Divisible e-cash in the standard model

Benoit Libert[1] and Malika Izabachène[2]

[1]UCL, Belgium

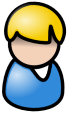
[2]UVSQ, France

April, 8^{th} 2011

E-cash real scenario

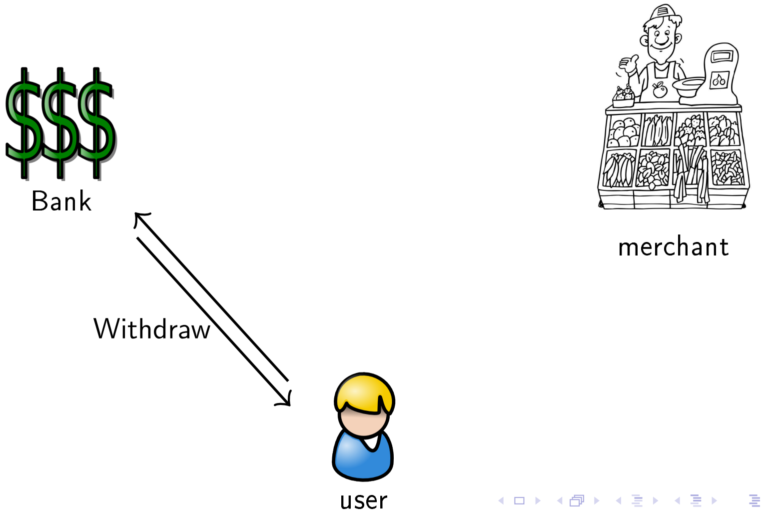
Bank

merchant

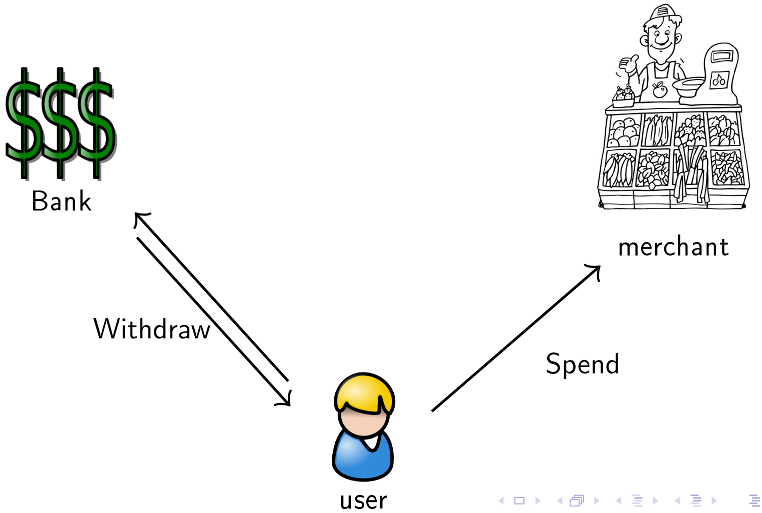


user

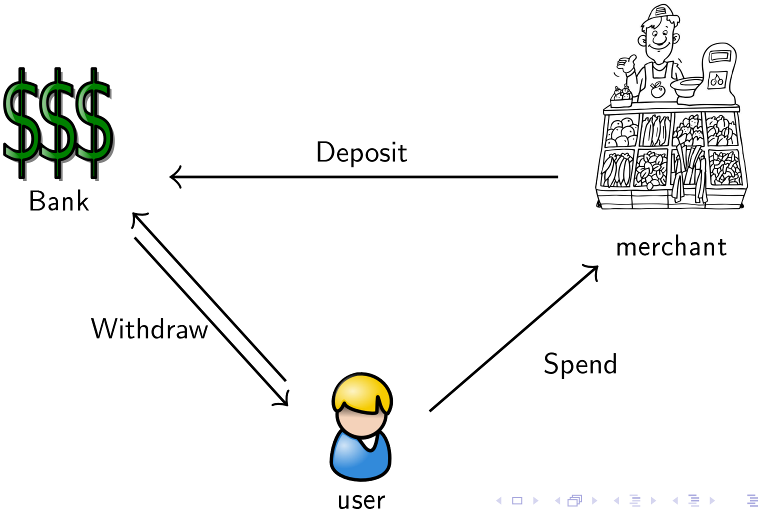
E-cash real scenario



E-cash real scenario



E-cash real scenario



Off-line ecash

Digital analogue of regular paper money

✓ Reduce the amount of interactions: users pay the merchant without the involvement of the bank

