Verifiable Elections That Scale for Free

Melissa Chase (MSR Redmond)
Markulf Kohlweiss (MSR Cambridge)
Anna Lysyanskaya (Brown University)
Sarah Meiklejohn (UC San Diego)

10,000-foot view of cryptographic voting

10,000-foot view of cryptographic voting

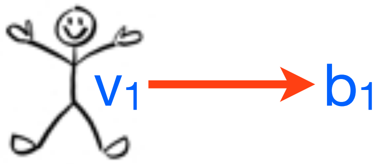
Phase 1: users encrypt votes to cast ballots

10,000-foot view of cryptographic voting

Phase 1: users encrypt votes to cast ballots

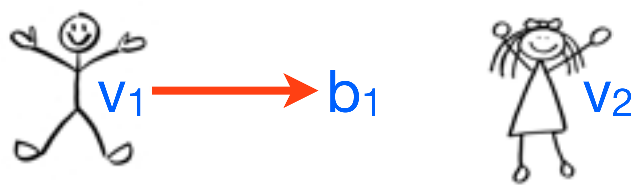
10,000-foot view of cryptographic voting

Phase 1: users encrypt votes to cast ballots



10,000-foot view of cryptographic voting

Phase 1: users encrypt votes to cast ballots

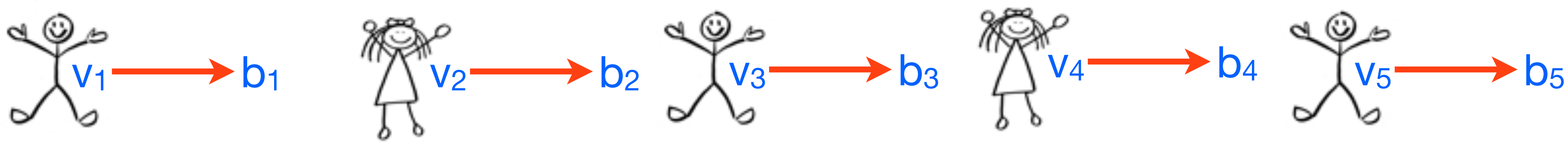


10,000-foot view of cryptographic voting

Phase 1: users encrypt votes to cast ballots

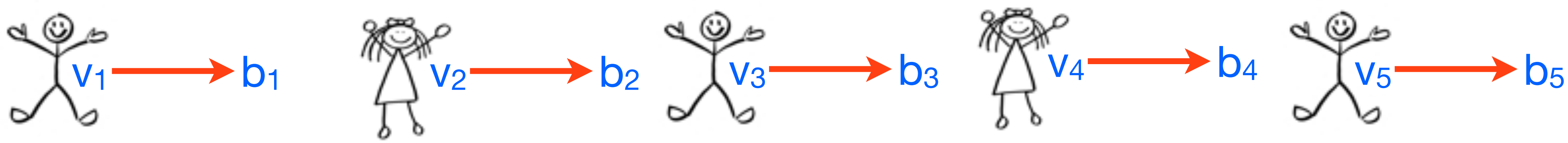
10,000-foot view of cryptographic voting

Phase 1: users encrypt votes to cast ballots



10,000-foot view of cryptographic voting

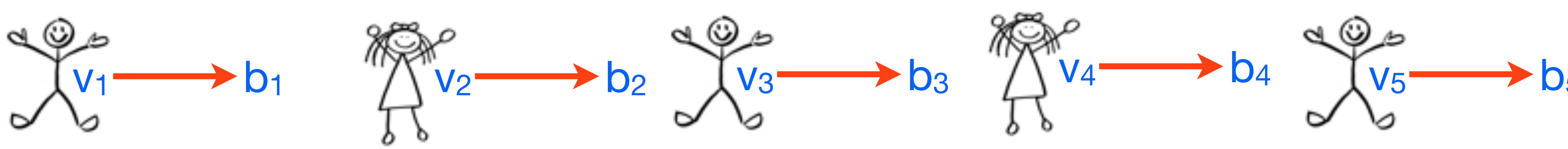
Phase 1: users encrypt votes to cast ballots



Phase 2: shuffle (permute and re-randomize) the ballots

10,000-foot view of cryptographic voting

Phase 1: users encrypt votes to cast ballots

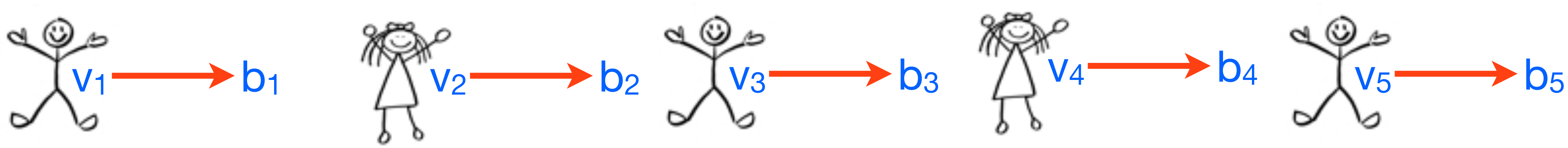


Phase 2: shuffle (permute and re-randomize) the ballots

b_1
b_2
b_3
b_4
b_5

10,000-foot view of cryptographic voting

Phase 1: users encrypt votes to cast ballots

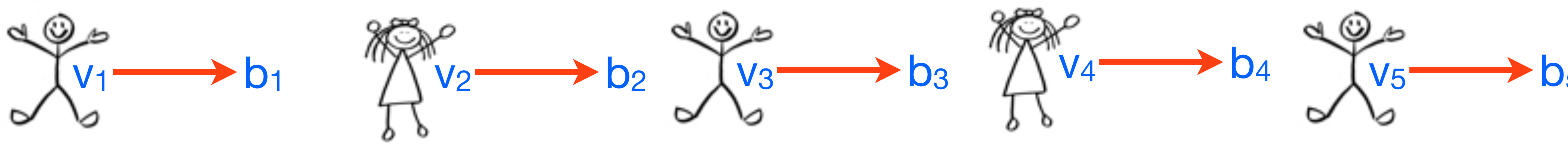


Phase 2: shuffle (permute and re-randomize) the ballots

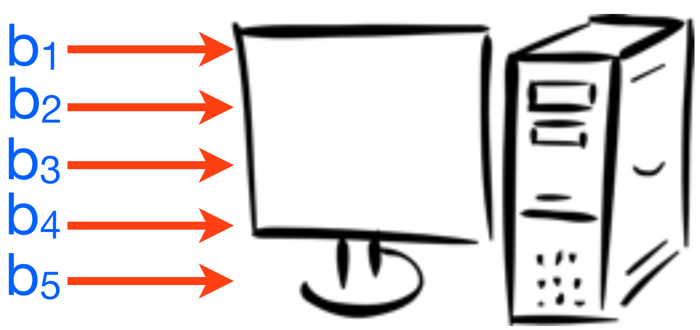
b_1
b_2
b_3
b_4
b_5

10,000-foot view of cryptographic voting

Phase 1: users encrypt votes to cast ballots

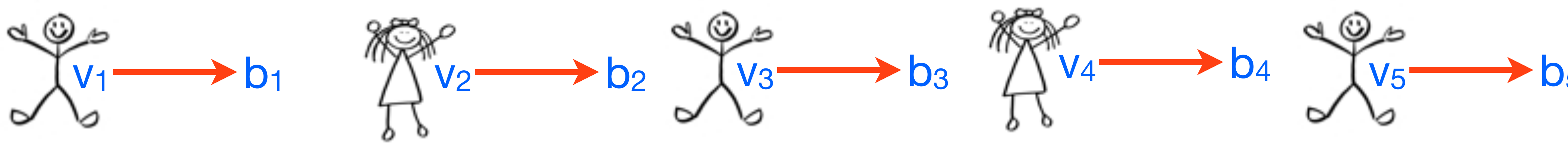


Phase 2: shuffle (permute and re-randomize) the ballots



10,000-foot view of cryptographic voting

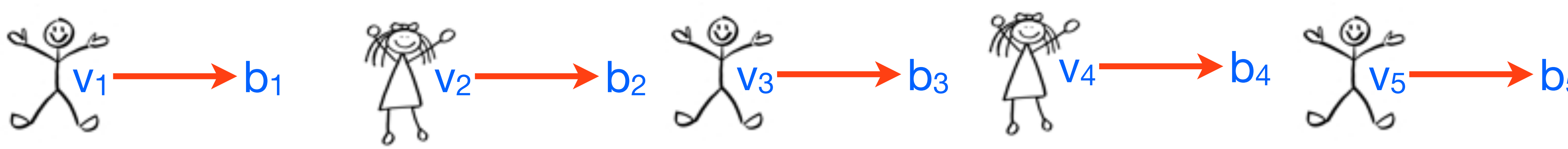
Phase 1: users encrypt votes to cast ballots



Phase 2: shuffle (permute and re-randomize) the ballots

10,000-foot view of cryptographic voting

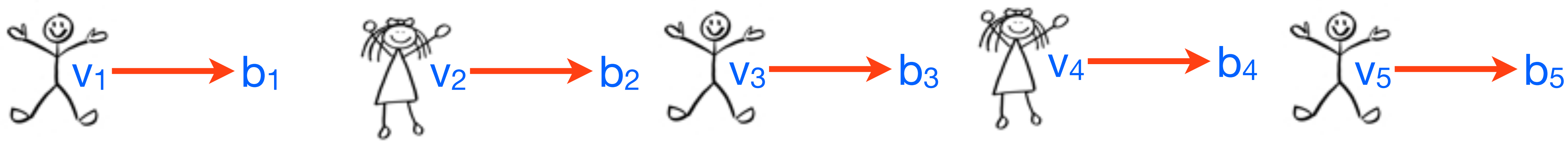
Phase 1: users encrypt votes to cast ballots



Phase 2: shuffle (permute and re-randomize) the ballots

10,000-foot view of cryptographic voting

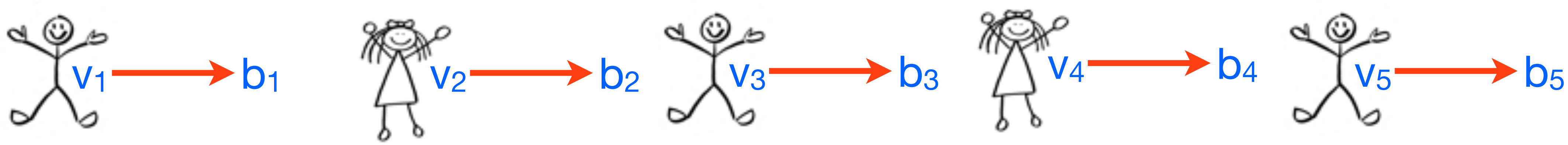
Phase 1: users encrypt votes to cast ballots



Phase 2: shuffle (permute and re-randomize) the ballots

10,000-foot view of cryptographic voting

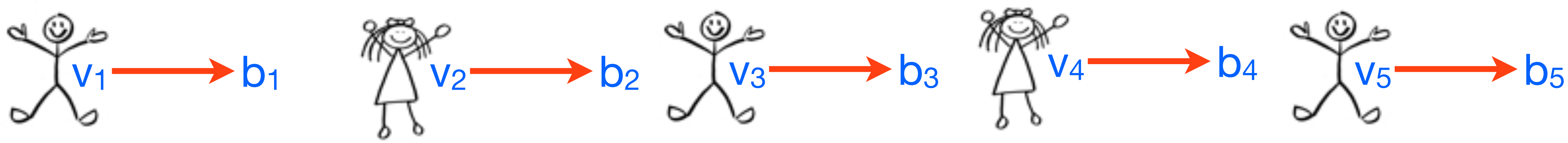
Phase 1: users encrypt votes to cast ballots



Phase 2: shuffle (permute and re-randomize) the ballots

10,000-foot view of cryptographic voting

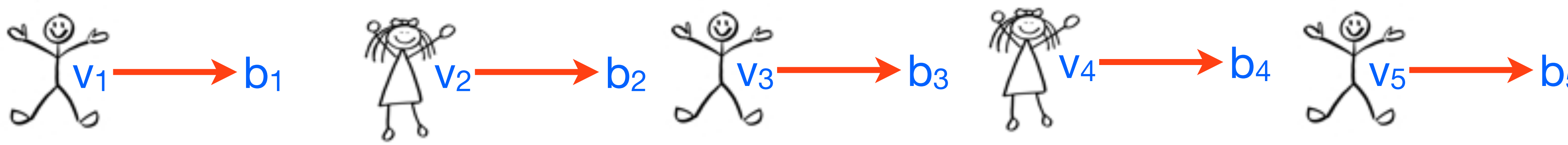
Phase 1: users encrypt votes to cast ballots



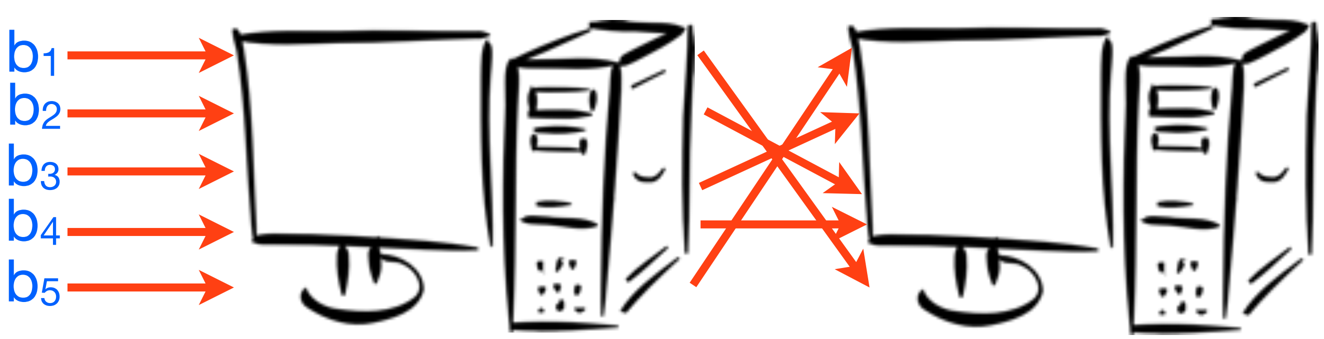
Phase 2: shuffle (permute and re-randomize) the ballots

10,000-foot view of cryptographic voting

Phase 1: users encrypt votes to cast ballots

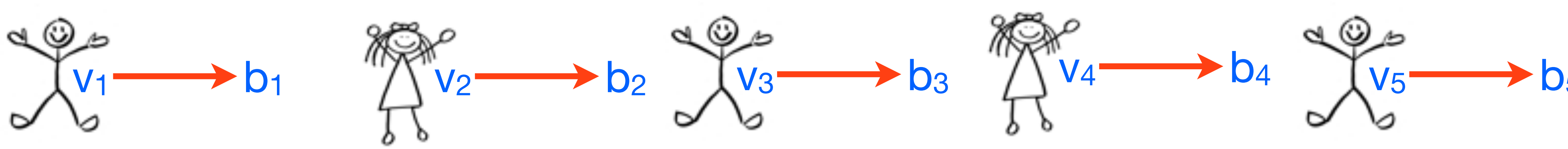


Phase 2: shuffle (permute and re-randomize) the ballots

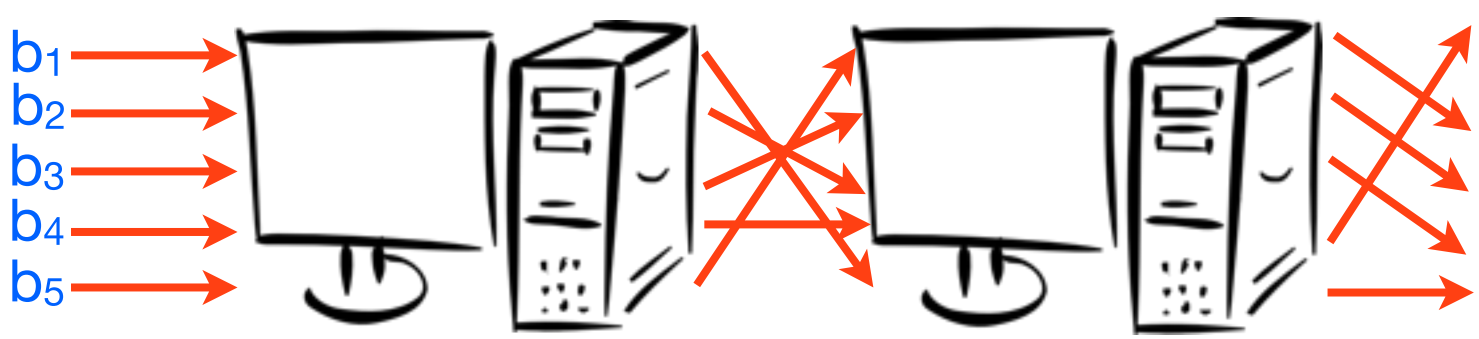


10,000-foot view of cryptographic voting

Phase 1: users encrypt votes to cast ballots

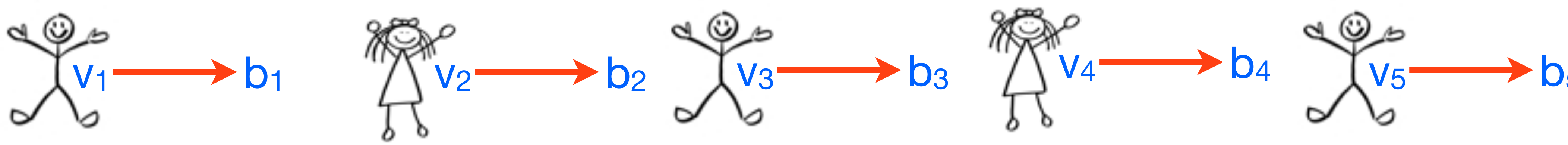


Phase 2: shuffle (permute and re-randomize) the ballots

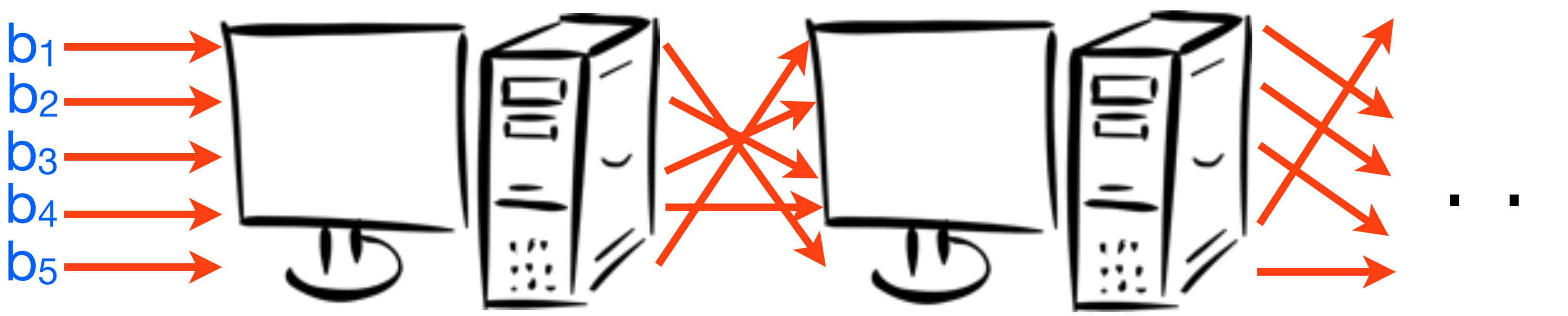


10,000-foot view of cryptographic voting

Phase 1: users encrypt votes to cast ballots

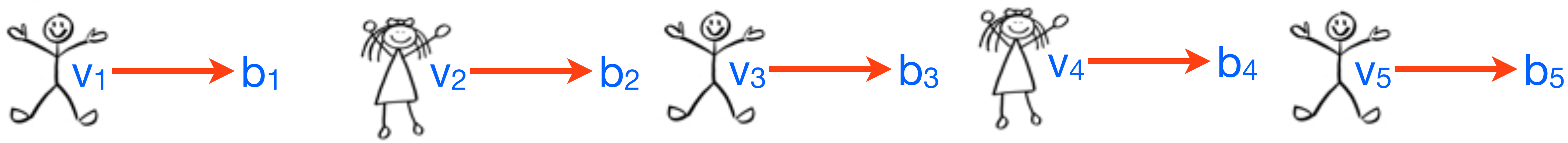


Phase 2: shuffle (permute and re-randomize) the ballots



10,000-foot view of cryptographic voting

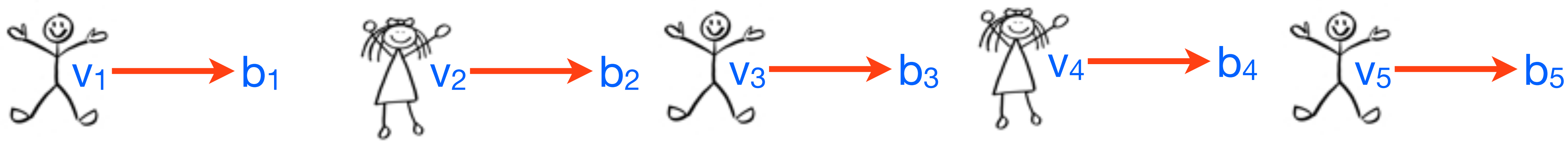
Phase 1: users encrypt votes to cast ballots



