A Proper Security Level for Postcompromise Secure Messaging

F Betül Durak and Serge Vaudenay

EPFL

New logo!
1. Ratcheting
2. BARK
3. ARCAD
4. Comparison of Protocols
1. Ratcheting
2. BARK
3. ARCAD
4. Comparison of Protocols
End-To-End Secure IM
Secure Bidirectional Communication

Init

$\text{st}_A(0) \rightarrow \text{Send} \rightarrow \text{Receive} \rightarrow \text{st}_A(1)$

$\text{pt}_1 \rightarrow \text{Send} \rightarrow \text{Receive} \rightarrow \text{pt}_1$

$\text{st}_A(1) \rightarrow \text{ct}_1 \rightarrow \text{st}_B(1)$

$\text{st}_B(0) \rightarrow \text{Receive} \rightarrow \text{Send} \rightarrow \text{pt}_2$

$\text{pt}_2 \rightarrow \text{Receive} \rightarrow \text{Send} \rightarrow \text{st}_B(2)$

$\text{st}_A(2) \rightarrow \text{ct}_2 \rightarrow \text{st}_B(2)$

$\text{st}_B(1) \rightarrow \text{Receive} \rightarrow \text{Send} \rightarrow \text{pt}_3$

$\text{pt}_3 \rightarrow \text{Receive} \rightarrow \text{Send} \rightarrow \text{st}_B(3)$

$\text{st}_A(3) \rightarrow \text{ct}_3 \rightarrow \text{st}_B(3)$

$\text{pt}_4 \rightarrow \text{Send} \rightarrow \text{Receive} \rightarrow \text{pt}_4$

$\text{ct}_4 \rightarrow \text{st}_A(4)$
Aim: Forward Secrecy

\[\text{Init}\]$^\text{\$}$

\[st_A(0)\]  \[st_B(0)\]

\[\text{Send}\]  \[\text{Receive}\]

\[\text{pt}_1\]  \[\text{pt}_1\]

\[\text{Receive}\]  \[\text{Send}\]

\[\text{pt}_2\]  \[\text{pt}_2\]

\[\text{Receive}\]  \[\text{Send}\]

\[\text{pt}_3\]  \[\text{pt}_3\]

\[\text{Send}\]  \[\text{Receive}\]

\[\text{pt}_4\]  \[\text{pt}_4\]

...
Aim: + Post-Compromise Security

1. **Init**
   - Send
   - Receive
   - st_A(0)
   - st_B(0)

2. **pt_1**
   - Send
   - Receive
   - st_A(1)
   - ct_1
   - pt_1

3. **pt_2**
   - Receive
   - Send
   - st_A(2)
   - ct_2
   - pt_2

4. **pt_3**
   - Receive
   - Send
   - st_A(3)
   - ct_3
   - HEALED!

5. **pt_4**
   - Send
   - Receive
   - HEALED!
   - ct_4
   - pt_4

6. **st_B(1)**
   - Send
   - Receive
   - pt_1

7. **st_B(2)**
   - Send
   - Receive
   - pt_2

8. **st_B(3)**
   - Send
   - Receive
   - HEALED!

9. **ct_1, ct_2, ct_3, ct_4**
    - Happy face
    - Sad face
    - Green face
    - Happy face

HEALED!
By the Way: Asynchronous + Random Role
Ratchet

State update

- in a one-way manner (for *forward security*)
- using randomness (for *post-compromise security*)
Bellare, Singh, Asha, Jaeger, Nyayapati, Stepanovs
Ratcheted Encryption and Key Exchange: The Security of Messaging

- unidirectional
- no receiver leakage allowed
- complicated definitions
Poettering, Rösler
Ratcheted Key Exchange, Revisited

Jaeger, Stepanovs
Optimal Channel Security Against Fine-Grained State Compromise: The Safety of Messaging

- both need key update primitives (HIBE, random oracles, ...)
- complicated definitions
with immediate decryption