Off-line ecash

Digital analogue of regular paper money

✓ Reduce the amount of interactions: users pay the merchant without the involvement of the bank

✓ Users' behaviour can be made more transparent to the bank

Off-line ecash

Digital analogue of regular paper money

✓ Reduce the amount of interactions: users pay the merchant without the involvement of the bank

✓ Users' behaviour can be made more transparent to the bank

✗ But coins can be easily duplicated

Off-line ecash

Digital analogue of regular paper money

✓ Reduce the amount of interactions: users pay the merchant without the involvement of the bank

✓ Users' behaviour can be made more transparent to the bank

✗ But coins can be easily duplicated

✗ Some additional communication cost (to verify validity of a coin)

Off-line ecash

Digital analogue of regular paper money

✓ Reduce the amount of interactions: users pay the merchant without the involvement of the bank

✓ Users' behaviour can be made more transparent to the bank

✗ But coins can be easily duplicated

`Technical challenge 1: How to detect misbehaviours?`

✗ Some additional communication cost (to verify validity of a coin)

Off-line ecash

Digital analogue of regular paper money

✓ Reduce the amount of interactions: users pay the merchant without the involvement of the bank

✓ Users' behaviour can be made more transparent to the bank

✗ But coins can be easily duplicated

`Technical challenge 1: How to detect misbehaviours?`

✗ Some additional communication cost (to verify validity of a coin)

`Technical challenge 2: How to reduce the communication complexity?`

Previous ecash system

- Compact e-cash system [CHL05, BBCKL09]
- Divisible e-cash [Okamoto95, CFT98] (anonymous but not unlinkable)
- Divisible e-cash [NS00] (anonymous and weak unlinkability)

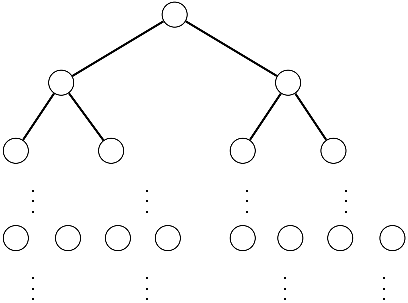
  ✗ requires TTP

  ✗ the merchant and the bank know which part of the coin is spent
- [CG07]: the first truly anonymous Divisible e-cash system
  $\rightsquigarrow$ relies on bounded accumulators and the ROM heuristic

This work: Divisible e-cash in the standard model with short parameters

Outline

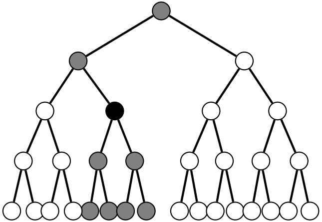
The tree-based approach



Deep	Value
$d = 0$	2^i
$d = 1$	2^{i-1}
$d = 2$	2^{i-2}
$\vdots$	$\vdots$
$d = i - \ell$	2^ℓ
$\vdots$	$\vdots$
$d = i$	1

Divisibility

Impossible to spend an ancestor or a descendant of a spent coin without being detected



Coin valued 2^2 is spent

Security Notions

Basic Properties

Anonymity	No coalition of bank and merchants can distinguish real spendings from simulated ones
Balance	No coalition of users can spend more coins than they withdrew
Identification	Given two fraudulent coins, $\mathcal{B}$ should be able to identify the double-spender
Exculpabiliy	No coalition of merchants and bank can falsely accuse a user from double-spending

Syntactic Definition

Syntactic Definition

- $\mathsf{CashSetup}(\lambda)$: generates params

Syntactic Definition

- CashSetup(λ): generates params
- BankKGen(params): defines $\mathsf{pk}_{\mathcal{B}}, \mathsf{sk}_{\mathcal{B}}$