Phase 2: shuffle (permute and re-randomize) the ballots

10,000-foot view of cryptographic voting

Phase 1: users encrypt votes to cast ballots

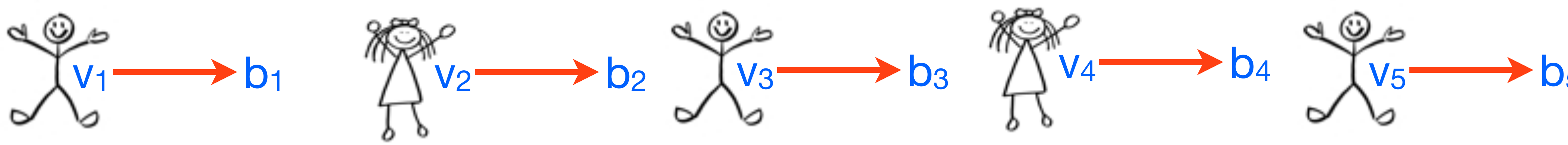


Phase 2: shuffle (permute and re-randomize) the ballots

Phase 3: threshold decrypt the shuffled ballots

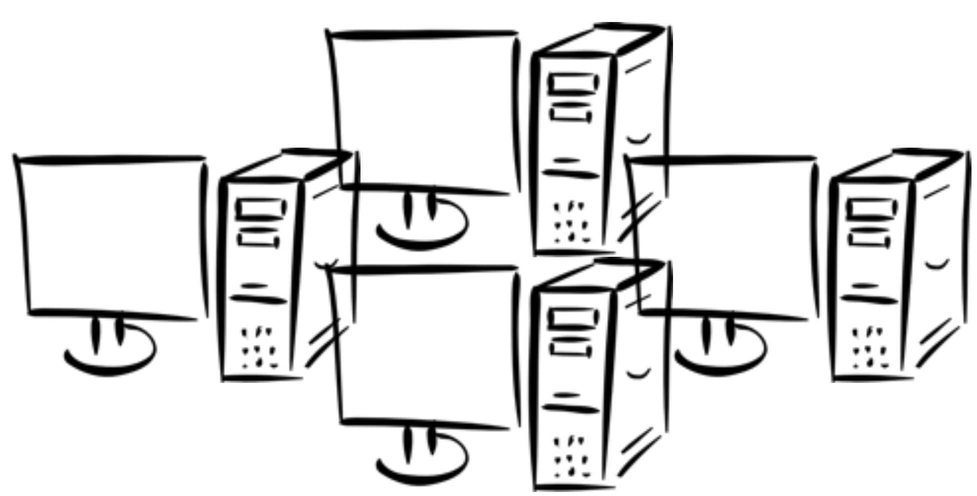
10,000-foot view of cryptographic voting

Phase 1: users encrypt votes to cast ballots



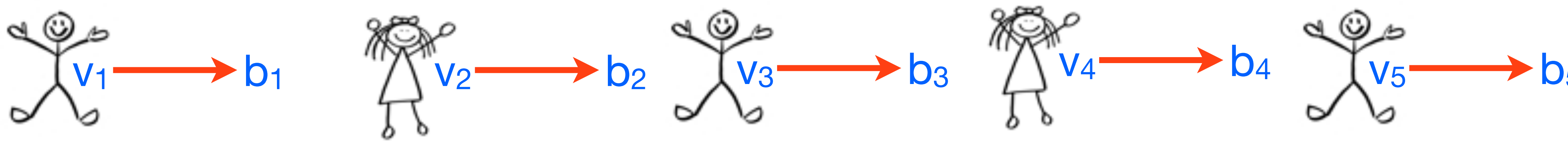
Phase 2: shuffle (permute and re-randomize) the ballots

Phase 3: threshold decrypt the shuffled ballots



10,000-foot view of cryptographic voting

Phase 1: users encrypt votes to cast ballots

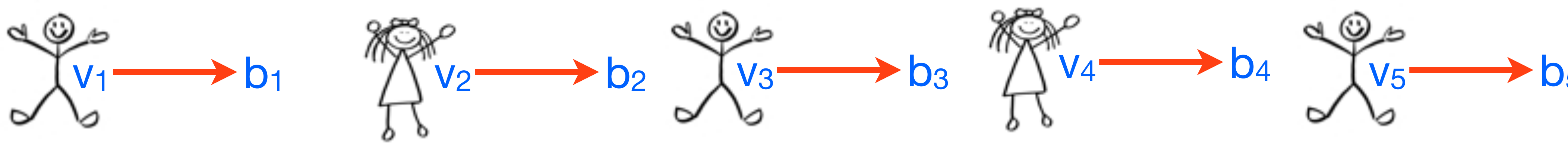


Phase 2: shuffle (permute and re-randomize) the ballots

Phase 3: threshold decrypt the shuffled ballots

10,000-foot view of cryptographic voting

Phase 1: users encrypt votes to cast ballots

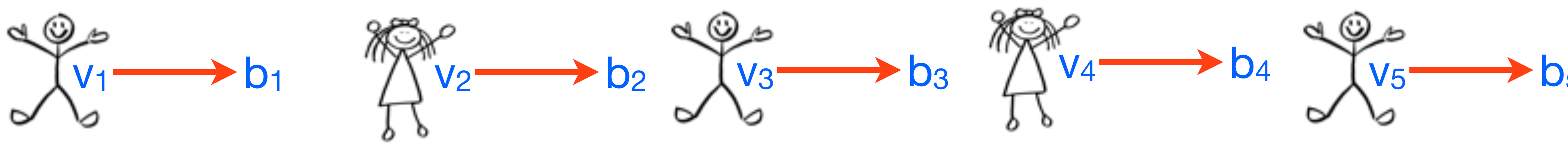


Phase 2: shuffle (permute and re-randomize) the ballots

Phase 3: threshold decrypt the shuffled ballots

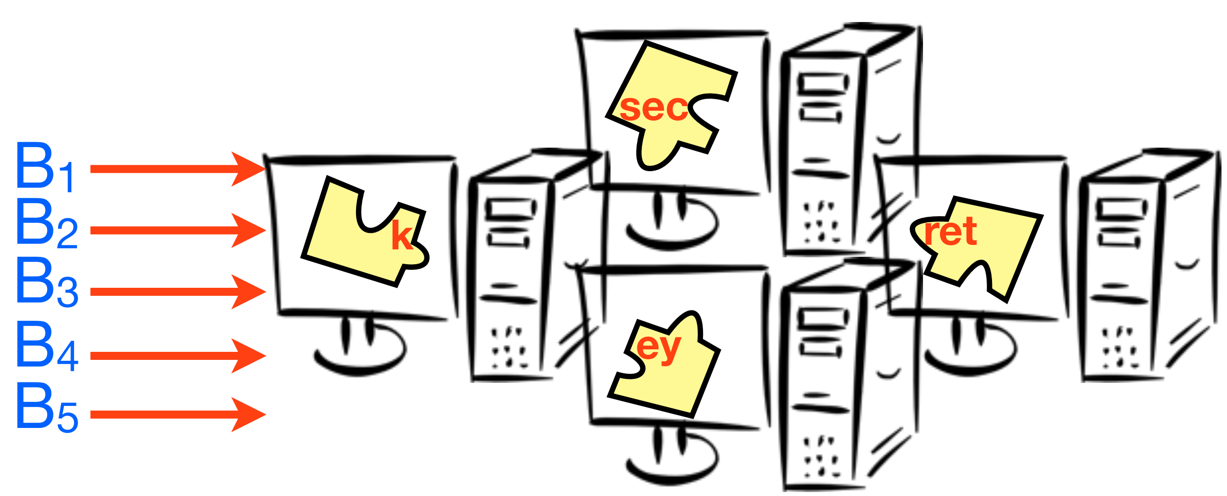
10,000-foot view of cryptographic voting

Phase 1: users encrypt votes to cast ballots



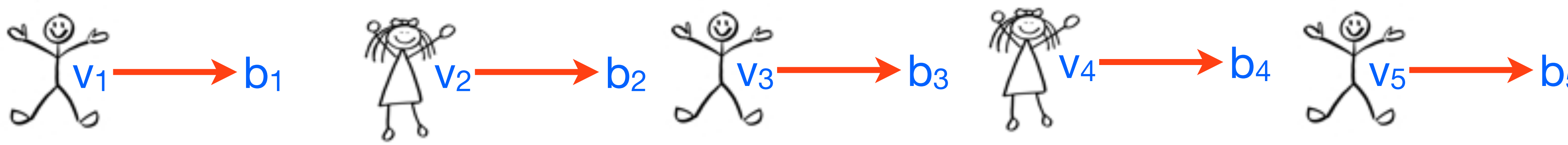
Phase 2: shuffle (permute and re-randomize) the ballots

Phase 3: threshold decrypt the shuffled ballots



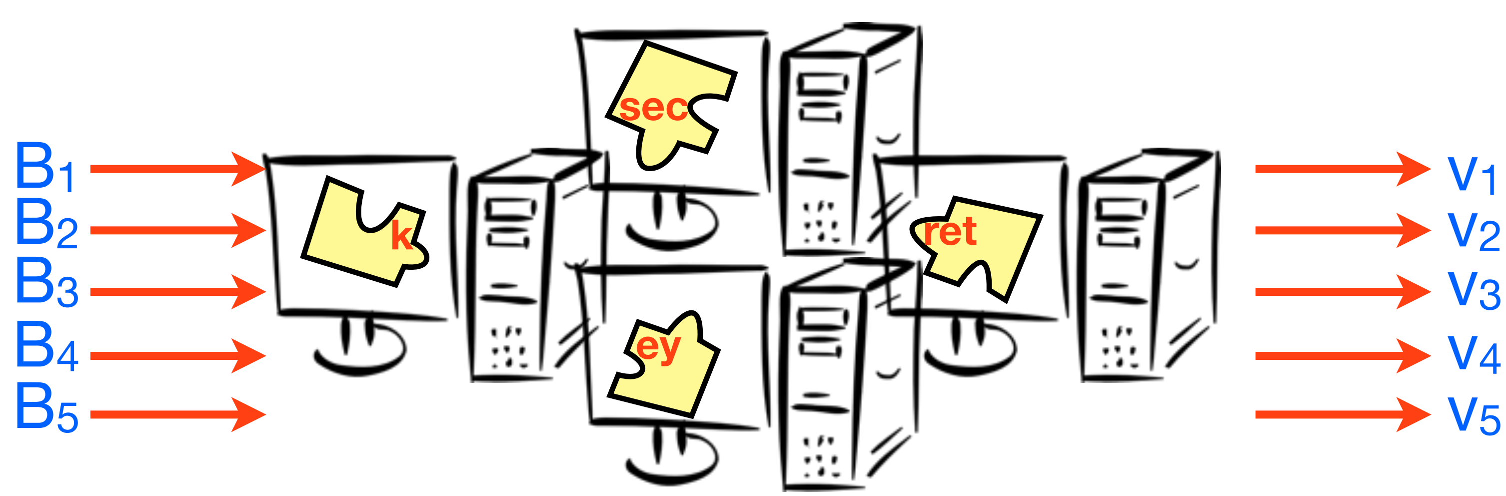
10,000-foot view of cryptographic voting

Phase 1: users encrypt votes to cast ballots



Phase 2: shuffle (permute and re-randomize) the ballots

Phase 3: threshold decrypt the shuffled ballots



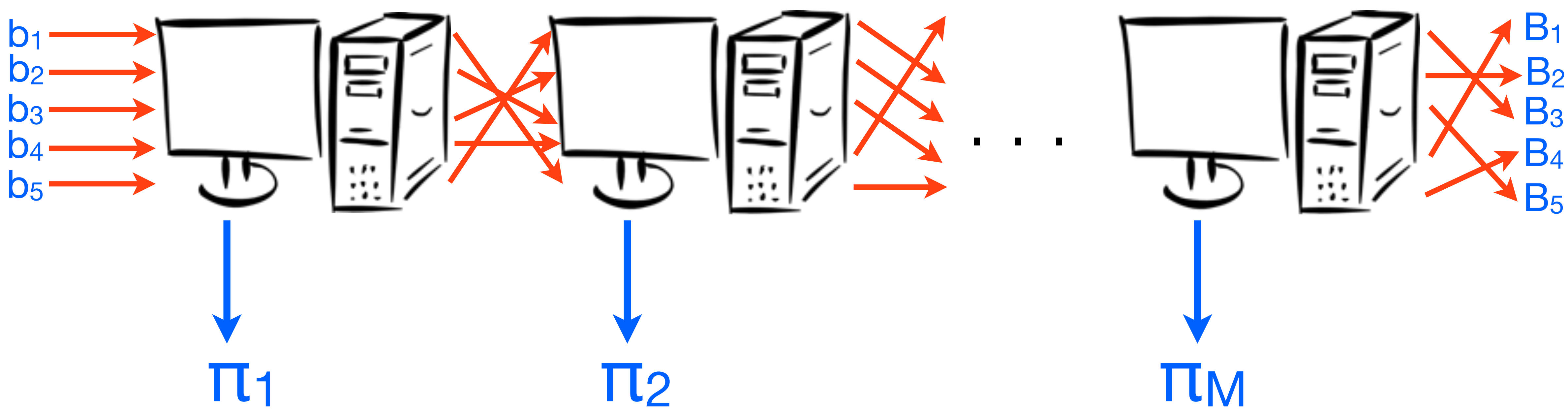
Verifiable elections [Ben87,Neff01,...,GL07]

Verifiable elections [Ben87,Neff01,...,GL07]

If we want to make this verifiable, meaning anyone can check that things went as they should, then one solution is to just add proofs everywhere

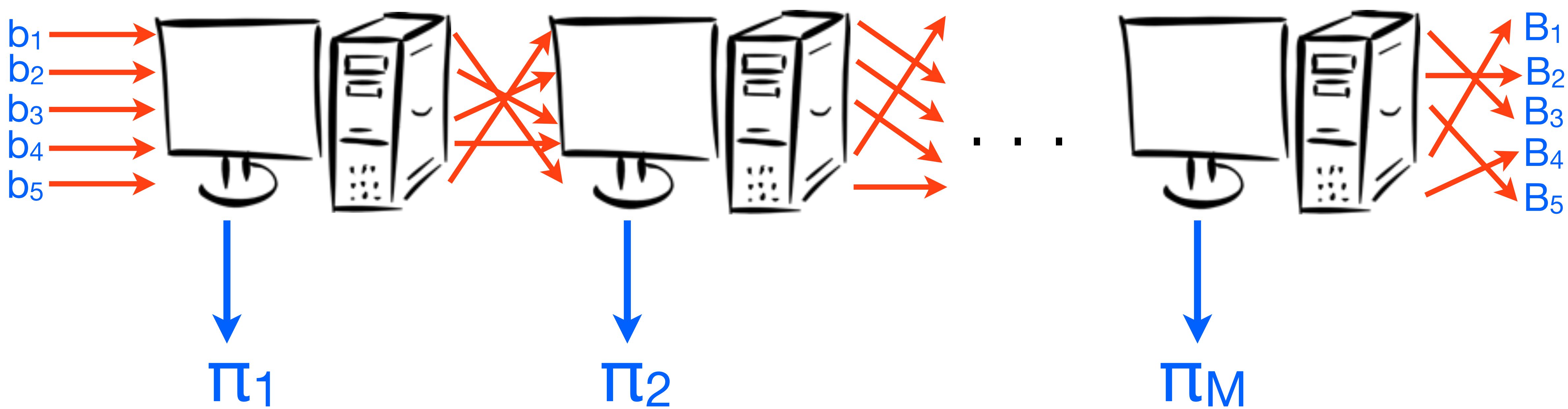
Verifiable elections [Ben87,Neff01,...,GL07]

If we want to make this verifiable, meaning anyone can check that things went as they should, then one solution is to just add proofs everywhere



Verifiable elections [Ben87,Neff01,...,GL07]