Alwen, Coretti, Dodis
The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol

Jost, Maurer, Mularczyk
Efficient Ratcheting: Almost-Optimal Guarantees for Secure Messaging

near-optimal security but better complexity — still high
Our Results

Durak, Vaudenay
Bidirectional Asynchronous Ratcheted Key Agreement with Linear Complexity
Eprint 2018/889

Caforio, Durak, Vaudenay
On-Demand Ratcheting with Security Awareness
Soon on Eprint
1 Ratcheting
2 BARK
3 ARCAD
4 Comparison of Protocols
BARK
Bidirectional Asynchronous Ratcheted Key Agreement
**Interface**

- **Setup**($1^λ$) $\rightarrow$ pp (the common public parameters)
- **Gen**(pp) $\rightarrow$ (sk, pk) (key pair of a participant)
- **Init**(pp, sk$_P$, pk$_P$, $P$) $\rightarrow$ st$_P$ (initial state)
- **Send**(st$_P$) $\rightarrow$ (st'$_P$, ct, $k$) (like KEM.Enc)
- **Receive**(st$_P$, ct) $\rightarrow$ (acc, st'$_P$, $k$) (like KEM.Dec)

**Initall**(pp):
1: Gen(pp) $\rightarrow$ (sk$_A$, pk$_A$)
2: Gen(pp) $\rightarrow$ (sk$_B$, pk$_B$)
3: st$_A$ $\leftarrow$ Init(pp, sk$_A$, pk$_B$, 0)

4: st$_B$ $\leftarrow$ Init(pp, sk$_B$, pk$_A$, 1)
5: z $\leftarrow$ (pp, pk$_A$, pk$_B$)
6: **return** (st$_A$, st$_B$, z)

we must specify what to give to the adversary
Correctness

For all sequence $\text{sched}$, $\Pr[\text{Correctness}(\text{sched}) \rightarrow 1] = 0$

**Oracle** $\text{RATCH}(P, \text{send})$
1: $(st_P, ct_P, k_P) \leftarrow \text{Send}(st_P)$
2: append $k_P$ to $\text{sent}_{key}^P$
3: $\text{return } ct_P$

**Oracle** $\text{RATCH}(P, \text{rec}, ct)$
4: $(\text{acc}, st'_P, k'_P) \leftarrow \text{Receive}(st_P, ct)$
5: if acc then
6: $st_P \leftarrow st'_P$
7: $k_P \leftarrow k'_P$
8: append $k_P$ to $\text{received}_{key}^P$
9: end if
10: $\text{return } \text{acc}$

**Game** $\text{Correctness}(\text{sched})$
1: $\text{Setup} \xrightarrow{\$} pp$
2: $\text{Initall}(pp) \xrightarrow{\$} (st_A, st_B, z)$
3: initialize two FIFO incoming$_P, P \in \{A, B\}$
4: $i \leftarrow 0$
5: loop
6: $i \leftarrow i + 1$
7: $(P, \text{role}) \leftarrow \text{sched}_i$
8: if role $= \text{send}$ then
9: $ct \leftarrow \text{RATCH}(P, \text{send})$
10: push $ct$ to incoming$_P$
11: else
12: if incoming$_P$ is empty then return 0
13: pull $ct$ from incoming$_P$
14: $\text{acc} \leftarrow \text{RATCH}(P, \text{rec}, ct)$
15: if acc $= \text{false}$ then return 1
16: end if
17: if received$_A$ not prefix of sent$_B$ then return 1
18: if received$_B$ not prefix of sent$_A$ then return 1
19: end loop
KIND Security

For all ppt $\mathcal{A}$, $\left| \Pr\left[ \text{KIND}^A_{0,C_{\text{clean}}} \rightarrow 1 \right] - \Pr\left[ \text{KIND}^A_{1,C_{\text{clean}}} \rightarrow 1 \right] \right| = \text{negl}$