Syntactic Definition

- $\mathsf{CashSetup}(\lambda)$: generates params
- $\mathsf{BankKGen}(\mathsf{params})$: defines $\mathsf{pk}_{\mathcal{B}}, \mathsf{sk}_{\mathcal{B}}$
- $\mathsf{UserKGen}(\mathsf{params})$: defines $\mathsf{pk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{U}}$

Syntactic Definition

- $\mathsf{CashSetup}(\lambda)$: generates params
- $\mathsf{BankKGen}(\mathsf{params})$: defines $\mathsf{pk}_{\mathcal{B}}, \mathsf{sk}_{\mathcal{B}}$
- $\mathsf{UserKGen}(\mathsf{params})$: defines $\mathsf{pk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{U}}$
- $\mathsf{Withdraw}\left(\mathcal{U}(\mathsf{pk}_{\mathcal{B}}, \mathsf{sk}_{\mathcal{U}}, i), \mathcal{B}(\mathsf{pk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{B}}, i)\right)$: allows $\mathcal{U}$ to obtain a divisible coin of value 2^i added to DB

Syntactic Definition

- $\mathsf{CashSetup}(\lambda)$: generates params
- $\mathsf{BankKGen}(\mathsf{params})$: defines $\mathsf{pk}_{\mathcal{B}}, \mathsf{sk}_{\mathcal{B}}$
- $\mathsf{UserKGen}(\mathsf{params})$: defines $\mathsf{pk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{U}}$
- $\mathsf{Withdraw}\,(\mathcal{U}(\mathsf{pk}_{\mathcal{B}}, \mathsf{sk}_{\mathcal{U}}, i), \mathcal{B}(\mathsf{pk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{B}}, i))$: allows $\mathcal{U}$ to obtain a divisible coin of value 2^i added to DB
- $\mathsf{Spend}(\mathsf{pk}_{\mathcal{B}}, \mathcal{W}, v, \mathsf{pk}_{\mathcal{M}}, \mathsf{info})$: allows $\mathcal{U}$ to spend a $coin = (*, \pi)$ of value v from wallet $\mathcal{W}$ to merchant $\mathsf{pk}_{\mathcal{M}}$

Syntactic Definition

- $\mathsf{CashSetup}(\lambda)$: generates params
- $\mathsf{BankKGen}(\mathsf{params})$: defines $\mathsf{pk}_{\mathcal{B}}, \mathsf{sk}_{\mathcal{B}}$
- $\mathsf{UserKGen}(\mathsf{params})$: defines $\mathsf{pk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{U}}$
- $\mathsf{Withdraw}\left(\mathcal{U}(\mathsf{pk}_{\mathcal{B}}, \mathsf{sk}_{\mathcal{U}}, i), \mathcal{B}(\mathsf{pk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{B}}, i)\right)$: allows $\mathcal{U}$ to obtain a divisible coin of value 2^i added to DB
- $\mathsf{Spend}(\mathsf{pk}_{\mathcal{B}}, \mathcal{W}, v, \mathsf{pk}_{\mathcal{M}}, \mathsf{info})$: allows $\mathcal{U}$ to spend a $coin = (*, \pi)$ of value v from wallet $\mathcal{W}$ to merchant $\mathsf{pk}_{\mathcal{M}}$
- $\mathsf{VerifyCoin}(\mathsf{pk}_{\mathcal{M}}, \mathsf{pk}_{\mathcal{B}}, v, coin)$: verifies π