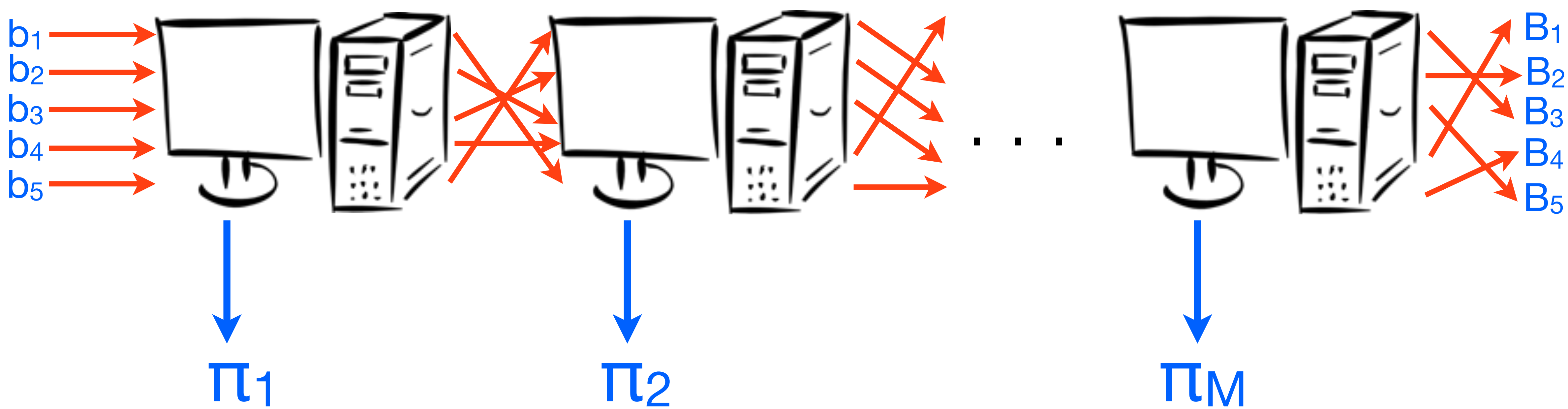
If we want to make this verifiable, meaning anyone can check that things went as they should, then one solution is to just add proofs everywhere



Then to check that election was fair, need to verify each π_i separately (for non-interactive solution; for interactive have [Abe98,FI07])

Verifiable elections [Ben87,Neff01,...,GL07]

If we want to make this verifiable, meaning anyone can check that things went as they should, then one solution is to just add proofs everywhere

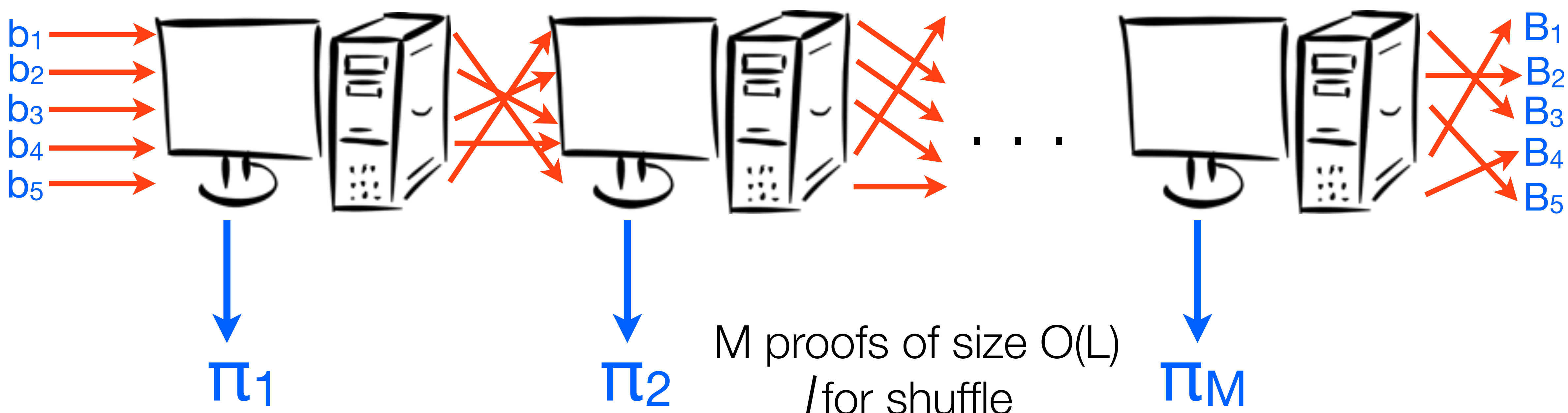


Then to check that election was fair, need to verify each π_i separately (for non-interactive solution; for interactive have [Abe98,FI07])

This means verifier input is of size $O(LM + LN)$
(L = # voters, M = # shufflers, N = # decrypters)

Verifiable elections [Ben87,Neff01,...,GL07]

If we want to make this verifiable, meaning anyone can check that things went as they should, then one solution is to just add proofs everywhere

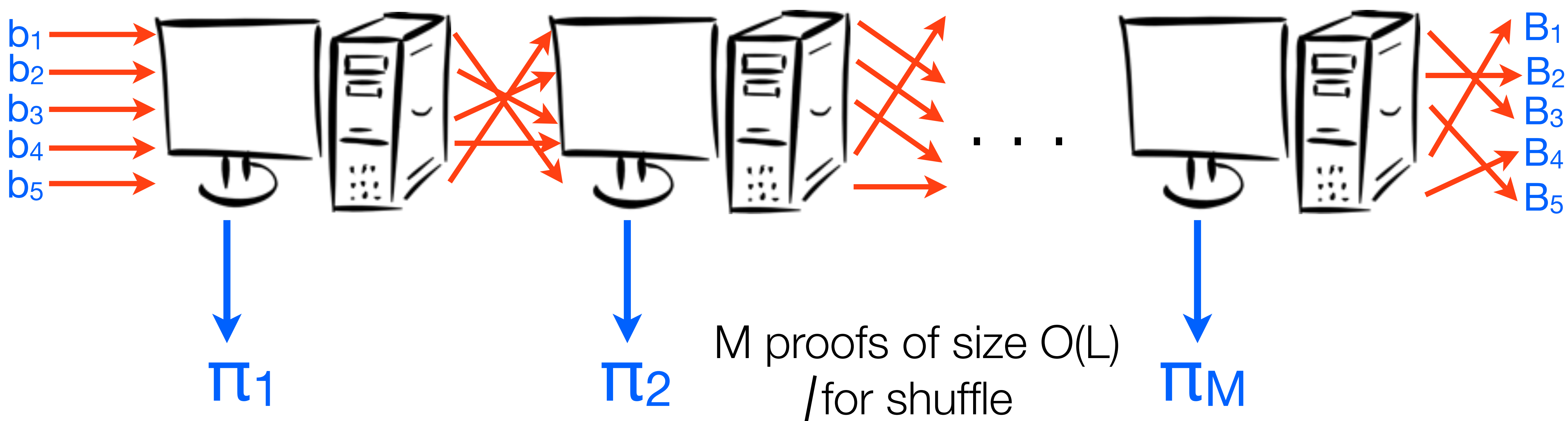


Then to check that election was fair, need to verify each π_i separately (for non-interactive solution; for interactive have [Abe98,FI07])

This means verifier input is of size O(LM + LN)
(L = # voters, M = # shufflers, N = # decrypters)

Verifiable elections [Ben87,Neff01,...,GL07]

If we want to make this verifiable, meaning anyone can check that things went as they should, then one solution is to just add proofs everywhere



Then to check that election was fair, need to verify each π_i separately (for non-interactive solution; for interactive have [Abe98,FI07])

N proofs of size O(L)
for threshold decryption

This means verifier input is of size O(LM + LN)
(L = # voters, M = # shufflers, N = # decrypters)

Our contributions

Our contributions

In this work we present an election with verifier input of size O(L+M+N)

Our contributions

In this work we present an election with verifier input of size O(L+M+N)

- Do so by using controlled-malleable zero-knowledge proofs [CKLM12]

Our contributions

In this work we present an election with verifier input of size O(L+M+N)

- Do so by using controlled-malleable zero-knowledge proofs [CKLM12]
- Define compact threshold decryption (like compactly verifiable shuffle) and a notion of vote privacy in an election

Our contributions

In this work we present an election with verifier input of size O(L+M+N)

- Do so by using controlled-malleable zero-knowledge proofs [CKLM12]
- Define compact threshold decryption (like compactly verifiable shuffle) and a notion of vote privacy in an election
- Give efficient instantiations of shuffle and threshold decryption schemes based on Decision Linear [BBS04] and two static assumptions [GL07]

Outline

Outline

Definitions

Outline

Definitions	Shuffling and decrypting

Outline

Definitions

Shuffling and decrypting

A voting scheme

Outline

Definitions

Shuffling and decrypting

A voting scheme

Conclusions

Outline

Definitions
Malleable proofs [CKLM12]
Compact shuffles [CKLM12]
Threshold decryption

Shuffling and decrypting

A voting scheme

Conclusions

Malleability for proofs [CKLM12]

Malleability for proofs [CKLM12]

Example: take a proof π_1 that b_1 is a bit and a proof π_2 that b_2 is a bit, and "maul" them somehow to get a proof that $b_1 \cdot b_2$ is a bit

Malleability for proofs [CKLM12]

Example: take a proof π_1 that b_1 is a bit and a proof π_2 that b_2 is a bit, and "maul" them somehow to get a proof that $b_1 \cdot b_2$ is a bit

More generally, a proof is malleable with respect to T if there exists an algorithm Eval that on input $(T, \{x_i, \pi_i\})$, outputs a proof π for $T(\{x_i\})$

Malleability for proofs [CKLM12]

Example: take a proof π_1 that b_1 is a bit and a proof π_2 that b_2 is a bit, and "maul" them somehow to get a proof that $b_1 \cdot b_2$ is a bit

More generally, a proof is malleable with respect to T if there exists an algorithm Eval that on input $(T,\{x_i,\pi_i\})$, outputs a proof π for $T(\{x_i\})$

- E.g., $T = \times$, x_i = "b_i is a bit"

Malleability for proofs [CKLM12]

Example: take a proof π_1 that b_1 is a bit and a proof π_2 that b_2 is a bit, and "maul" them somehow to get a proof that $b_1 \cdot b_2$ is a bit

More generally, a proof is malleable with respect to T if there exists an algorithm Eval that on input $(T,\{x_i,\pi_i\})$, outputs a proof π for $T(\{x_i\})$

- E.g., $T = \times$, x_i = "b_i is a bit"

Can define zero knowledge in the usual way as long as proofs are malleable only with respect to operations under which the language is closed

Malleability for proofs [CKLM12]

Example: take a proof π_1 that b_1 is a bit and a proof π_2 that b_2 is a bit, and "maul" them somehow to get a proof that $b_1 \cdot b_2$ is a bit

More generally, a proof is malleable with respect to T if there exists an algorithm Eval that on input $(T,\{x_i,\pi_i\})$, outputs a proof π for $T(\{x_i\})$

- E.g., $T = \times$, x_i = "b_i is a bit"

Can define zero knowledge in the usual way as long as proofs are malleable only with respect to operations under which the language is closed

But how to define a strong notion of soundness like extractability?

Controlled-malleable proofs (cm-NIZKs) [CKLM12]

Controlled-malleable proofs (cm-NIZKs) [CKLM12]

Consider an allowable set of transformations $\mathcal{T}$

Controlled-malleable proofs (cm-NIZKs) [CKLM12]

Consider an allowable set of transformations $\mathcal{T}$

High-level idea: extractor can pull out either a witness (fresh proof), or a previously queried instance and a transformation in $\mathcal{T}$ from that instance to the new one (validly transformed proof)

Controlled-malleable proofs (cm-NIZKs) [CKLM12]

Consider an allowable set of transformations $\mathcal{T}$

High-level idea: extractor can pull out either a witness (fresh proof), or a previously queried instance and a transformation in $\mathcal{T}$ from that instance to the new one (validly transformed proof)

A bit more formally: from (x,π) the extractor outputs (w,x′,T) such that either (1) (x,w)∈R or (2) x′ was queried (to simulator) and x = T(x′) for T∈$\mathcal{T}$

Controlled-malleable proofs (cm-NIZKs) [CKLM12]

Consider an allowable set of transformations $\mathcal{T}$

High-level idea: extractor can pull out either a witness (fresh proof), or a previously queried instance and a transformation in $\mathcal{T}$ from that instance to the new one (validly transformed proof)

A bit more formally: from (x,π) the extractor outputs (w,x′,T) such that either (1) (x,w)∈R or (2) x′ was queried (to simulator) and x = T(x′) for T∈$\mathcal{T}$

We call the proof CM-SSE (controlled malleable simulation sound extractable) if no PPT adversary A can violate these two conditions

Controlled-malleable proofs (cm-NIZKs) [CKLM12]

Consider an allowable set of transformations $\mathcal{T}$

High-level idea: extractor can pull out either a witness (fresh proof), or a previously queried instance and a transformation in $\mathcal{T}$ from that instance to the new one (validly transformed proof)

A bit more formally: from (x,π) the extractor outputs (w,x',T) such that either (1) $(x,w)\in R$ or (2) x' was queried (to simulator) and $x = T(x')$ for $T\in\mathcal{T}$

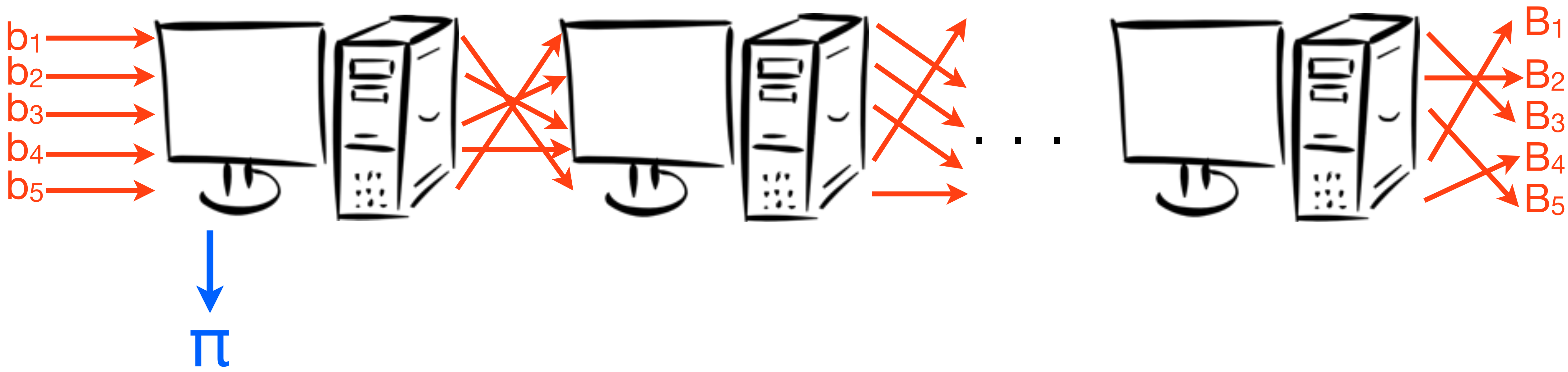
We call the proof CM-SSE (controlled malleable simulation sound extractable) if no PPT adversary A can violate these two conditions

If a proof is zero knowledge, CM-SSE, and strongly derivation private, then we call it a cm-NIZK

(like function privacy for homomorphic encryption)

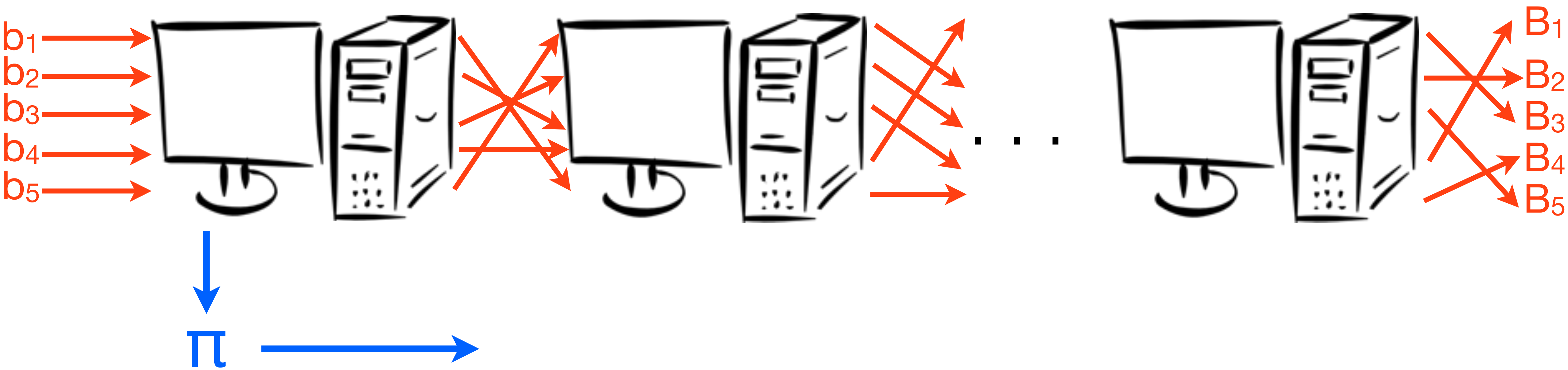
Compactly verifiable shuffles [CKLM12]

Compactly verifiable shuffles [CKLM12]



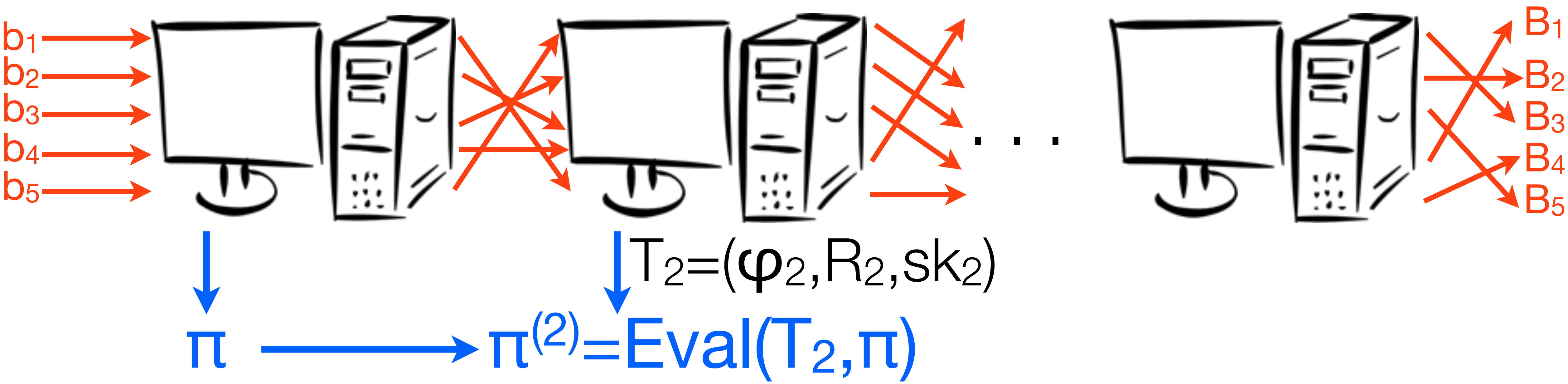
Initial mix server still outputs a fresh proof π, but now subsequent servers "maul" this proof using permutation φ_i, re-randomization R_i, and secret key sk_i

Compactly verifiable shuffles [CKLM12]