**Game** $\text{KIND}^A_{b,C_{\text{clean}}}$

1: Setup $\xrightarrow{\$} \text{pp}$
2: $\text{Initall}(\text{pp}) \xrightarrow{\$} (\text{st}_A, \text{st}_B, z)$
3: $b' \leftarrow A^{\text{RATCH}, \text{EXP}_{\text{st}}, \text{EXP}_{\text{key}}, \text{TEST}(z)}$
4: if $\neg C_{\text{clean}}$ then return ⊥
5: return $b'$

**Oracle** $\text{TEST}(P)$

1: if $b = 1$ then
2: return $k_P$
3: else
4: return random $\{0, 1\}^{\lfloor k_P \rfloor}$
5: end if

**Oracle** $\text{EXP}_{\text{key}}(P)$

1: return $k_P$

**Oracle** $\text{EXP}_{\text{st}}(P)$

1: return $\text{st}_P$

- the EXP oracles can be used for trivial attacks without forgeries
- not easy to identify trivial attacks in the case of forgeries

exclude trivial attacks
A Few Technical Notions: Matching Status

$P$ in **matching status** at time $t \iff \exists t', t' \in [t, \infty)$ such that:

\[
\begin{align*}
\text{received}^P_{msg}(t) &= \text{sent}^P_{msg}(\tilde{t}) \\
\text{received}^{\overline{P}}_{msg}(\tilde{t}) &= \text{sent}^{\overline{P}}_{msg}(t')
\end{align*}
\]

**Property**

If $P$ in matching status at time $t$...

- $\overline{P}$ in matching status at time $\tilde{t}$
- $P$ in matching status before $t$
- $k_P(t) = k_{\overline{P}}(\tilde{t})$
$k_P(t)$ directly leaks if we are in one of those configurations:
A Few Technical Notions: Indirect Leakage

\( k_P(t) \) indirectly leaks if \( P \) is in matching status at time \( t \) and

- either the corresponding \( k_{\overline{P}}(\overline{t}) \) directly leaks
- or we are in this configuration:
A Few Cleanness Notions

- $C_{\text{leak}}$: the tested $k_{P_{\text{test}}}$ leaks neither directly nor indirectly mandatory: we must have this clause in $C_{\text{clean}}$
- $C_{\text{trivial forge}}^{P_{\text{test}}}$: $P_{\text{test}}$ had no trivial forgery before TEST
- $C_{\text{trivial forge}}^{A,B}$: neither $A$ nor $B$ had a trivial forgery before seeing the ct making the tested $k_{P_{\text{test}}}$

$$(C_{\text{leak}} \land C_{\text{trivial forge}}^{P_{\text{test}}})$$-KIND security $\quad \leftarrow$ PR18 and JS18 (optimal)

$$\downarrow$$

$$(C_{\text{leak}} \land C_{\text{trivial forge}}^{A,B})$$-KIND security $\quad \leftarrow$ BARK (sub-optimal)
Why Optimal Security?

- seems to somehow imply HIBE...
- how would $P_{\text{test}}$ know he accepted no forgery?
- by making sure that he can still communicate with $\overline{P}_{\text{test}}$
- $\implies$ happy with
  - $C_{\text{ratchet}}$: the ct making the tested $k_{P_{\text{test}}}$ initiated a round trip
  - $P \xrightarrow{\text{ct}} \overline{P} \xrightarrow{\text{ct'}} P$

\[
(\neg C_{\text{leak}} \land C_{\text{trivial forge}}^{P_{\text{test}}})\text{-KIND security} \quad \xleftarrow{\text{PR18 and JS18 (optimal)}}
\]

\[
(\neg C_{\text{leak}} \land C_{\text{trivial forge}}^{A,B})\text{-KIND security} \quad \xleftarrow{\text{BARK (sub-optimal)}}
\]

\[
(\neg C_{\text{leak}} \land C_{\text{ratchet}})\text{-KIND security} \quad \xleftarrow{\text{we are happy here}}
\]
A Naive Signcryption

- encrypt and authenticate pt
- can authenticate ad at the same time
- sender state \( st_S = (sk_S, pk_R) \)
- receiver state \( st_R = (sk_R, pk_S) \)
Signcryption → Multiple-Key Signcryption (Onion)