Syntactic Definition

- CashSetup(λ): generates params
- BankKGen(params): defines $\mathsf{pk}_\mathcal{B}, \mathsf{sk}_\mathcal{B}$
- UserKGen(params): defines $\mathsf{pk}_\mathcal{U}, \mathsf{sk}_\mathcal{U}$
- Withdraw $(\mathcal{U}(\mathsf{pk}_\mathcal{B}, \mathsf{sk}_\mathcal{U}, i), \mathcal{B}(\mathsf{pk}_\mathcal{U}, \mathsf{sk}_\mathcal{B}, i))$: allows $\mathcal{U}$ to obtain a divisible coin of value 2^i added to DB
- Spend($\mathsf{pk}_\mathcal{B}, \mathcal{W}, v, \mathsf{pk}_\mathcal{M}$, info): allows $\mathcal{U}$ to spend a $coin = (*, \pi)$ of value v from wallet $\mathcal{W}$ to merchant $\mathsf{pk}_\mathcal{M}$
- VerifyCoin($\mathsf{pk}_\mathcal{M}, \mathsf{pk}_\mathcal{B}, v, coin$): verifies π
- Deposit($\mathsf{pk}_\mathcal{B}, \mathsf{pk}_\mathcal{M}, v$, DB): allows the bank to detect a cheating attempt from the $\mathcal{U}$ or $\mathcal{M}$. In case of double-spending, returns the two coins c_a and c_b

Syntactic Definition

- CashSetup(λ): generates params
- BankKGen(params): defines $\mathsf{pk}_{\mathcal{B}}, \mathsf{sk}_{\mathcal{B}}$
- UserKGen(params): defines $\mathsf{pk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{U}}$
- Withdraw $(\mathcal{U}(\mathsf{pk}_{\mathcal{B}}, \mathsf{sk}_{\mathcal{U}}, i), \mathcal{B}(\mathsf{pk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{B}}, i))$: allows $\mathcal{U}$ to obtain a divisible coin of value 2^i added to DB
- Spend$(\mathsf{pk}_{\mathcal{B}}, \mathcal{W}, v, \mathsf{pk}_{\mathcal{M}}, \mathsf{info})$: allows $\mathcal{U}$ to spend a $coin = (*, \pi)$ of value v from wallet $\mathcal{W}$ to merchant $\mathsf{pk}_{\mathcal{M}}$
- VerifyCoin$(\mathsf{pk}_{\mathcal{M}}, \mathsf{pk}_{\mathcal{B}}, v, coin)$: verifies π
- Deposit$(\mathsf{pk}_{\mathcal{B}}, \mathsf{pk}_{\mathcal{M}}, v, \mathsf{DB})$: allows the bank to detect a cheating attempt from the $\mathcal{U}$ or $\mathcal{M}$. In case of double-spending, returns the two coins c_a and c_b
- Identify$(\mathsf{pk}_{\mathcal{B}}, c_a, c_b)$: given the two double-spent coins, retrieves the cheating user's public key

Pairings

$\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ groups of prime order p

Cryptographic bilinear maps

Consider $e : \mathbb{G}_1 \times \mathbb{G}_2 \mapsto \mathbb{G}_T$ s.t.

- bilinear: $e(g_1^a, g_2^b) = e(g_1, g_2)^{ab}$
- non-degenerated: $e(g_1, g_2) \neq 1$
- efficiently computable

F-Unforgeable Signature (1/2)

- SigSetup(λ): outputs params
- SigKG(params, n): outputs pk and sk for block of size n
- Sign$(\text{sk}, \mathbf{m})$: outputs a signature σ on block $\mathbf{m}$
- Verify$(\text{pk}, \mathbf{m}, \sigma)$: verifies whether σ is a valid signature on $\mathbf{m}$

F-Unforgeable Signature (1/2)

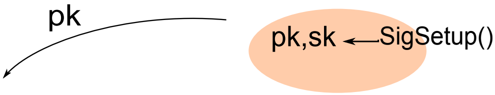
- SigSetup(λ): outputs params
- SigKG(params, n): outputs pk and sk for block of size n
- Sign(sk, $\mathbf{m}$): outputs a signature σ on block $\mathbf{m}$
- Verify(pk, $\mathbf{m}, \sigma$): verifies whether σ is a valid signature on $\mathbf{m}$

F-Unforgeability

F-Unforgeable Signature (1/2)

- SigSetup(λ): outputs params
- SigKG(params, n): outputs pk and sk for block of size n
- Sign(sk, $\mathbf{m}$): outputs a signature σ on block $\mathbf{m}$
- Verify(pk, $\mathbf{m}, \sigma$): verifies whether σ is a valid signature on $\mathbf{m}$