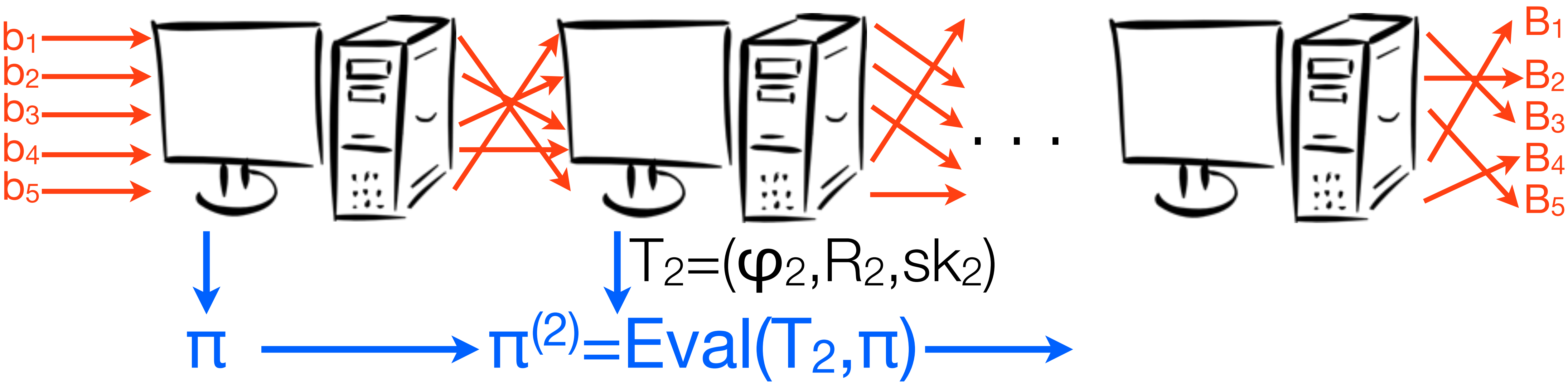
Initial mix server still outputs a fresh proof π, but now subsequent servers "maul" this proof using permutation φ_i, re-randomization R_i, and secret key sk_i

Compactly verifiable shuffles [CKLM12]



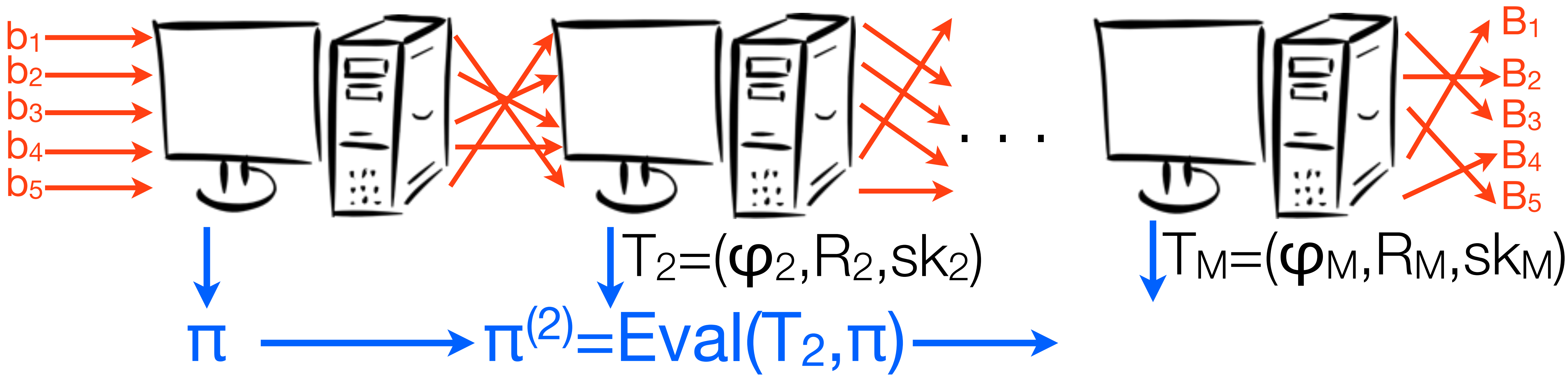
Initial mix server still outputs a fresh proof π, but now subsequent servers “maul” this proof using permutation φ_i, re-randomization R_i, and secret key sk_i

Compactly verifiable shuffles [CKLM12]



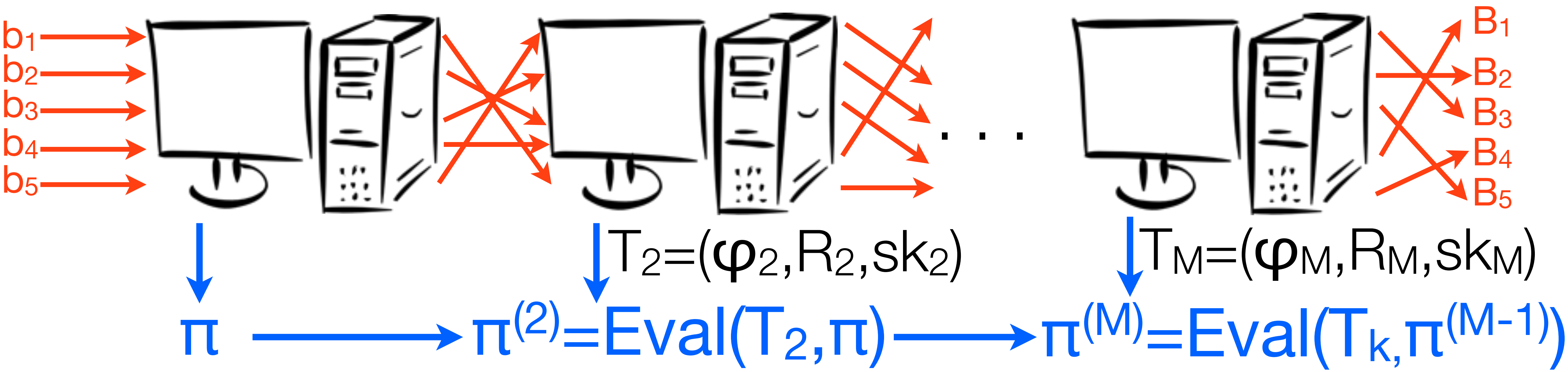
Initial mix server still outputs a fresh proof π, but now subsequent servers "maul" this proof using permutation φ_i, re-randomization R_i, and secret key sk_i

Compactly verifiable shuffles [CKLM12]



Initial mix server still outputs a fresh proof π, but now subsequent servers "maul" this proof using permutation φ_i, re-randomization R_i, and secret key sk_i

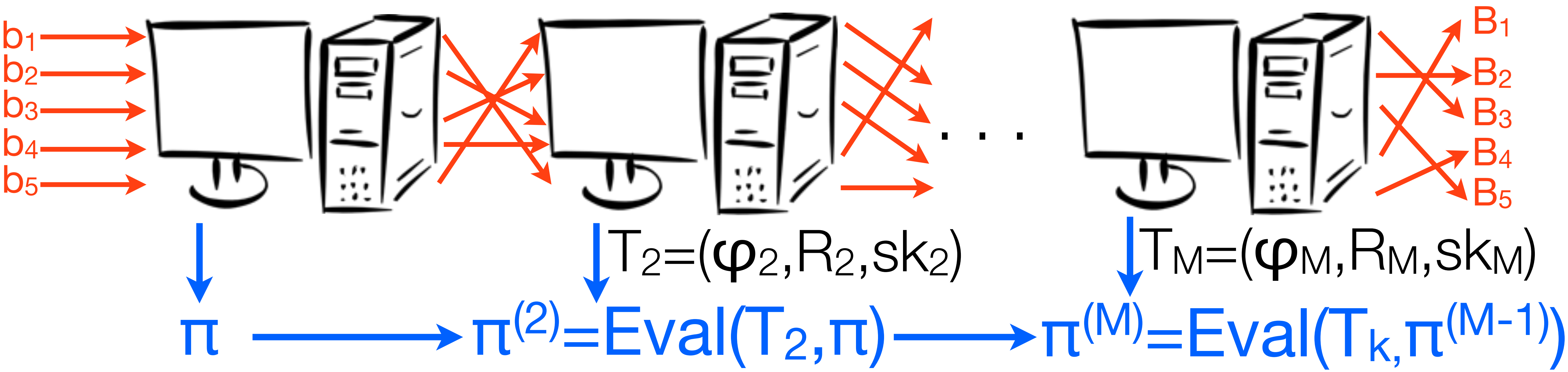
Compactly verifiable shuffles [CKLM12]



Initial mix server still outputs a fresh proof π, but now subsequent servers "maul" this proof using permutation φ_i, re-randomization R_i, and secret key sk_i

We call this shuffle compactly verifiable, as the last proof $\pi^{(M)}$ can now be used to verify the correctness of the whole shuffle (under an appropriate definition)

Compactly verifiable shuffles [CKLM12]



Initial mix server still outputs a fresh proof π, but now subsequent servers "maul" this proof using permutation φ_i, re-randomization R_i, and secret key sk_i

We call this shuffle compactly verifiable, as the last proof $\pi^{(M)}$ can now be used to verify the correctness of the whole shuffle (under an appropriate definition)

So if there are L ciphertexts and M servers, proof size can be O(L+M)

Compact threshold decryption

Compact threshold decryption

$C = \mathrm{Enc}(pk, m)$

Compact threshold decryption

$C = \mathrm{Enc}(pk, m) \rightarrow$

KeyGen

Enc

Compact threshold decryption

C=Enc(pk,m)→ 

KeyGen
Enc

Compact threshold decryption

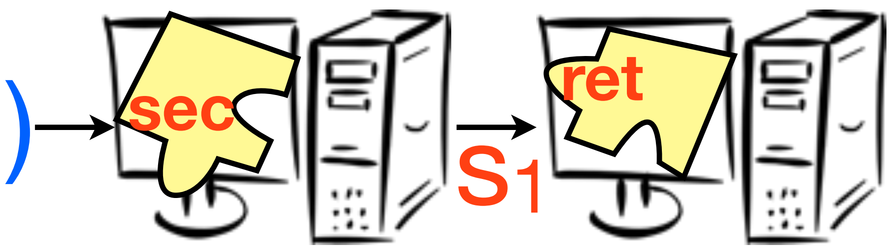
$C = Enc(pk, m) \rightarrow$ 

$\rightarrow s_1$

KeyGen
Enc

Compact threshold decryption

$C=Enc(pk,m)$

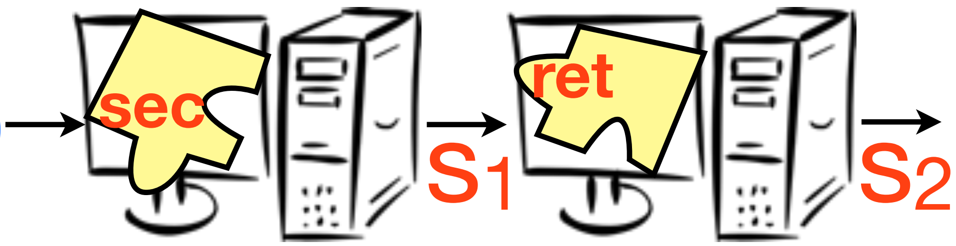


KeyGen

Enc

Compact threshold decryption

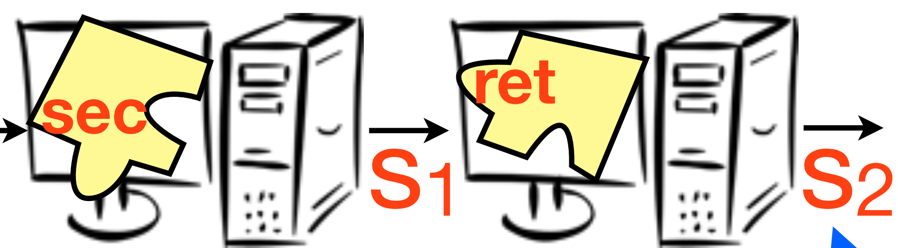
$C = Enc(pk, m)$



KeyGen
Enc

Compact threshold decryption

C=Enc(pk,m) →



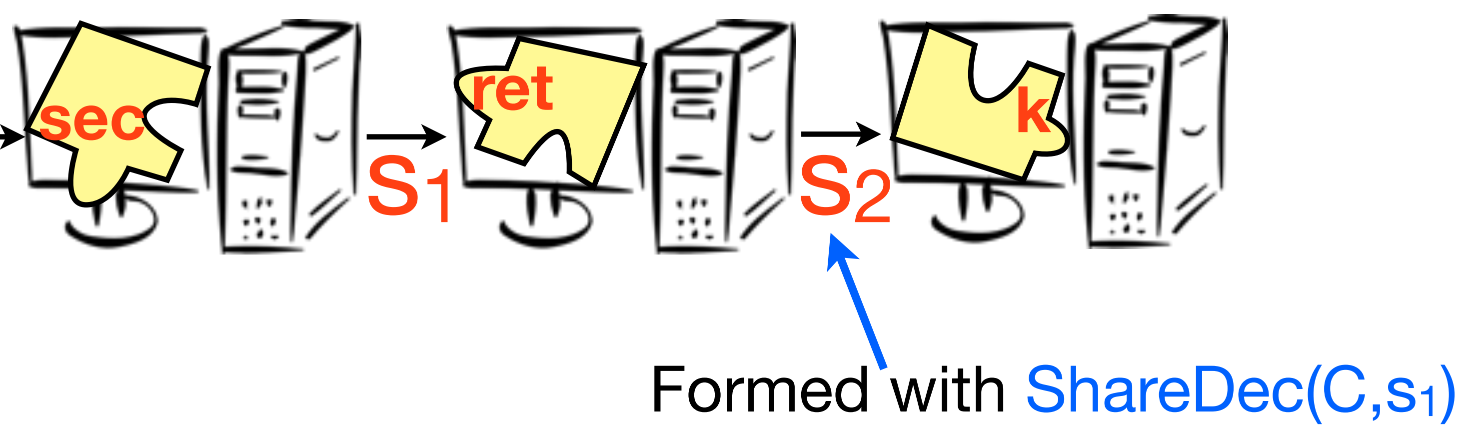
Formed with ShareDec(C,s_1)

Shares contain proof of correctness

KeyGen
Enc
ShareDec
(ShareProve)

Compact threshold decryption

$C = Enc(pk, m)$

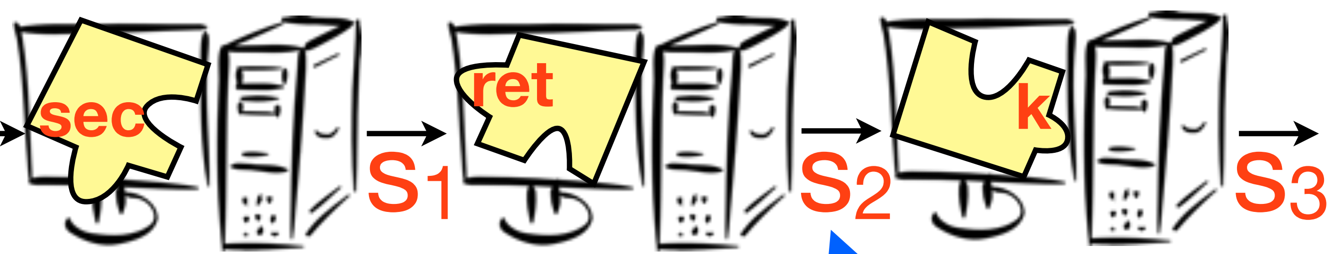


Shares contain proof of correctness

KeyGen
Enc
ShareDec
(ShareProve)
ShareVerify

Compact threshold decryption

$C=\mathrm{Enc}(pk,m)$ →



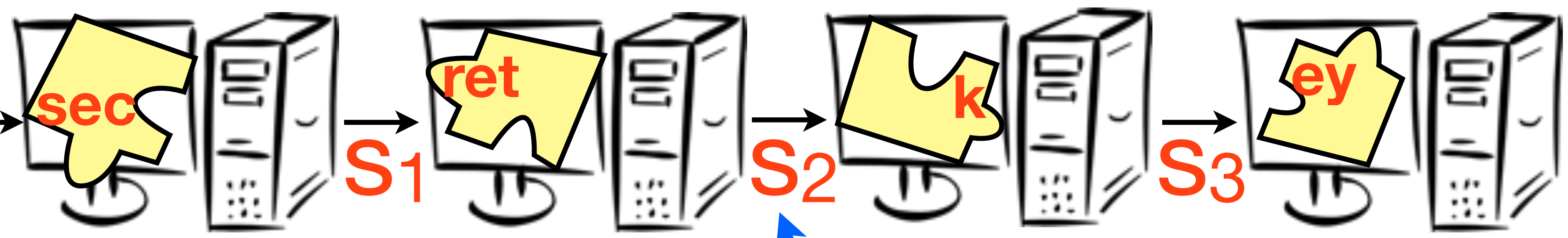
Formed with ShareDec(C,s_1)

Shares contain proof of correctness

KeyGen
Enc
ShareDec
(ShareProve)
ShareVerify

Compact threshold decryption

C=Enc(pk,m) →

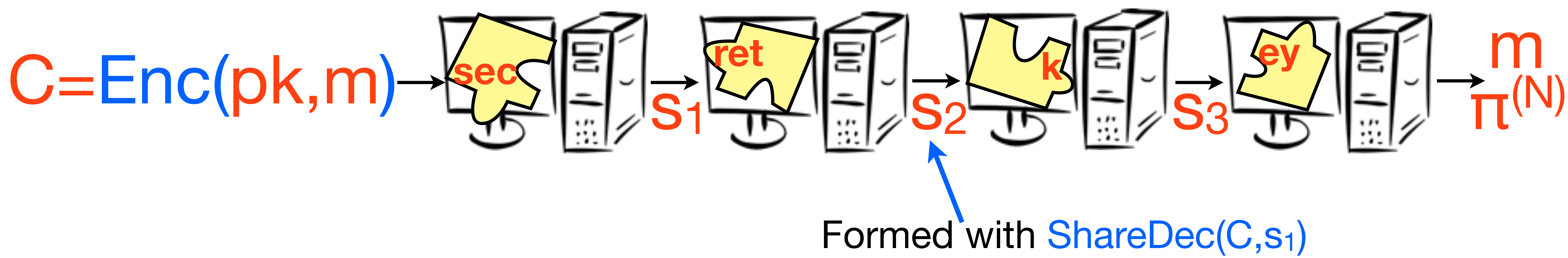


Formed with ShareDec(C,s_1)