The diagram illustrates the process of signcryption with multiple keys. The notation used is:

- **Enc**: Encryption
- **Dec**: Decryption
- **SC**: Signcryption
- **pt**: Plaintext
- **ad**: Adversary
- **st**: Signer
- **ct**: Ciphertext

The process involves:

1. **Enc** plaintext to **SC.Enc** and then to **SC.Enc** again.
2. XOR with an auxiliary value ($\odot$) to get **SC.Enc**.
3. **Dec** **SC.Enc** to get **Enc**.
4. XOR with the auxiliary value to get **pt**.

This process is repeated with different keys for additional security layers, resembling an onion structure.
M-Key Signcryption → Unidirectional Ratchet

- generate the next send state while sending
- transmit the next receive state while sending
- flush all accumulated states
Unidirectional Ratchet → Bidirectional Ratchet

- Generate a state for *replying* at sending
- Accumulate receive states at sending
authenticate the chain of sent messages while sending
Our Protocol: BARK (Setup, Gen, Init)

**BARK.Setup**
1: \( H.\text{Gen}(1^\lambda) \xrightarrow{\$} hk \)
2: \textbf{return} \( hk \)

**BARK.Gen(hk)**
1: \( \text{SC.\text{Gen}}_S \xrightarrow{\$} (sk_S, pk_S) \)
2: \( \text{SC.\text{Gen}}_R \xrightarrow{\$} (sk_R, pk_R) \)
3: \( sk \leftarrow (sk_S, sk_R) \)
4: \( pk \leftarrow (pk_S, pk_R) \)
5: \textbf{return} \( (sk, pk) \)

**BARK.Init(hk, sk_{P}, pk_{P}, P)**
1: parse \( sk_{P} = (sk_S, sk_R) \)
2: parse \( pk_{P} = (pk_S, pk_R) \)
3: \( st_{P}^{\text{send}} \leftarrow (sk_S, pk_R) \)
4: \( st_{P}^{\text{rec}} \leftarrow (sk_R, pk_S) \)
5: \( st_{P} \leftarrow (hk, (st_{A}^{\text{send}}, (st_{A}^{\text{rec}}, \perp, \perp)) \)
6: \textbf{return} \( st_{P} \)

**st** = \[
\langle \text{hash key} \rangle \\
\langle \text{list of send states} \rangle \\
\langle \text{list of receive states} \rangle \\
\langle \text{sent hash} \rangle \\
\langle \text{receive hash} \rangle 
\]

SV 2019

arcad

WSM 2019
Our Protocol: BARK (Send)

BARK.Send(st_P)
1: parse st_P = (hk, (st_{send,1}^P, ... , st_{send,u}^P), (st_{rec,1}^P, ... , st_{rec,v}^P), Hsent, Hreceived)
2: pick k
3: onion.Init(1^k) \xrightarrow{\$} (st_{Snew}^P, st_{rec,v+1}^P) \quad \triangleright \text{append a new receive state to the st_{rec} list}
4: pt \leftarrow (st_{Snew}^P, k) \quad \triangleright \text{then, st}_{Snew}^P \text{ is erased to avoid leaking}
5: take the smallest i s.t. st_{send,i}^P \neq \bot \quad \triangleright i = u - n \text{ if we had } n \text{ Receive since the last Send}
6: onion.Send(hk, st_{send,i}^P, ... , st_{send,u}^P, Hsent, pt) \xrightarrow{\$} (st_{send,u}^P, ct) \quad \triangleright \text{update st}_{send,u}^P
7: st_{send,i}^P, ... , st_{send,u-1}^P \leftarrow \bot \quad \triangleright \text{flush the send state list: only st}_{send,u}^P \text{ remains}
8: ct \leftarrow (Hsent, ct) \quad \triangleright \text{the onion has } u - i + 1 = n + 1 \text{ layers}
9: Hsent' \leftarrow H.Eval(hk, ct)
10: st_P' \leftarrow (hk, (st_{send,1}^P, ... , st_{send,u}^P), (st_{rec,1}^P, ... , st_{rec,v+1}^P), Hsent', Hreceived)
11: return (st_P', ct)