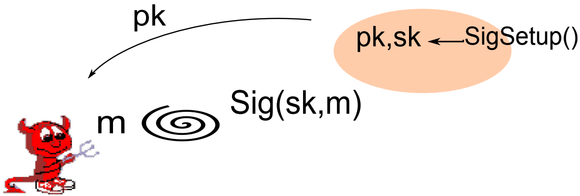
F-Unforgeability



F-Unforgeable Signature (1/2)

- SigSetup(λ): outputs params
- SigKG(params, n): outputs pk and sk for block of size n
- Sign(sk, $\mathbf{m}$): outputs a signature σ on block $\mathbf{m}$
- Verify(pk, $\mathbf{m}, \sigma$): verifies whether σ is a valid signature on $\mathbf{m}$

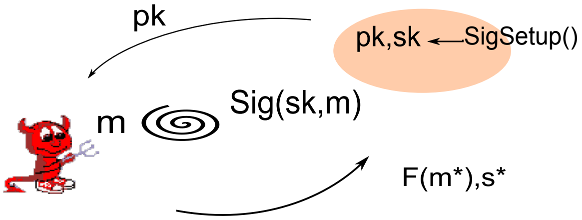
F-Unforgeability



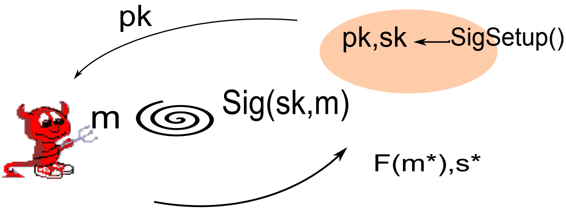
F-Unforgeable Signature (1/2)

- SigSetup(λ): outputs params
- SigKG(params, n): outputs pk and sk for block of size n
- Sign(sk, $\mathbf{m}$): outputs a signature σ on block $\mathbf{m}$
- Verify(pk, $\mathbf{m}, \sigma$): verifies whether σ is a valid signature on $\mathbf{m}$

F-Unforgeability



F-Unforgeable Signature (2/2)



$\mathcal{A}$ outputs $(F(\mathbf{m}*), \mathbf{s}*)$ and wins if:

$$\mathsf{Verify}(\mathsf{pk}, \mathbf{m}*, \mathbf{s}*) \text{ and } \mathbf{s}* \notin \{\mathsf{Sign}(\mathsf{sk}, \mathbf{m}_1), \cdots, \mathsf{Sign}(\mathsf{sk}, \mathbf{m}_{q_\sigma})\}$$

Sign and Prove

- $\mathsf{SigProve}(\mathsf{params}, \mathsf{pk}, \sigma, \mathbf{m})$: NI proof of possession of a valid F-unforgeable signature on $\mathbf{m}$:

$$\mathbf{C_m} + \mathtt{NIZK}\{\sigma \mid \mathsf{Verify}(\mathsf{pk}, \mathbf{m}, \sigma) = 1\}$$

- $\mathsf{SigIssue}(\mathsf{sk}, \mathbf{C_m}) \leftrightarrow \mathsf{SigObtain}(\mathsf{pk}, \mathbf{C_m}, \mathbf{open})$: allows $\mathcal{U}$ to obtain a signature on a committed vector $\mathbf{m}$

Groth Sahai proof system [GS07]

NIZK proofs for pairing product equations (PPE):

$$\prod_{j=1}^{n} e(A_j, Y_j) \prod_{j=1}^{n} e(X_i, B_i) \prod_{i=1}^{m} \prod_{j=1}^{n} e(X_i, Y_j)^{\gamma_{i,j}} = t_T,$$

where $*$ are variables and t_T, the A_j's and B_i's are constants

General strategy: Commit on variables and Prove statements NI

[GS07], [GS08] hardly compatible with Groth Sahai toolbox

`Technical challenge:` simulate NIZK proofs for PPE

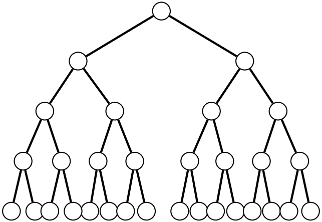
Construction Overview (1/4)