Shares contain proof of correctness

KeyGen
Enc
ShareDec
(ShareProve)
ShareVerify

Compact threshold decryption

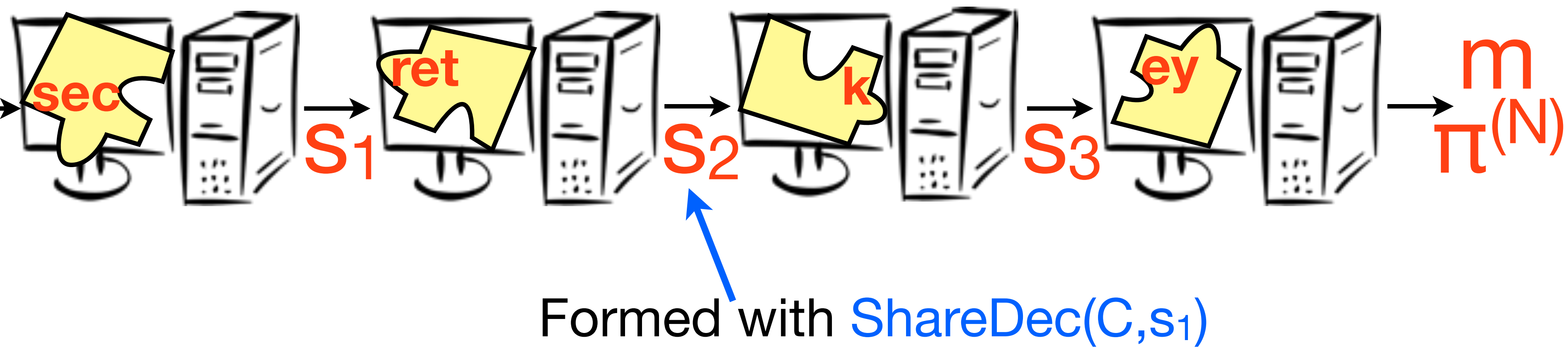


Shares contain proof of correctness

KeyGen
Enc
ShareDec
(ShareProve)
ShareVerify

Compact threshold decryption

$C=\mathrm{Enc}(pk,m)$ →

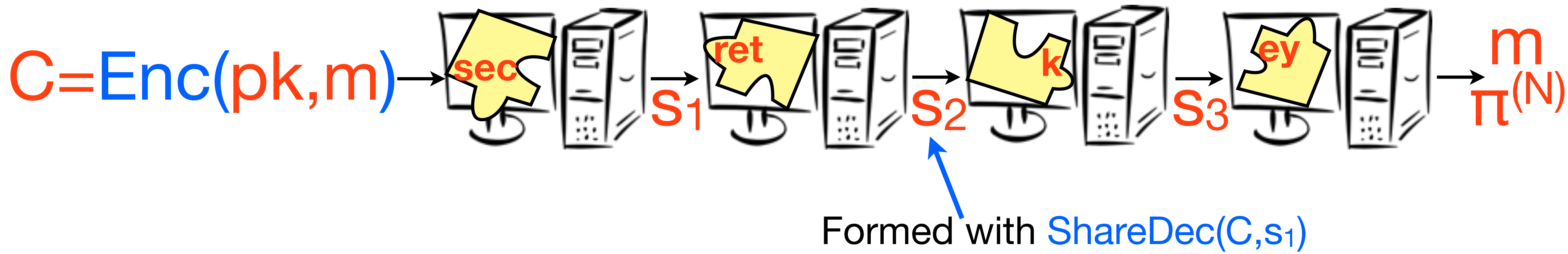


Shares contain proof of correctness

Servers can decrypt in any order; not fixed

KeyGen
Enc
ShareDec
(ShareProve)
ShareVerify

Compact threshold decryption



Shares contain proof of correctness

Servers can decrypt in any order; not fixed

Once again, final proof $\pi^{(N)}$ suffices for whole decryption, meaning total proof size can again be O(L+N) instead of O(LN) (again, under an appropriate definition)

KeyGen
Enc
ShareDec
(ShareProve)
ShareVerify

Outline

Definitions

Shuffling and decrypting

A compact verifiable shuffle

Threshold decryption

A voting scheme

Conclusions

Preliminary: BBS encryption [BBS04]

Preliminary: BBS encryption [BBS04]

Setup: generate a symmetric prime-order bilinear group (p, G, G_T, e, g)

Preliminary: BBS encryption [BBS04]

Setup: generate a symmetric prime-order bilinear group (p,G,G_T,e,g)

KeyGen(crs): $\alpha,\beta \leftarrow F_p$; $f = g^\alpha$, $h = g^\beta$; output $sk = (\alpha,\beta)$ and $pk = (f,h)$

Preliminary: BBS encryption [BBS04]

Setup: generate a symmetric prime-order bilinear group (p,G,G_T,e,g)

KeyGen(crs): $\alpha,\beta \leftarrow F_p$; $f = g^\alpha$, $h = g^\beta$; output sk = (α,β) and pk = (f,h)

Enc(crs,pk,M): $r,s \leftarrow F_p$; $u = f^r$, $v = h^s$, $w = g^{r+s}M$; return (u,v,w)

Preliminary: BBS encryption [BBS04]

Setup: generate a symmetric prime-order bilinear group (p,G,G_T,e,g)

KeyGen(crs): $\alpha,\beta \leftarrow F_p$; $f = g^\alpha$, $h = g^\beta$; output $sk = (\alpha,\beta)$ and $pk = (f,h)$

Enc(crs,pk,M): $r,s \leftarrow F_p$; $u = f^r$, $v = h^s$, $w = g^{r+s}M$; return (u,v,w)

Dec(crs,sk,(u,v,w)): return $u^{-1/\alpha}v^{-1/\beta}w$

Part 1: Compact verifiable shuffle

Part 1: Compact verifiable shuffle

Our concrete shuffle is based on the Groth-Lu shuffle [GL07]

Part 1: Compact verifiable shuffle

Our concrete shuffle is based on the Groth-Lu shuffle [GL07]

- CRS of size O(M), proofs of size O(L) (but M of them)
- Based on static pairing-based assumptions

Part 1: Compact verifiable shuffle

Our concrete shuffle is based on the Groth-Lu shuffle [GL07]

- CRS of size O(M), proofs of size O(L) (but M of them)
- Based on static pairing-based assumptions

Basically, alter their proofs and make them malleable (i.e., show they satisfy CM-friendliness)

Part 1: Compact verifiable shuffle

Our concrete shuffle is based on the Groth-Lu shuffle [GL07]

- CRS of size O(M), proofs of size O(L) (but M of them)
- Based on static pairing-based assumptions

Basically, alter their proofs and make them malleable (i.e., show they satisfy CM-friendliness)

End up with CRS of size O(M), proofs of size O(L+M) (improvement over [CKLM12], which had constant-sized CRS but proofs of size $O(L^2+M)$)

Part 2: Compact threshold decryption (KeyGen)

Part 2: Compact threshold decryption (KeyGen)

To split BBS decryption key sk = (α,β), just pick $\alpha_1,\beta_1,\ldots,\alpha_{N-1},\beta_{N-1} \leftarrow F_p$ and set $\alpha_N = -1/\alpha - \sum \alpha_i$ and $\beta_N = -1/\beta - \sum \beta_i$; then $\alpha_1 + \ldots + \alpha_N = -1/\alpha$ and $\beta_1 + \ldots + \beta_N = -1/\beta$

Part 2: Compact threshold decryption (KeyGen)

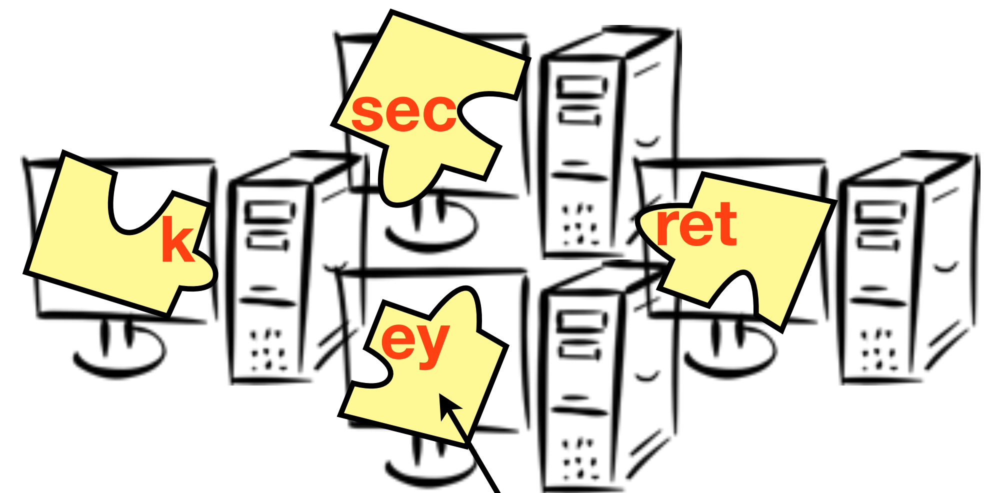
To split BBS decryption key sk = (α,β), just pick $\alpha_1,\beta_1,\ldots,\alpha_{N-1},\beta_{N-1} \leftarrow F_p$ and set $\alpha_N = -1/\alpha - \sum \alpha_i$ and $\beta_N = -1/\beta - \sum \beta_i$; then $\alpha_1 + \ldots + \alpha_N = -1/\alpha$ and $\beta_1 + \ldots + \beta_N = -1/\beta$

Part 2: Compact threshold decryption (KeyGen)

To split BBS decryption key sk = (α,β), just pick $\alpha_1,\beta_1,\ldots,\alpha_{N-1},\beta_{N-1} \leftarrow F_p$ and set $\alpha_N = -1/\alpha - \sum \alpha_i$ and $\beta_N = -1/\beta - \sum \beta_i$; then $\alpha_1 + \ldots + \alpha_N = -1/\alpha$ and $\beta_1 + \ldots + \beta_N = -1/\beta$

Part 2: Compact threshold decryption (KeyGen)

To split BBS decryption key sk = (α,β), just pick $\alpha_1,\beta_1,\ldots,\alpha_{N-1},\beta_{N-1} \leftarrow F_p$ and set $\alpha_N = -1/\alpha - \sum \alpha_i$ and $\beta_N = -1/\beta - \sum \beta_i$; then $\alpha_1 + \ldots + \alpha_N = -1/\alpha$ and $\beta_1 + \ldots + \beta_N = -1/\beta$

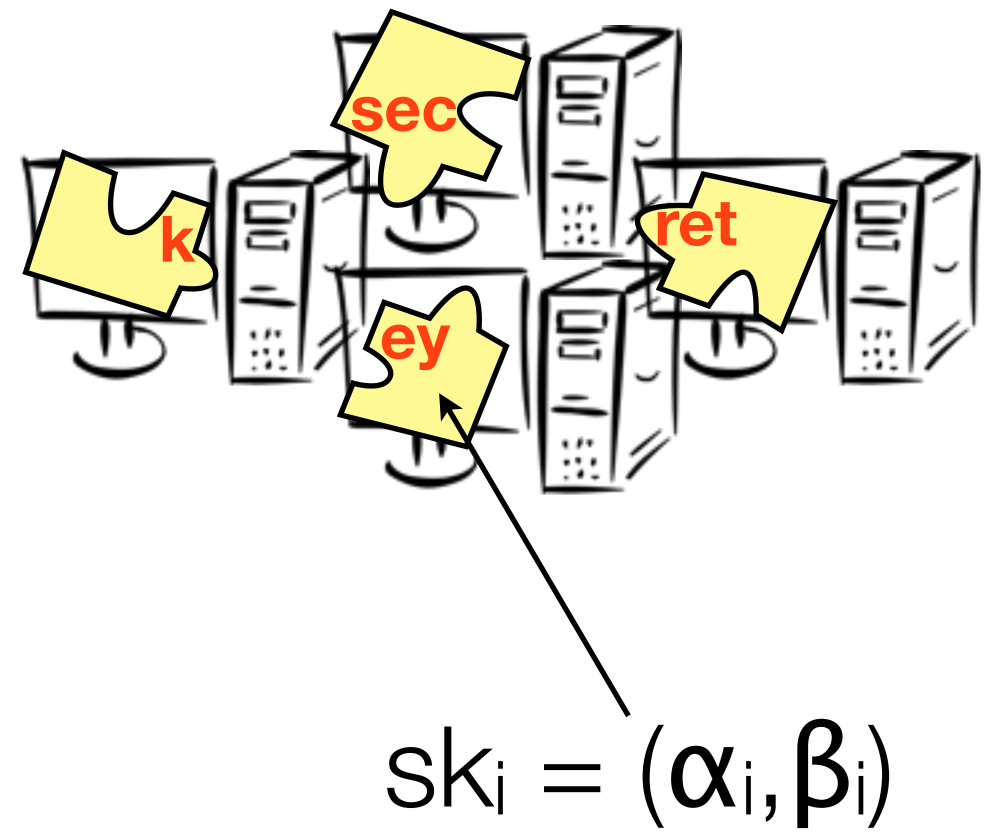


$sk_i = (\alpha_i,\beta_i)$

Part 2: Compact threshold decryption (KeyGen)

To split BBS decryption key sk = (α,β), just pick $\alpha_1,\beta_1,\ldots,\alpha_{N-1},\beta_{N-1} \leftarrow F_p$ and set $\alpha_N = -1/\alpha - \sum \alpha_i$ and $\beta_N = -1/\beta - \sum \beta_i$; then $\alpha_1 + \ldots + \alpha_N = -1/\alpha$ and $\beta_1 + \ldots + \beta_N = -1/\beta$

Observe that for c = (u,v,w) = Enc(pk,m):



Part 2: Compact threshold decryption (KeyGen)

To split BBS decryption key sk = (α,β), just pick $\alpha_1,\beta_1,\ldots,\alpha_{N-1},\beta_{N-1} \leftarrow F_p$ and set $\alpha_N = -1/\alpha - \sum \alpha_i$ and $\beta_N = -1/\beta - \sum \beta_i$; then $\alpha_1 + \ldots + \alpha_N = -1/\alpha$ and $\beta_1 + \ldots + \beta_N = -1/\beta$

Observe that for $c = (u,v,w) = Enc(pk,m)$:

$$w \prod u^{\alpha_j} \cdot v^{\beta_j} = u^{\alpha_1+\ldots+\alpha_k} \cdot v^{\beta_1+\ldots+\beta_k} \cdot w$$

$$= u^{-1/\alpha} \cdot v^{-1/\beta} w$$

$sk_i = (\alpha_i,\beta_i)$

Part 2: Compact threshold decryption (KeyGen)

To split BBS decryption key sk = (α,β), just pick $\alpha_1,\beta_1,\ldots,\alpha_{N-1},\beta_{N-1} \leftarrow F_p$ and set $\alpha_N = -1/\alpha - \sum \alpha_i$ and $\beta_N = -1/\beta - \sum \beta_i$; then $\alpha_1 + \ldots + \alpha_N = -1/\alpha$ and $\beta_1 + \ldots + \beta_N = -1/\beta$

Observe that for $c = (u,v,w) = \mathrm{Enc}(pk,m)$:

$$w \prod u^{\alpha_j} \cdot v^{\beta_j} = u^{\alpha_1+\ldots+\alpha_k} \cdot v^{\beta_1+\ldots+\beta_k} \cdot w$$

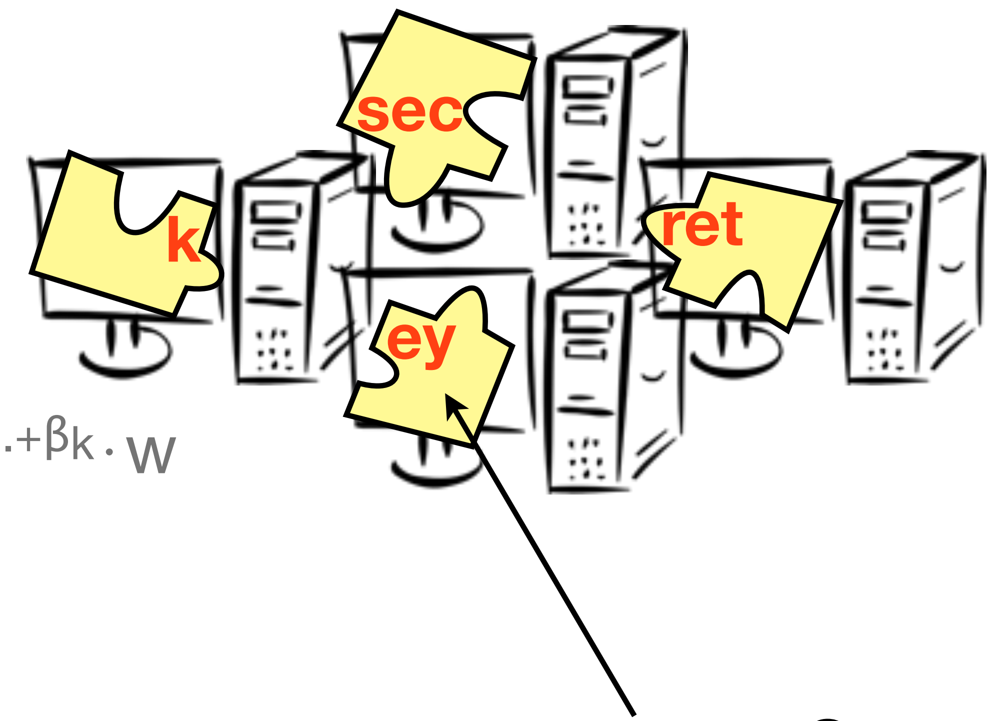
$$= u^{-1/\alpha} \cdot v^{-1/\beta} w$$

$$= m$$

$sk_i = (\alpha_i,\beta_i)$

Part 2: Compact threshold decryption (KeyGen)

To split BBS decryption key sk = (α,β), just pick $\alpha_1,\beta_1,\ldots,\alpha_{N-1},\beta_{N-1} \leftarrow F_p$ and set $\alpha_N = -1/\alpha - \sum \alpha_i$ and $\beta_N = -1/\beta - \sum \beta_i$; then $\alpha_1 + \ldots + \alpha_N = -1/\alpha$ and $\beta_1 + \ldots + \beta_N = -1/\beta$



Observe that for $c = (u,v,w) = \mathrm{Enc}(pk,m)$:

$$w \prod u^{\alpha_j} \cdot v^{\beta_j} = u^{\alpha_1+\ldots+\alpha_k} \cdot v^{\beta_1+\ldots+\beta_k} \cdot w$$

$$= u^{-1/\alpha} \cdot v^{-1/\beta} w$$

$$= m$$

$sk_i = (\alpha_i,\beta_i)$

Also want verification key $vk = (\mathrm{Com}(sk_1)=(\mathrm{Com}(\alpha_1),\mathrm{Com}(\beta_1)),\ldots,\mathrm{Com}(sk_N))$

Part 2: Compact threshold decryption (ShareDec)

Part 2: Compact threshold decryption (ShareDec)

So say decrypter with $sk_j = (\alpha_j, \beta_j)$ gets share (s, I, π)

Part 2: Compact threshold decryption (ShareDec)

So say decrypter with $sk_j = (\alpha_j, \beta_j)$ gets share (s, I, π)

partial decryption

Part 2: Compact threshold decryption (ShareDec)