- create a new onion channel for return
- add st_{rec} in list of receive states
- concatenate st_{Snew} to key
- onion.encrypt with all send states
- authenticate sent hash and the onion depth
Our Protocol: BARK (Receive)

BARK.Receive(st_P, ct)
1: parse \( st_P = (hk, (st^{send,1}_P, \ldots, st^{send,u}_P), (st^{rec,1}_P, \ldots, st^{rec,v}_P), H_{sent}, H_{received}) \)
2: parse \( ct = (h, ct) \) \( \triangleright \) the onion has \( n + 1 \) layers
3: set \( n+1 \) to the number of components in \( ct \)
4: if \( h \neq H_{received} \) then return \((false, st_P, \perp)\)
5: set \( i \) to the smallest index such that \( st^{rec,i}_P \neq \perp \)
6: if \( i + n > v \) then return \((false, st_P, \perp)\)
7: onion.Receive(hk, \( st^{rec,i}_P, \ldots, st^{rec,i+n-1}_P, H_{received}, ct \) \( \rightarrow (acc, st^{rec,i+n-1}_P, pt) \)
8: if \( acc = false \) then return \((false, st_P, \perp)\)
9: parse \( pt = (st^{send,u+1}_P, k) \) \( \triangleright \) a new send state is added in the list
10: \( st^{send,i}_P, \ldots, st^{send,i+n-2}_P \leftarrow \perp \) \( \triangleright n \) entries of \( st^{rec}_P \) were erased
11: \( st^{rec,i+n-1}_P \leftarrow st^{rec,i+n-1}_P \) \( \triangleright \) update \( st^{rec}_P \) stage 2: update \( st^{rec,i+n}_P \)
12: \( H_{received}' \leftarrow H.Eval(hk, ct) \)
13: \( st_P' \leftarrow (hk, (st^{send,1}_P, \ldots, st^{send,u+1}_P), (st^{rec,1}_P, \ldots, st^{rec,v}_P), H_{sent}, H_{received}') \)
14: return \((acc, st_P', k)\)

- onion.decrypt with receive states (onion encryption)
- authenticate received hash and the onion depth
- remove all but the last used receive states
- get \( st^{send} \) and add in list
### Example

<table>
<thead>
<tr>
<th>Alice send states</th>
<th>receive states</th>
<th>messages</th>
<th>Bob send states</th>
<th>receive states</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{send } k^A_1)</td>
<td>(\text{receive } k^B_1)</td>
<td>([\text{st}^A_1, k^A_1]<em>{\text{st}</em>{1,0}})</td>
<td>(\text{send } k^B_1)</td>
<td>(\text{receive } k^A_1)</td>
</tr>
<tr>
<td>(\text{send } k^A_2)</td>
<td>(\text{receive } k^B_2)</td>
<td>([\text{st}^A_1, k^A_2]<em>{\text{st}</em>{1,0,\text{st}<em>{2,0}}\text{st}</em>{3,0}})</td>
<td>(\text{send } k^B_2)</td>
<td>(\text{receive } k^A_2)</td>
</tr>
<tr>
<td>(\text{receive } k^B_3)</td>
<td>(\text{send } k^A_3)</td>
<td>([\text{st}^A_1, k^A_3]<em>{\text{st}</em>{1,0,\text{st}<em>{2,0}}\text{st}</em>{3,0}})</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FORGE Security

For all ppt $\mathcal{A}$, $\Pr[\text{FORGE}^\mathcal{A}\rightarrow 1] = \text{negl}$

