research on VDBs** from authenticated data structures has been **stagnant** for almost ten years

Wrapping Up

- **Simplicity** is important both for real-world security and for research progress
  - Complicated SNARK-based stacks may often be an overkill and not worth the additional risks
- **Research on VDBs** from authenticated data structures has been **stagnant** for almost ten years
- **qedb is** a new VDB aiming at being:

Wrapping Up

- **Simplicity** is important both for real-world security and for research progress
  - Complicated SNARK-based stacks may often be an overkill and not worth the additional risks
- **Research on VDBs** from authenticated data structures has been **stagnant** for almost ten years
- **qedb is** a new VDB aiming at being:
  - **performant**

Wrapping Up

- **Simplicity** is important both for real-world security and for research progress
  - Complicated SNARK-based stacks may often be an overkill and not worth the additional risks
- **Research on VDBs** from authenticated data structures has been **stagnant** for almost ten years
- **qedb is** a new VDB aiming at being:
  - **performant**
    - (can scale to million of rows; proof size independent of |DB|; verifier might even be directly on chain)

Wrapping Up

- **Simplicity** is important both for real-world security and for research progress
  - Complicated SNARK-based stacks may often be an overkill and not worth the additional risks
- **Research on VDBs** from authenticated data structures has been **stagnant** for almost ten years
- **qedb is** a new VDB aiming at being:
  - **performant**
    - (can scale to million of rows; proof size independent of |DB|; verifier might even be directly on chain)
  - **astonishingly simple**

Wrapping Up

- **Simplicity** is important both for real-world security and for research progress
  - Complicated SNARK-based stacks may often be an overkill and not worth the additional risks
- **Research on VDBs** from authenticated data structures has been **stagnant** for almost ten years
- **qedb is** a new VDB aiming at being:
  - **performant**
    - (can scale to million of rows; proof size independent of |DB|; verifier might even be directly on chain)
  - **astonishingly simple**
    - (an unsupervised LLM could implement it from its idealized description)

Wrapping Up

- **Simplicity** is important both for real-world security and for research progress
  - Complicated SNARK-based stacks may often be an overkill and not worth the additional risks
- **Research on VDBs** from authenticated data structures has been **stagnant** for almost ten years
- **qedb is** a new VDB aiming at being:
  - **performant**
    - (can scale to million of rows; proof size independent of |DB|; verifier might even be directly on chain)
  - **astonishingly simple**
    - (an unsupervised LLM could implement it from its idealized description)
- Framework behind qedb's design is plausibly of independent interest

Wrapping Up

- **Simplicity** is important both for real-world security and for research progress
  - Complicated SNARK-based stacks may often be an overkill and not worth the additional risks
- **Research on VDBs** from authenticated data structures has been **stagnant** for almost ten years
- **qedb is** a new VDB aiming at being:
  - **performant**
    - (can scale to million of rows; proof size independent of |DB|; verifier might even be directly on chain)
  - **astonishingly simple**
    - (an unsupervised LLM could implement it from its idealized description)
- Framework behind qedb's design is plausibly of independent interest
- **Future work:**

Wrapping Up

- **Simplicity** is important both for real-world security and for research progress
  - Complicated SNARK-based stacks may often be an overkill and not worth the additional risks
- **Research on VDBs** from authenticated data structures has been **stagnant** for almost ten years
- **qedb is** a new VDB aiming at being:
  - **performant**
    - (can scale to million of rows; proof size independent of |DB|; verifier might even be directly on chain)
  - **astonishingly simple**
    - (an unsupervised LLM could implement it from its idealized description)
- Framework behind qedb's design is plausibly of independent interest
- **Future work:**
  - Beyond SQL? (Key-Value, etc)

Wrapping Up

- **Simplicity** is important both for real-world security and for research progress
  - Complicated SNARK-based stacks may often be an overkill and not worth the additional risks
- **Research on VDBs** from authenticated data structures has been **stagnant** for almost ten years
- **qedb is** a new VDB aiming at being:
  - **performant**
    - (can scale to million of rows; proof size independent of |DB|; verifier might even be directly on chain)
  - **astonishingly simple**
    - (an unsupervised LLM could implement it from its idealized description)
- Framework behind qedb's design is plausibly of independent interest
- **Future work:**
  - Beyond SQL? (Key-Value, etc)
  - Zero-knowledge (hiding)

Wrapping Up

- **Simplicity** is important both for real-world security and for research progress
  - Complicated SNARK-based stacks may often be an overkill and not worth the additional risks
- **Research on VDBs** from authenticated data structures has been **stagnant** for almost ten years
- **qedb is** a new VDB aiming at being:
  - **performant**
    - (can scale to million of rows; proof size independent of |DB|; verifier might even be directly on chain)
  - **astonishingly simple**
    - (an unsupervised LLM could implement it from its idealized description)
- Framework behind qedb's design is plausibly of independent interest
- **Future work:**
  - Beyond SQL? (Key-Value, etc)
  - Zero-knowledge (hiding)
  - A lookup-singularity for VDBs?

Wrapping Up

- **Simplicity** is important both for real-world security and for research progress
  - Complicated SNARK-based stacks may often be an overkill and not worth the additional risks
- **Research on VDBs** from authenticated data structures has been **stagnant** for almost ten years
- **qedb is** a new VDB aiming at being:
  - **performant**
    - (can scale to million of rows; proof size independent of |DB|; verifier might even be directly on chain)
  - **astonishingly simple**
    - (an unsupervised LLM could implement it from its idealized description)
- Framework behind qedb's design is plausibly of independent interest
- **Future work:**
  - Beyond SQL? (Key-Value, etc)
  - Zero-knowledge (hiding)
  - A lookup-singularity for VDBs?
  - Formally verified implementation?

Wrapping Up

- **Simplicity** is important both for real-world security and for research progress
  - Complicated SNARK-based stacks may often be an overkill and not worth the additional risks
- **Research on VDBs** from authenticated data structures has been **stagnant** for almost ten years
- **qedb is** a new VDB aiming at being:
  - **performant**
    - (can scale to million of rows; proof size independent of |DB|; verifier might even be directly on chain)
  - **astonishingly simple**
    - (an unsupervised LLM could implement it from its idealized description)
- Framework behind qedb's design is plausibly of independent interest
- **Future work:**
  - Beyond SQL? (Key-Value, etc)
  - Zero-knowledge (hiding)
  - A lookup-singularity for VDBs?
  - Formally verified implementation?

Thanks!

Wrapping Up

- **Simplicity** is important both for real-world security and for research progress
  - Complicated SNARK-based stacks may often be an overkill and not worth the additional risks
- **Research on VDBs** from authenticated data structures has been **stagnant** for almost ten years
- **qedb is** a new VDB aiming at being:
  - **performant**
    - (can scale to million of rows; proof size independent of |DB|; verifier might even be directly on chain)
  - **astonishingly simple**
    - (an unsupervised LLM could implement it from its idealized description)
- Framework behind qedb's design is plausibly of independent interest
- **Future work:**
  - Beyond SQL? (Key-Value, etc)
  - Zero-knowledge (hiding)
  - A lookup-singularity for VDBs?
  - Formally verified implementation?

Thanks!

Questions?