So say decrypter with $sk_j = (\alpha_j, \beta_j)$ gets share (s, I, π)

partial decryption → s

participants → I

Part 2: Compact threshold decryption (ShareDec)

So say decrypter with $sk_j = (\alpha_j, \beta_j)$ gets share (s, I, π)

- s: partial decryption
- I: participants
- π: proof of correct partial decryption

Part 2: Compact threshold decryption (ShareDec)

So say decrypter with $sk_j = (\alpha_j, \beta_j)$ gets share (s, I, π)

- First check ShareVerify(s, I, π)

partial decryption

participants

proof of correct partial decryption

Part 2: Compact threshold decryption (ShareDec)

So say decrypter with $sk_j = (\alpha_j, \beta_j)$ gets share (s, I, π)

partial decryption → s; participants → I; proof of correct partial decryption → π

- First check ShareVerify(s, I, π)
- Then compute $s_j = u^{\alpha_j} \cdot v^{\beta_j}$ (initial decrypter does $u^{\alpha_k} v^{\beta_k} w$)

Part 2: Compact threshold decryption (ShareDec)

So say decrypter with $sk_j = (\alpha_j, \beta_j)$ gets share (s, I, π)

partial decryption — participants — proof of correct partial decryption

- First check ShareVerify(s, I, π)
- Then compute $s_j = u^{\alpha_j} \cdot v^{\beta_j}$ (initial decrypter does $u^{\alpha_k} v^{\beta_k} w$)
- Compute $vk_c = \prod_{i \in I} vk_i$

Part 2: Compact threshold decryption (ShareDec)

So say decrypter with $sk_j = (\alpha_j,\beta_j)$ gets share (s,I,π)

partial decryption → s; participants → I; proof of correct partial decryption → π

- First check ShareVerify(s,I,π)
- Then compute $s_j = u^{\alpha_j} \cdot v^{\beta_j}$ (initial decrypter does $u^{\alpha_k} v^{\beta_k} w$)
- Compute $vk_c = \prod_{i \in I} vk_i$
- Compute $s' = s \cdot s_j$ and $\pi' \leftarrow$ Eval(crs,T,(vk_c,c,s),π) for $T = (s_j, g^{\alpha_j}, g^{\beta_j})$

Part 2: Compact threshold decryption (ShareDec)

So say decrypter with $sk_j = (\alpha_j,\beta_j)$ gets share (s,I,π)

partial decryption (s); participants (I); proof of correct partial decryption (π)

- First check ShareVerify(s,I,π)
- Then compute $s_j = u^{\alpha_j} \cdot v^{\beta_j}$ (initial decrypter does $u^{\alpha_k}v^{\beta_k}w$)
- Compute $vk_c = \prod_{i \in I} vk_i$
- Compute $s' = s \cdot s_j$ and $\pi' \leftarrow$ Eval$(crs,T,(vk_c,c,s),\pi)$ for $T = (s_j,g^{\alpha_j},g^{\beta_j})$

(vk_c,c,s): "the participants represented in vk_c have correctly partially decrypted c to produce s"

Part 2: Compact threshold decryption (ShareDec)

So say decrypter with $sk_j = (\alpha_j, \beta_j)$ gets share (s, I, π)

partial decryption → s; participants → I; proof of correct partial decryption → π

- First check ShareVerify(s, I, π)
- Then compute $s_j = u^{\alpha_j} \cdot v^{\beta_j}$ (initial decrypter does $u^{\alpha_k} v^{\beta_k} w$)
- Compute $vk_c = \prod_{i \in I} vk_i$
- Compute $s' = s \cdot s_j$ and $\pi' \leftarrow$ Eval$(crs, T, (vk_c, c, s), \pi)$ for $T = (s_j, g^{\alpha_j}, g^{\beta_j})$

T: (1) folds s_j into s (2) folds commitments into vk_c

(vk_c, c, s): "the participants represented in vk_c have correctly partially decrypted c to produce s"

Part 2: Compact threshold decryption (ShareDec)

So say decrypter with $sk_j = (\alpha_j,\beta_j)$ gets share (s,I,π)

partial decryption → s; participants → I; proof of correct partial decryption → π

- First check ShareVerify(s,I,π)
- Then compute $s_j = u^{\alpha_j} \cdot v^{\beta_j}$ (initial decrypter does $u^{\alpha_k}v^{\beta_k}w$)
- Compute $vk_c = \prod_{i\in I} vk_i$
- Compute $s' = s \cdot s_j$ and $\pi' \leftarrow$ Eval$(crs,T,(vk_c,c,s),\pi)$ for $T = (s_j,g^{\alpha_j},g^{\beta_j})$
- Output $(s', I\cup\{j\}, \pi')$

T: (1) folds s_j into s (2) folds commitments into vk_c

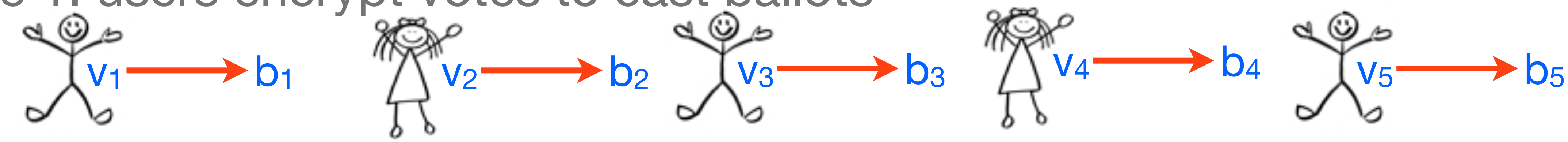
(vk_c,c,s): "the participants represented in vk_c have correctly partially decrypted c to produce s"

Outline

Definitions	Shuffling and decrypting
A voting scheme	Conclusions

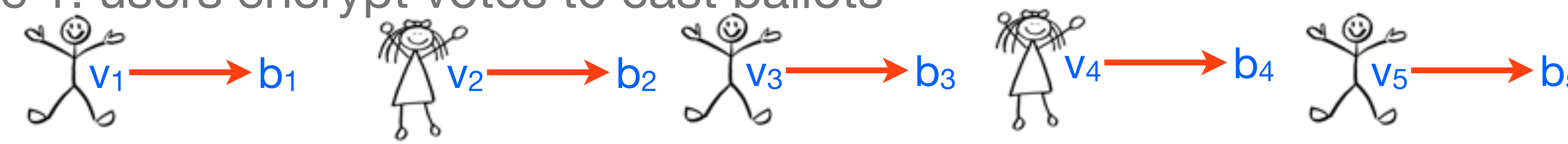
Instantiating cryptographic voting

Phase 1: users encrypt votes to cast ballots



Instantiating cryptographic voting

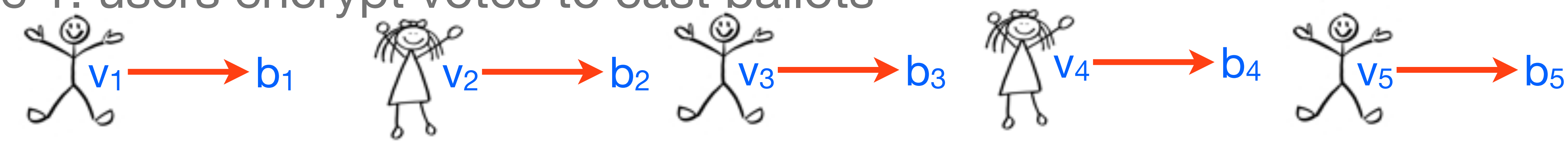
Phase 1: users encrypt votes to cast ballots



Set up KeyGen for BBS encryption, vk and crs for threshold decryption proofs, crs for shuffle proofs

Instantiating cryptographic voting

Phase 1: users encrypt votes to cast ballots

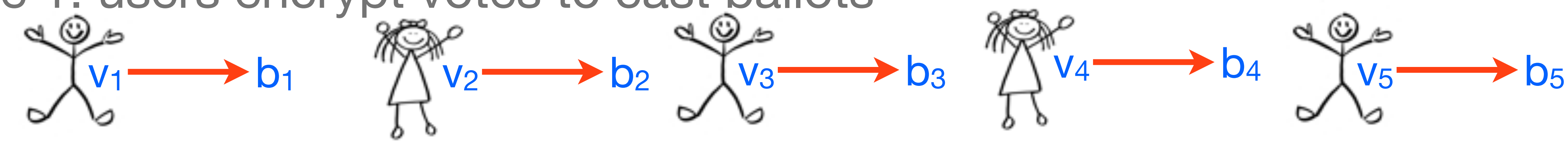


Set up KeyGen for BBS encryption, vk and crs for threshold decryption proofs, crs for shuffle proofs

For voter i, $b_i = (c_i=\text{BBSEnc}(pk,v_i),\pi_i=\text{PoK}(c_i,v_i))$

Instantiating cryptographic voting

Phase 1: users encrypt votes to cast ballots



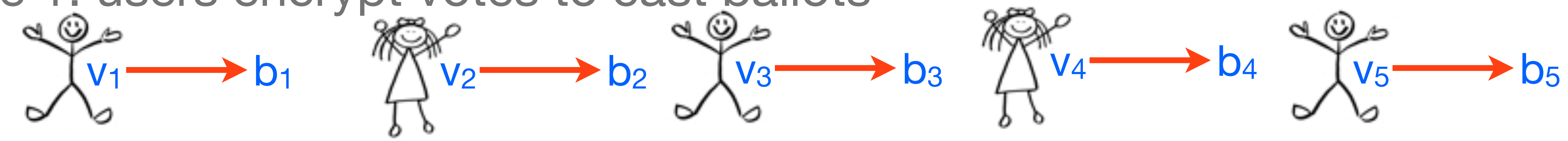
Set up KeyGen for BBS encryption, vk and crs for threshold decryption proofs, crs for shuffle proofs

For voter i, $b_i = (c_i=\mathrm{BBSEnc}(pk,v_i),\pi_i=\mathrm{PoK}(c_i,v_i))$

The vk for threshold decryption is size O(N); for shuffles the crs is size O(M)

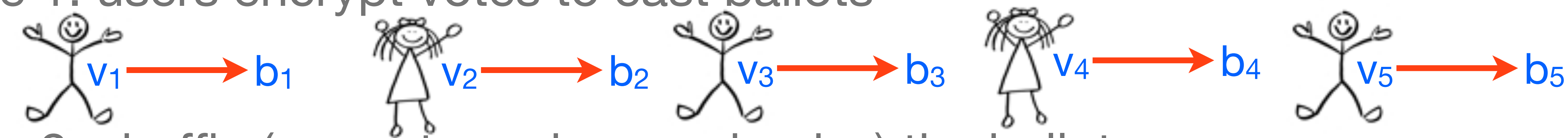
Instantiating cryptographic voting

Phase 1: users encrypt votes to cast ballots



Instantiating cryptographic voting

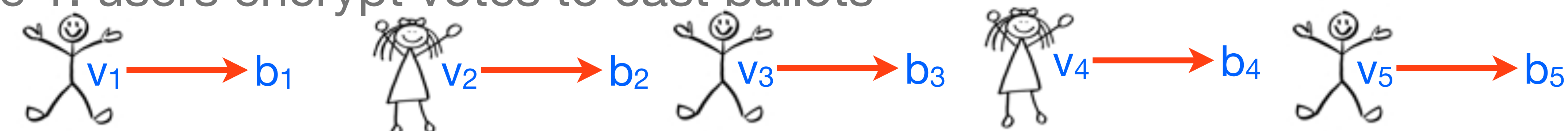
Phase 1: users encrypt votes to cast ballots



Phase 2: shuffle (permute and re-randomize) the ballots

Instantiating cryptographic voting

Phase 1: users encrypt votes to cast ballots

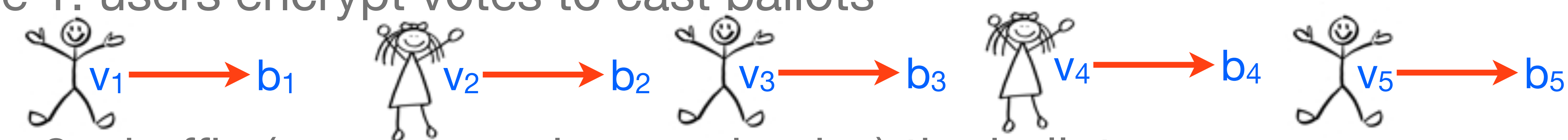


Phase 2: shuffle (permute and re-randomize) the ballots

Intermediate mix server j mauls the previous proof using $T_j = (\varphi_j, R_j, sk_j)$

Instantiating cryptographic voting

Phase 1: users encrypt votes to cast ballots



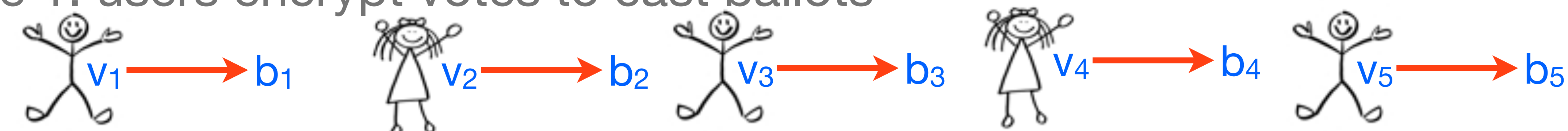
Phase 2: shuffle (permute and re-randomize) the ballots

Intermediate mix server j mauls the previous proof using $T_j = (\varphi_j, R_j, sk_j)$

Resulting proof at the end is of size O(L+M)

Instantiating cryptographic voting

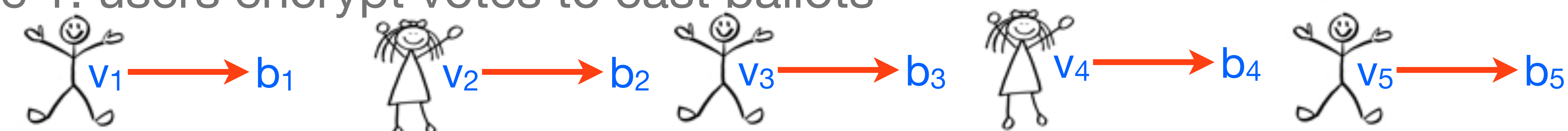
Phase 1: users encrypt votes to cast ballots



Phase 2: shuffle (permute and re-randomize) the ballots

Instantiating cryptographic voting

Phase 1: users encrypt votes to cast ballots

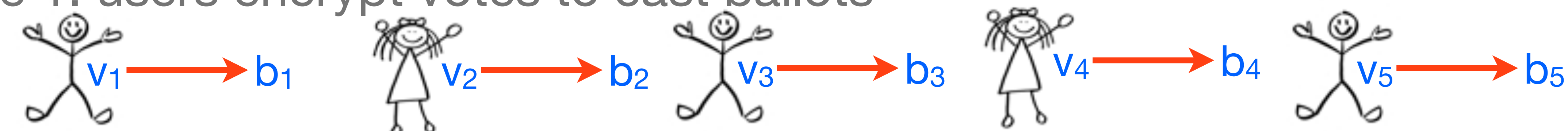


Phase 2: shuffle (permute and re-randomize) the ballots

Phase 3: threshold decrypt the shuffled ballots

Instantiating cryptographic voting

Phase 1: users encrypt votes to cast ballots



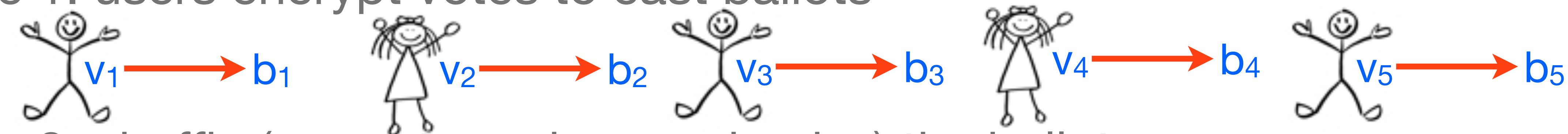
Phase 2: shuffle (permute and re-randomize) the ballots

Phase 3: threshold decrypt the shuffled ballots

Resulting proof from cumulative threshold decryption is O(L+N), so total verifier input size? $O(M) + O(N) + O(L+M) + O(L+N) = O(L+M+N)$

Instantiating cryptographic voting

Phase 1: users encrypt votes to cast ballots



Phase 2: shuffle (permute and re-randomize) the ballots

Phase 3: threshold decrypt the shuffled ballots

Resulting proof from cumulative threshold decryption is O(L+N), so total verifier input size? O(M) + O(N) + O(L+M) + O(L+N) = O(L+M+N)

Also show this satisfies notion of vote privacy for elections

Outline

Definitions	Shuffling and decrypting
A voting scheme	Conclusions

Conclusions and open problems

Conclusions and open problems

The notion of compact threshold decryption allows for proofs of size O(L+N)

Conclusions and open problems

The notion of compact threshold decryption allows for proofs of size O(L+N)

This means, theoretically, that election verification size can be O(L+M+N)

Conclusions and open problems

The notion of compact threshold decryption allows for proofs of size O(L+N)

This means, theoretically, that election verification size can be O(L+M+N)

Provided a concrete election meeting this bound

Conclusions and open problems

The notion of compact threshold decryption allows for proofs of size O(L+N)

This means, theoretically, that election verification size can be O(L+M+N)

Provided a concrete election meeting this bound

Full version is online at `eprint.iacr.org/2012/697`

Conclusions and open problems

The notion of compact threshold decryption allows for proofs of size O(L+N)

This means, theoretically, that election verification size can be O(L+M+N)

Provided a concrete election meeting this bound

Full version is online at `eprint.iacr.org/2012/697`

Thanks!
Any questions?