- BankKGen(params): run SigSetup$(\lambda, 2)$ to obtain $\mathsf{pk}_{\mathcal{B}}, \mathsf{sk}_{\mathcal{B}}$
  UserKGen(params): define $\mathsf{pk}_{\mathcal{U}} = e(g,h)^{\mathsf{sk}_{\mathcal{U}}}$, with $\mathsf{sk}_{\mathcal{U}} \overset{\mathcal{R}}{\leftarrow} \mathbb{Z}_p$
- Withdraw$(\mathcal{U}(), \mathcal{B}())$:

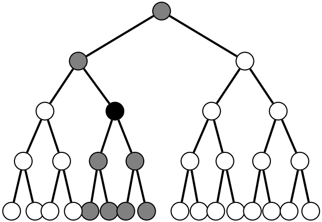
$$\begin{array}{lcl}
\mathcal{U} & & \mathcal{B} \\
s' \overset{\mathcal{R}}{\leftarrow} \mathbb{Z}_p & \xrightarrow{\mathbf{C}'_{\mathbf{s}}, \pi_{\mathbf{open}}} & s'' \\
 & \xleftarrow{s''} & \\
s = s' + s'' & \xrightarrow{\mathbf{C}_{\mathbf{s}}} & \\
\mathcal{W} = (s, \mathsf{sk}_{\mathcal{U}}, \sigma, \mathsf{state}) & \overset{\mathsf{SigObtain}, \mathsf{SigIssue}}{\longleftrightarrow} & \mathsf{update}(*)
\end{array}$$

Construction Overview (2/4)

Spend anonymously in the tree a coin of value $v = 2^2$ in $\mathcal{W} = (s, t, \mathsf{sk}_{\mathcal{U}}, \sigma, \mathsf{state})$ to $\mathcal{M}$ identified by info



No coin is spent



One coin is spent

Figure: Binary tree for spending one coin in a sub-wallet of 2^4 coins

Construction Overview (3/4)

1. Define path: (x_0, x_1, x_2) s.t. $x_{j+1} = 2x_j + b_j$
   Compute $S = h^s$
2. Compute $\pi_1 \leftarrow \mathsf{SigProve}(\mathsf{pk}, (s, \mathsf{sk}_{\mathcal{U}}), \sigma)$
3. Commit to the path and prove well-formedness
4. Compute coin's serial number $Y_j* = \mathsf{PRF}_s(x_j)$ for $j = 1, 2$
5. Prove everything is done consistently

Construction Overview (4/4)

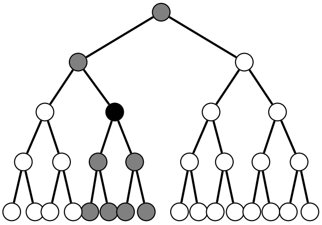
Double-spending Detection:

Add $T_{j,1} = h^{d_{j,1}}, T_{j,2} = e(Y_j, T_{j,1})$, for $j = 1, 2$ and

Use Y_2* to check for entry s in DB with $i = 2$:

- if $\ell_s = 3 > 2$ and if $T^*_{3,2} == e(Y_1, T^*_{3,1})$
- if $\ell_s = 1 < 2$ and if $T_{2,2} == e(Y^*_2, T_{2,1})$
- if $\ell_s = 2 = \ell$ and if $Y_2 == Y^*_2$

$\cdots \mathcal{U}$ is guilty



Double-spender Identification: similar to [CHL05]

Trickier: add an additional seed t and embedd $\mathsf{pk}_{\mathcal{U}}$ in each node

Conclusion

Improve efficiency of the Spend algorithm:

- Other data structure that enables more efficient coin diversification and coin derivation?
- Guarantee more efficient spending to prove statements about each node in less than $|\text{path}|$ proofs?

Improve efficiency of the Deposit algorithm