Regular verifiable threshold decryption [SG98]

Regular verifiable threshold decryption [SG98]

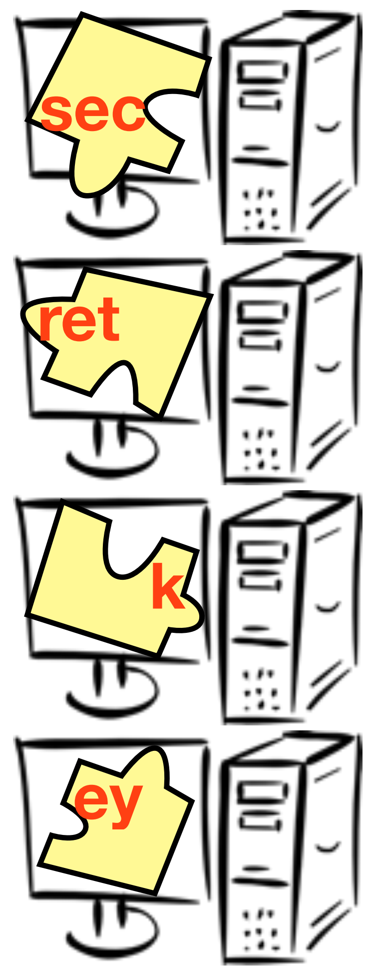
$C = Enc(pk, m)$

Regular verifiable threshold decryption [SG98]

C=Enc(pk,m)

Regular verifiable threshold decryption [SG98]

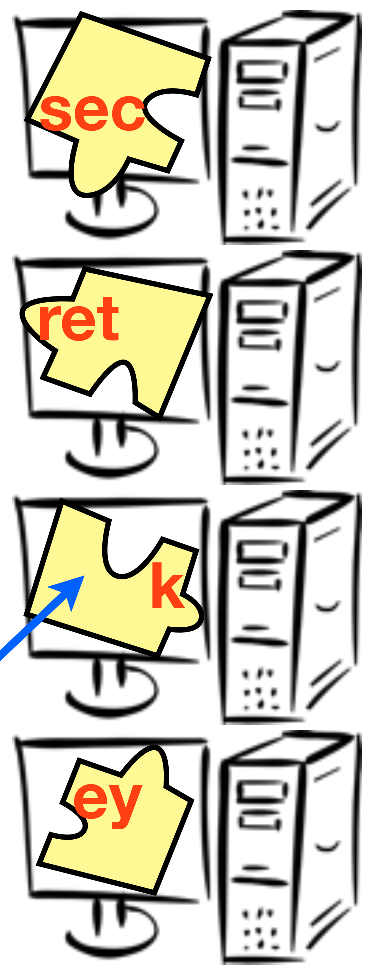
C=Enc(pk,m)



KeyGen

Regular verifiable threshold decryption [SG98]

C=Enc(pk,m)



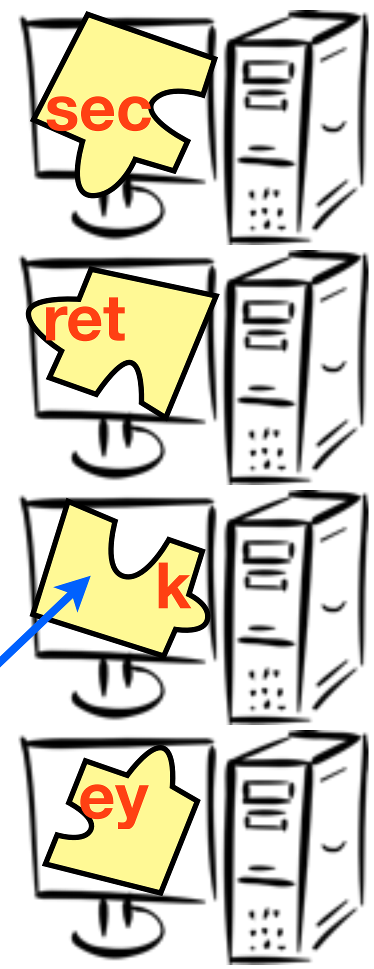
Formed with KeyGen

KeyGen

Regular verifiable threshold decryption [SG98]

C=Enc(pk,m)

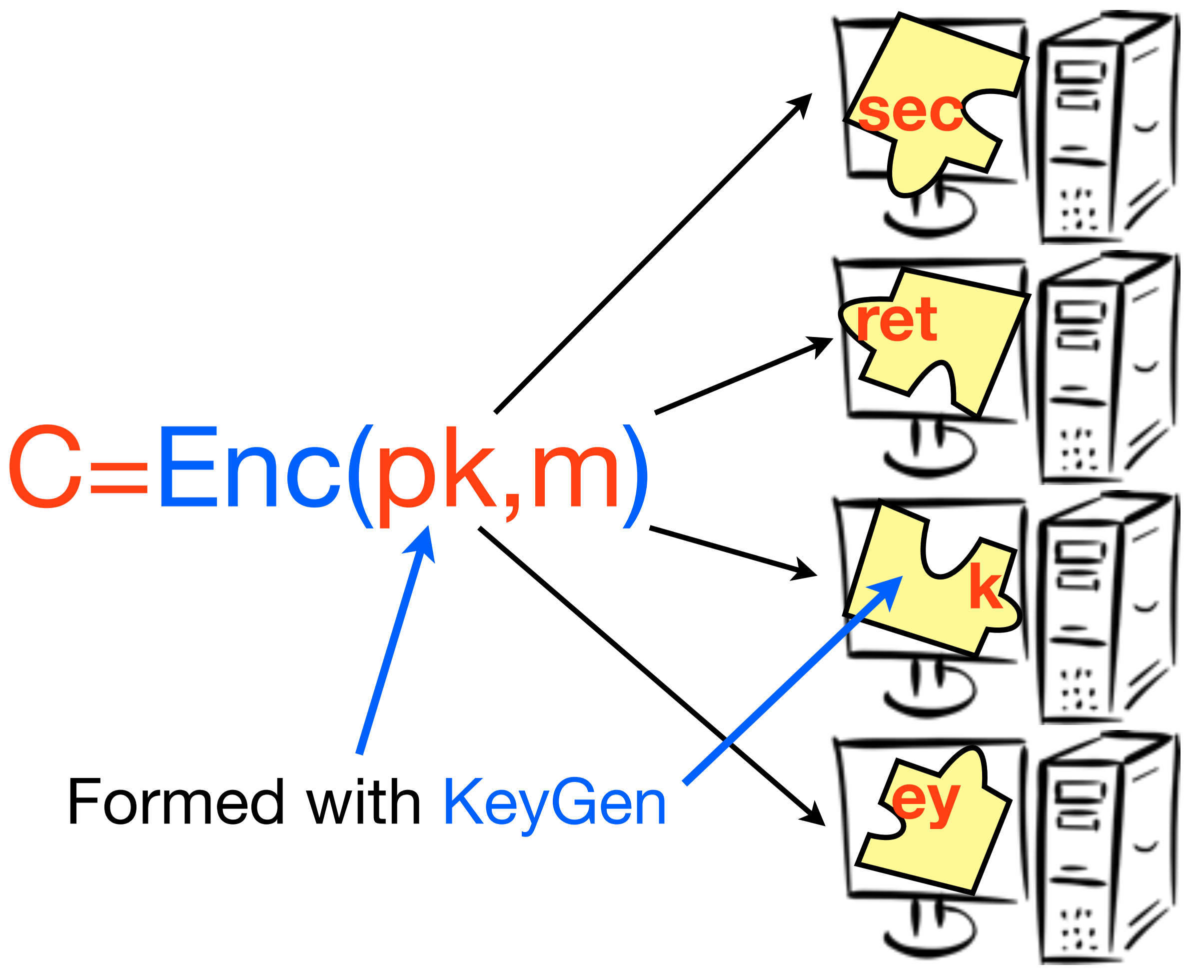
Formed with KeyGen



KeyGen

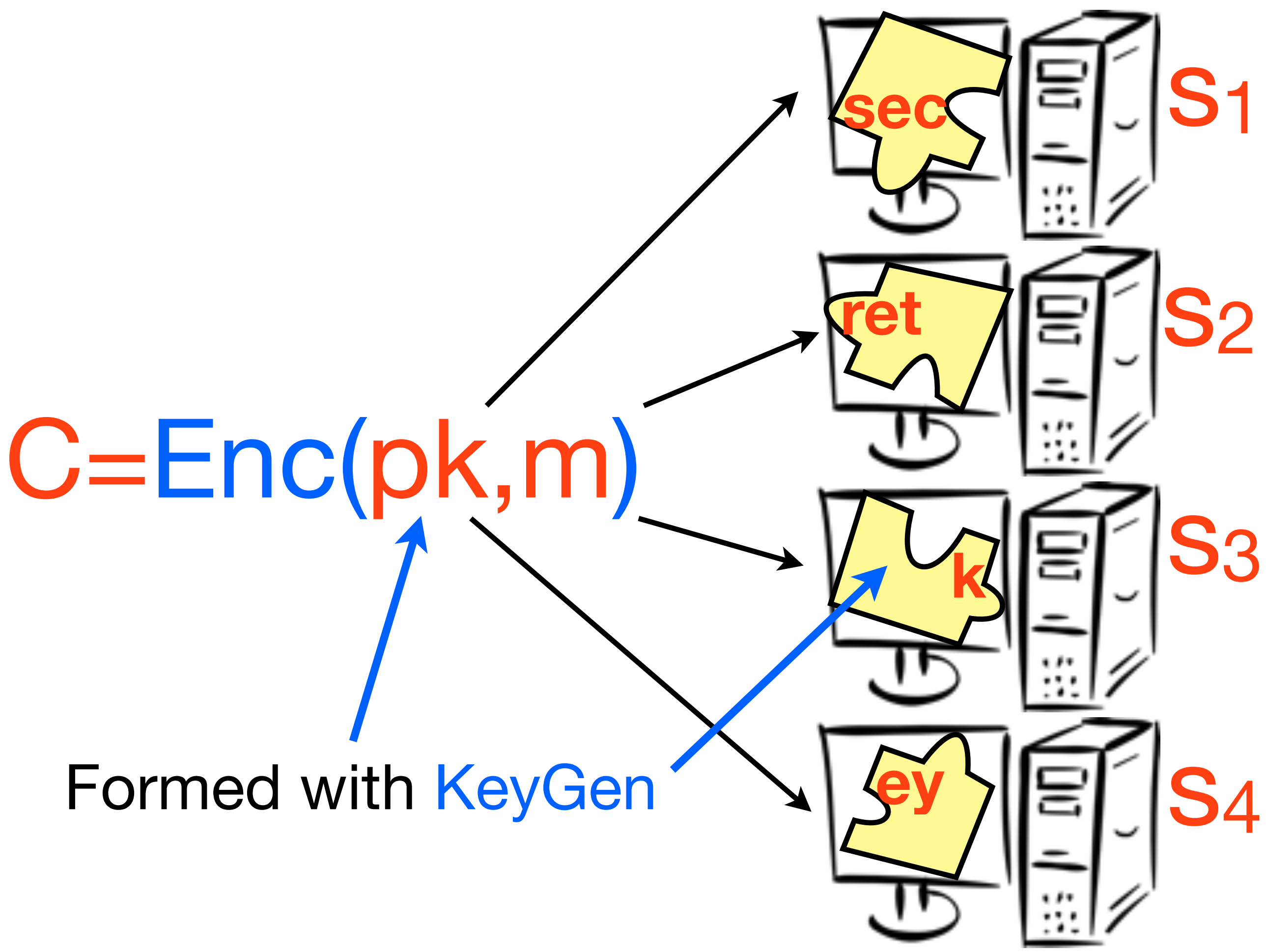
Enc

Regular verifiable threshold decryption [SG98]



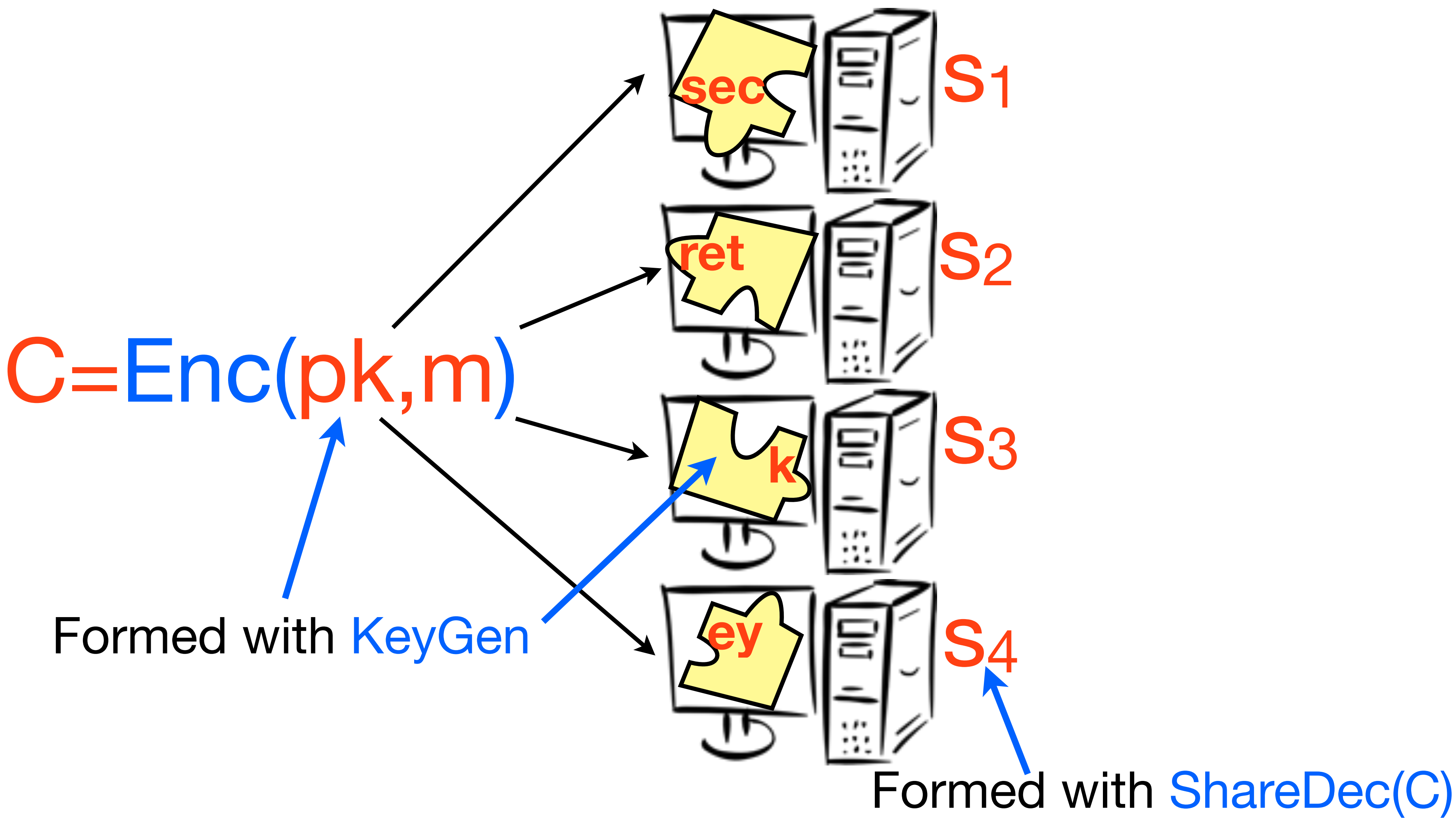
KeyGen
Enc

Regular verifiable threshold decryption [SG98]



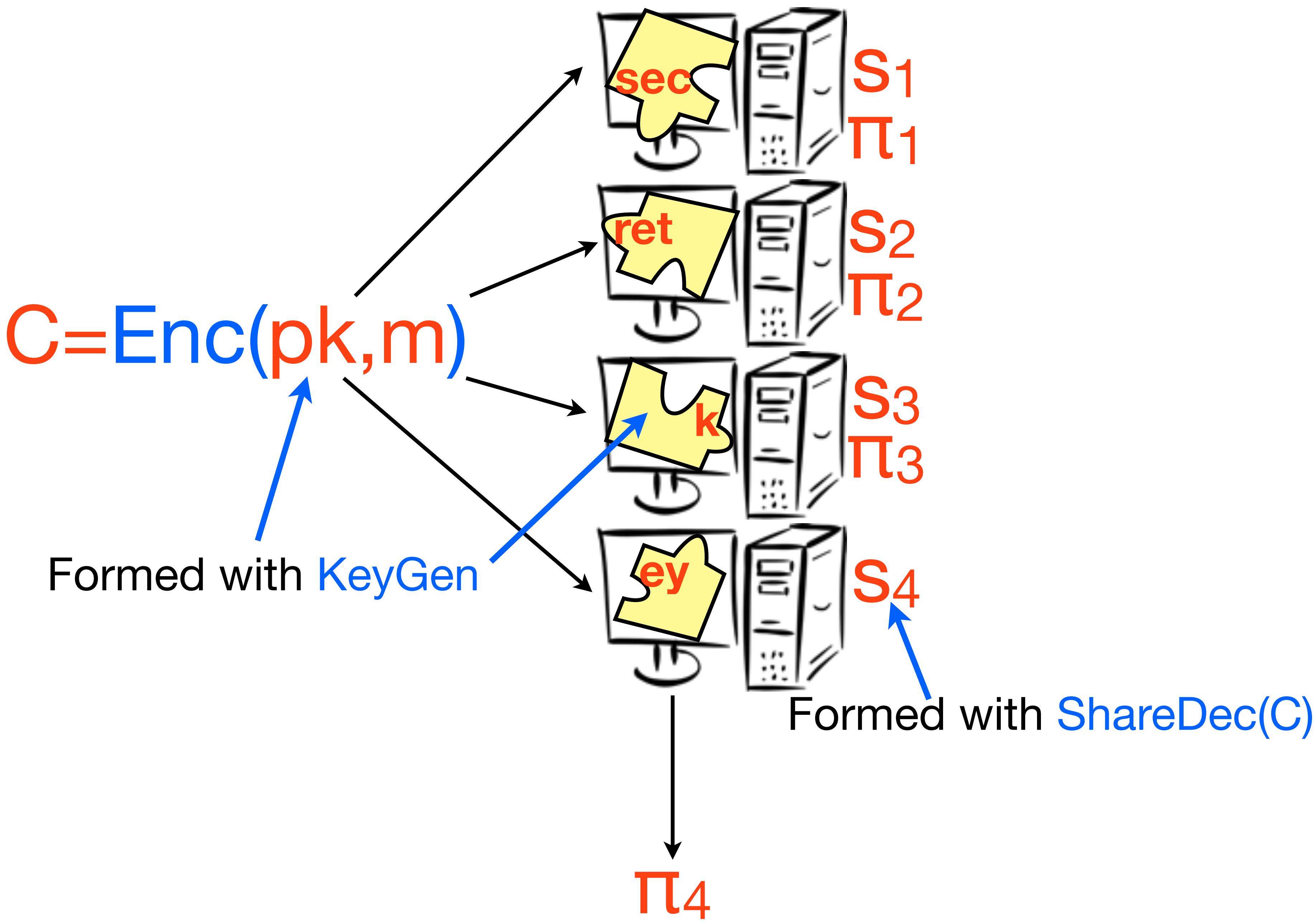
KeyGen
Enc

Regular verifiable threshold decryption [SG98]



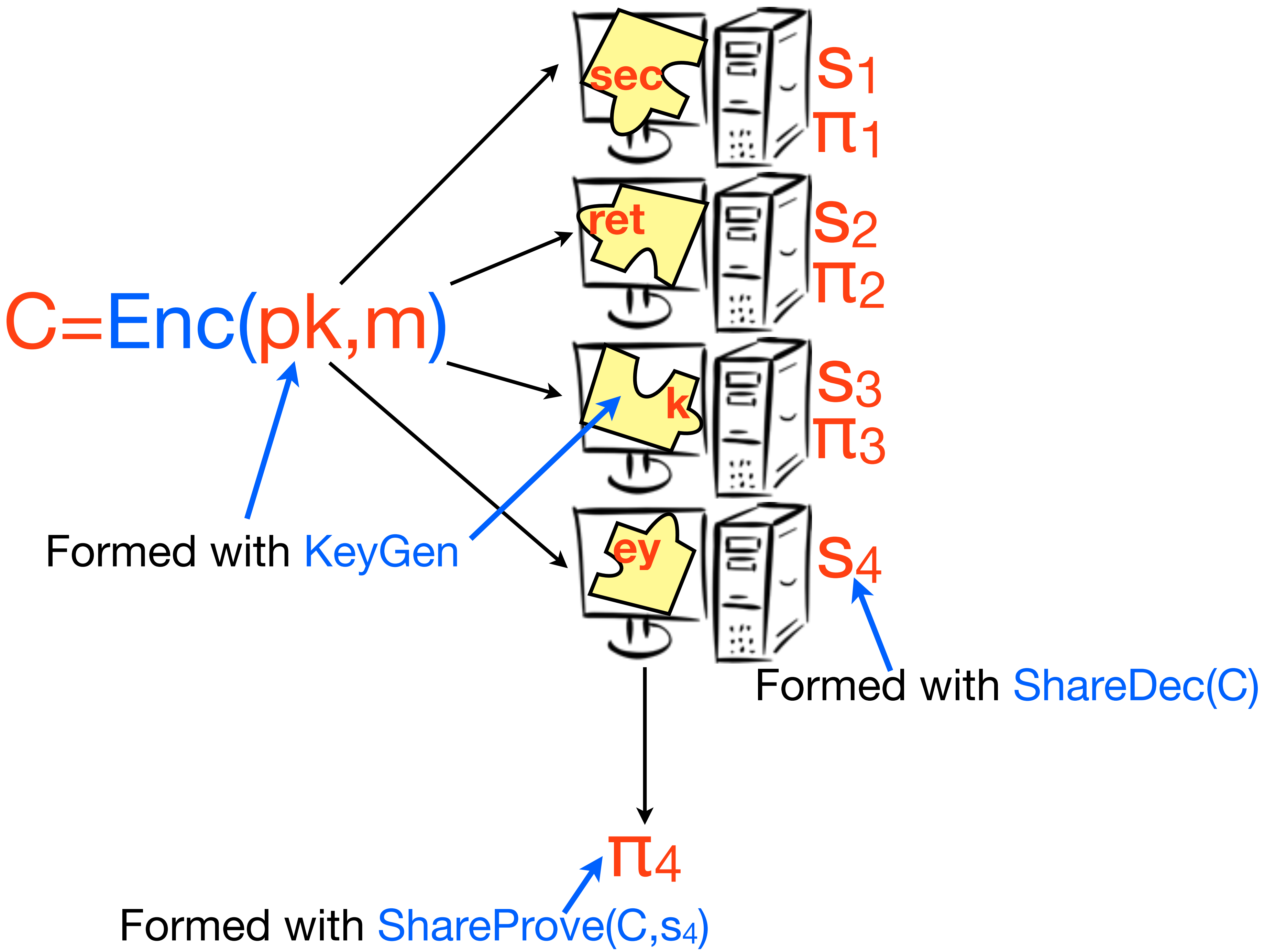
KeyGen
Enc
ShareDec

Regular verifiable threshold decryption [SG98]



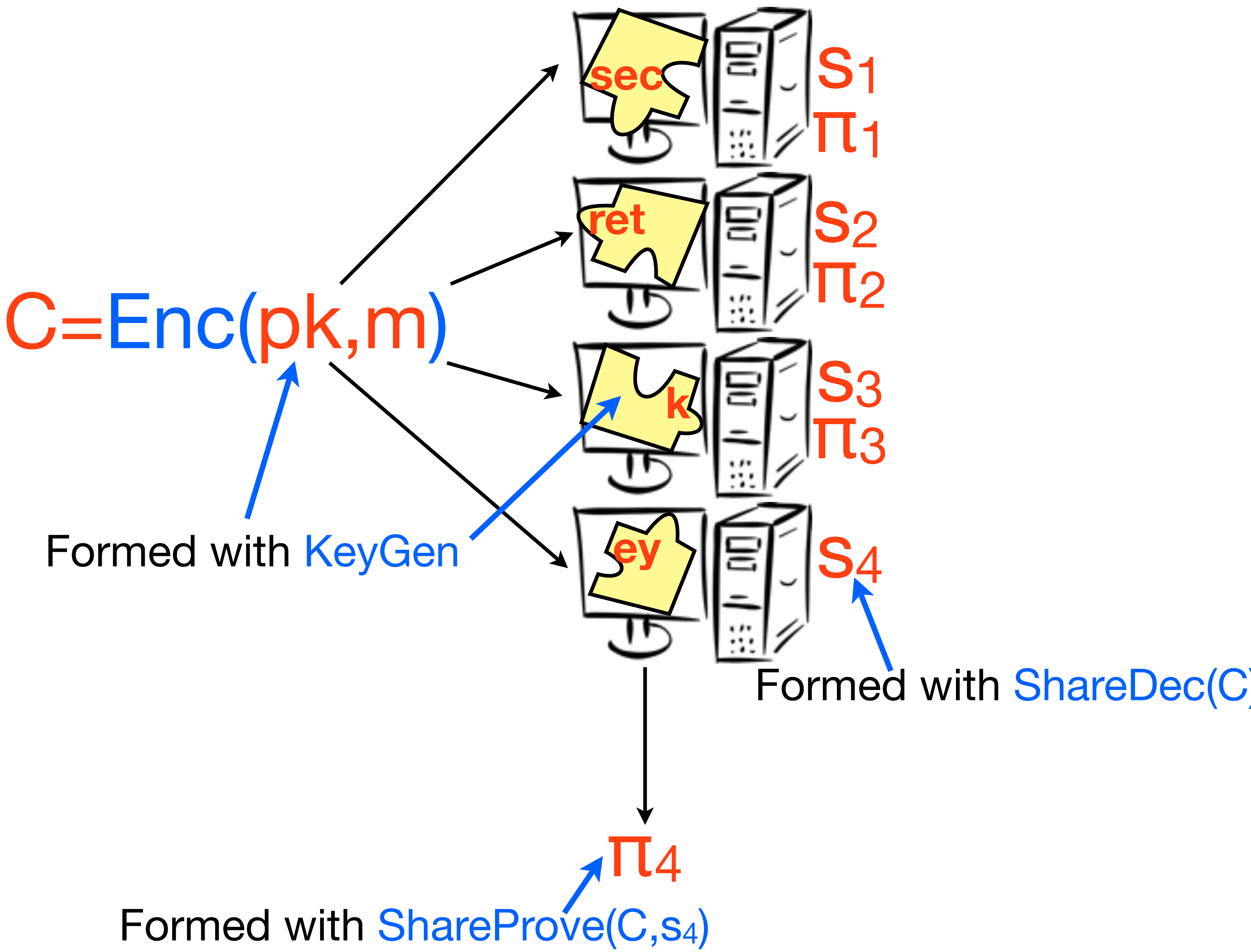
KeyGen
Enc
ShareDec

Regular verifiable threshold decryption [SG98]



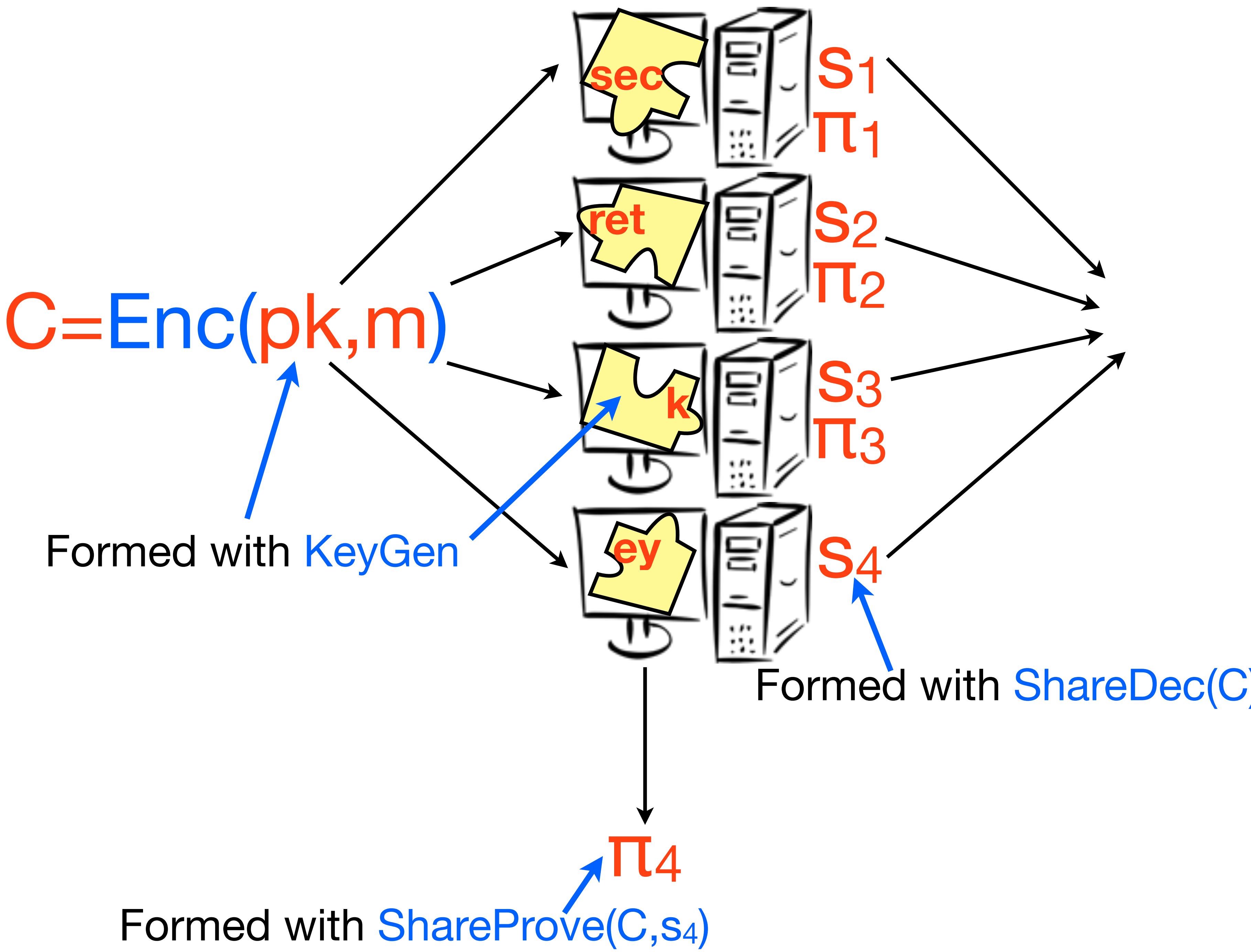
KeyGen
Enc
ShareDec
ShareProve

Regular verifiable threshold decryption [SG98]



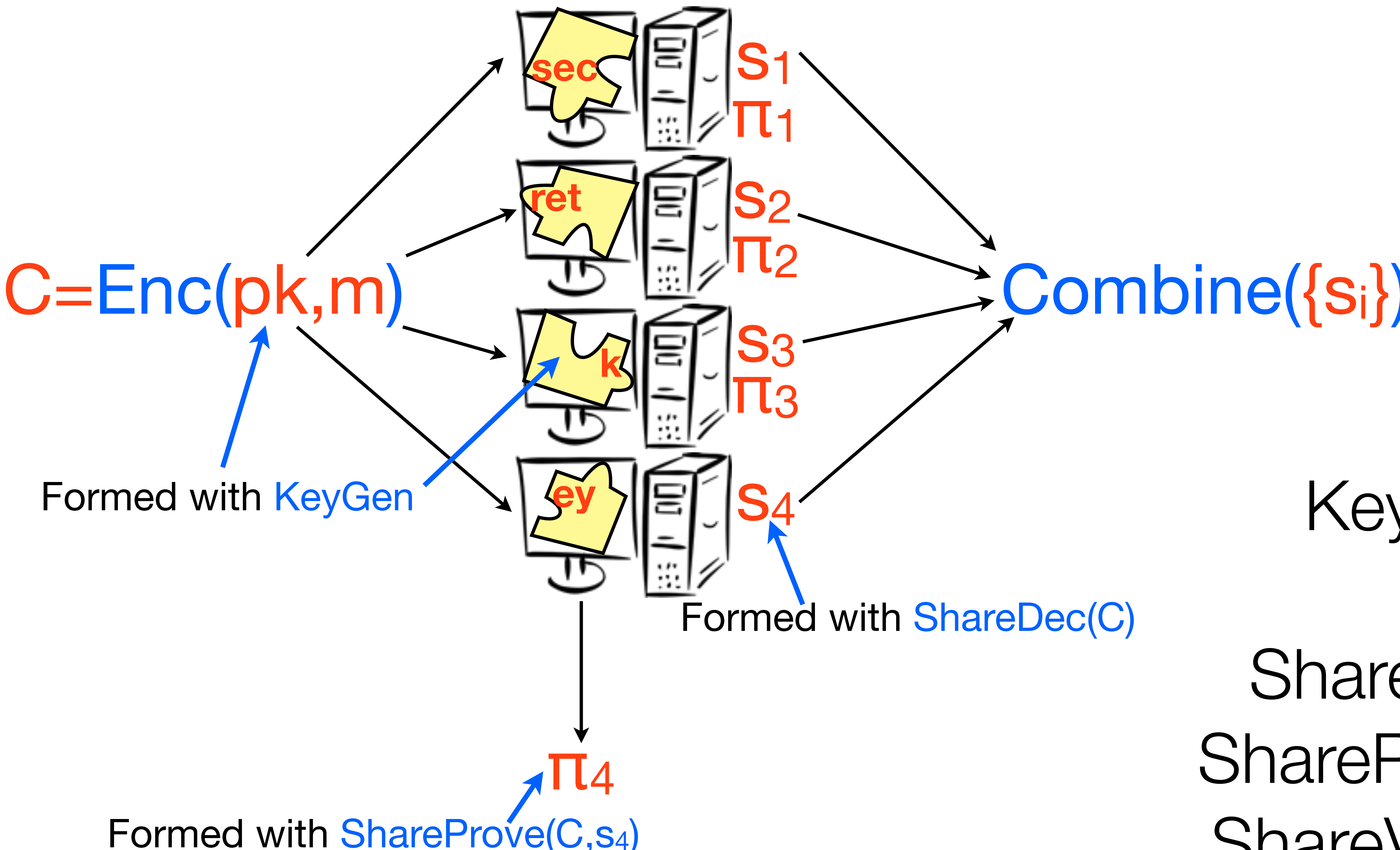
KeyGen
Enc
ShareDec
ShareProve
ShareVerify

Regular verifiable threshold decryption [SG98]



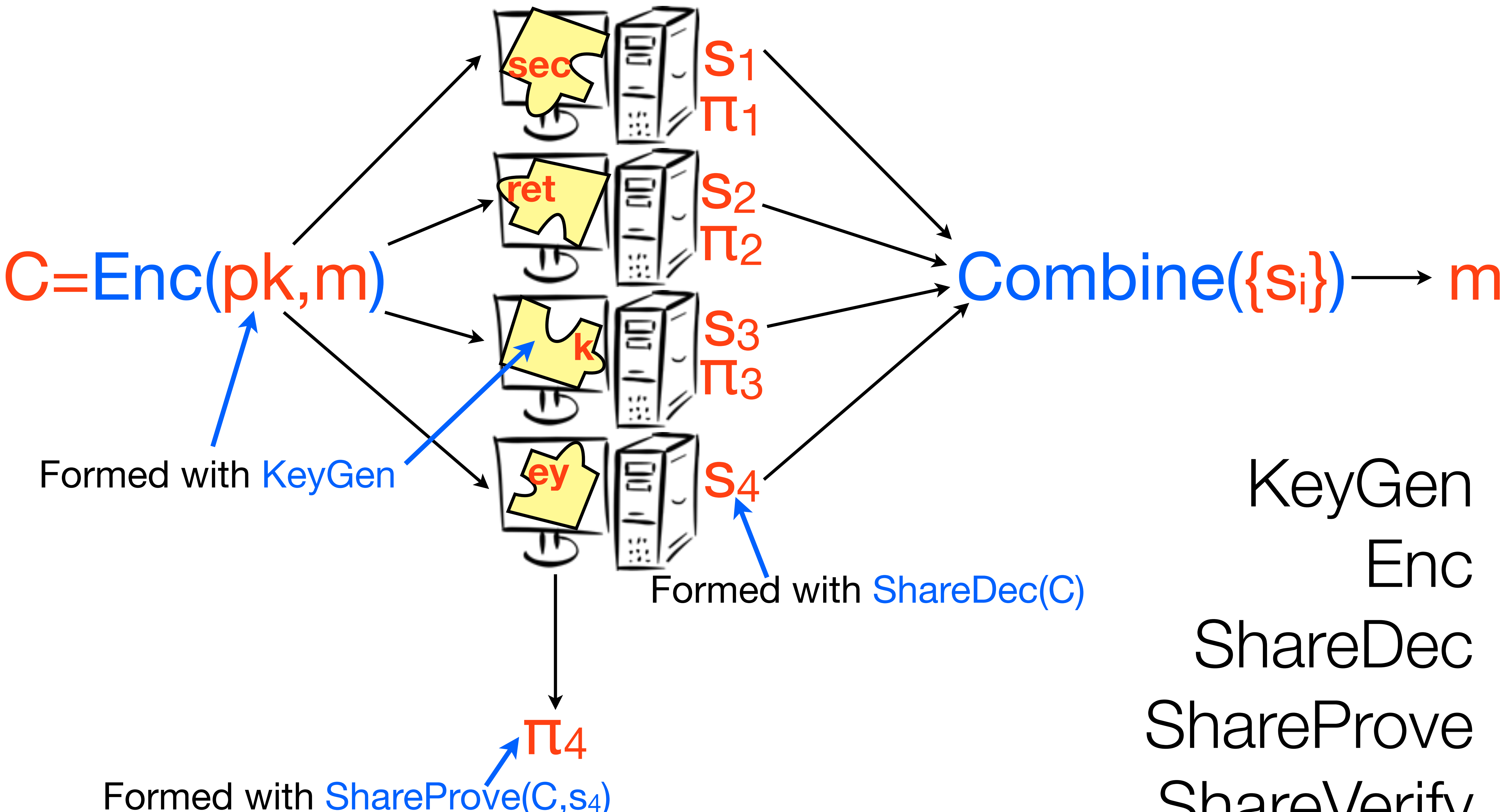
KeyGen
Enc
ShareDec
ShareProve
ShareVerify

Regular verifiable threshold decryption [SG98]



KeyGen
Enc
ShareDec
ShareProve
ShareVerify
Combine

Regular verifiable threshold decryption [SG98]



KeyGen
Enc
ShareDec
ShareProve
ShareVerify
Combine