**Game FORGE$^\mathcal{A}$**

1. Setup $\xrightarrow{\$} pp$
2. $\text{InitAll}(pp) \xrightarrow{\$} (st_A, st_B, z)$
3. $(P, ct) \leftarrow A^{\text{RATCH, EXP}_\text{st, EXP}_\text{key}}(z)$
4. if there is a participant NOT in a matching status then return 0
5. $\text{RATCH}(P, \text{rec}, ct) \rightarrow \text{acc}$
6. if acc = false then return 0
7. if $P$ is in a matching status then return 0
8. if $ct$ is a trivial forgery for $P$ then return 0
9. return 1

This notion is interesting to have in order to reduce exclusion of forgeries to exclusion of trivial forgeries in KIND security:

$$(C_{\text{leak}} \land C_{\text{forge}}^*)\text{-KIND security} \xrightarrow{(+\text{FORGE})} (C_{\text{leak}} \land C_{\text{trivial forge}}^*)\text{-KIND security}$$
RECOVER Security
For all ppt $\mathcal{A}$, $\Pr[\text{RECOVER}^\mathcal{A} \rightarrow 1] = \text{negl}$

**Game RECOVER$^\mathcal{A}$**
1: Setup $\xrightarrow{\$} pp$
2: Initall$(pp) \xrightarrow{\$} (st_A, st_B, z)$
3: set all lists to $\emptyset$
4: $P \leftarrow \mathcal{A}^{\text{RATCH, \text{EXP}_{st}, \text{EXP}_{key}}(z)}$
5: if we can parse as follows then return 1

$$
\text{sent}_p^b_{\text{msg}} = ([\text{seq}_2], ct, [\text{seq}_3])
\downarrow
\uparrow
\text{received}_p^b_{\text{msg}} = ([\text{seq}_1], ct)
$$
6: return 0

This notion is interesting to have in order to make sure that a round trip communication between honest participants implies no forgery.

$$(C_{\text{leak}} \land C_{A,B}^{\text{trivial forge}})\text{-KIND security} \xrightarrow{(+\text{RECOVER})} (C_{\text{leak}} \land C_{\text{ratchet}})\text{-KIND security}$$
Security of BARK

Theorem

If

- $H$ is collision-resistant,
- Sign is EF-OTCPA-secure,
- PKC is IND-CCA-secure,
- Sym is IND-OTCCA-secure,

then BARK is

- RECOVER-secure,
- FORGE-secure, and
- KIND-secure for cleanness $C_{\text{leak}} \land C_{\text{trivial forge}}^{A,B}$.
1. Ratcheting

2. BARK

3. ARCAD

4. Comparison of Protocols
new interface for Send:

\[ \text{Send}(st, \text{ad}, pt) \rightarrow st', ct \]

encrypt pt and authenticate ad at the same time

new interface for Receive:

\[ \text{Receive}(st, \text{ad}, ct) \rightarrow \text{acc}, st', pt \]
liteARCAD: Our Symmetric Protocol

- same as previous protocol with AE instead of SC
- much faster
- no post-compromise security
- still forward security
On-Demand Ratcheting

- use a flag in ad denoted by ad.flag
  - ad.flag = true: ratchet
  - ad.flag = false: live with symmetric crypto

- hybrid security notion...
  - adapt BARK as ARCAD_{DV} for ratchet
  - use liteARCAD for symmetric crypto
Hybrid Ratcheting

\[
\begin{align*}
\begin{pmatrix} s_t^{1_{\text{main}}} \end{pmatrix} & \rightarrow \text{main} \\
\begin{pmatrix} s_t^{2_{\text{main}}} \\ s_t^{2,1_{\text{sub}}} \end{pmatrix} & \rightarrow \text{main} \\
\begin{pmatrix} s_t^{3_{\text{main}}} \\ s_t^{3,1_{\text{sub}}} \end{pmatrix} & \rightarrow \text{main} \\
& \rightarrow \ldots
\end{align*}
\]
Hybrid Ratcheting: Results

- combining $\text{ARCAD}_{DV} + \text{liteARCAD}$, we obtain the best performances if we scarcely ratchet
- privacy is preserved (with hybrid cleanness...)
- unforgeability degrades a bit
- a final protocol transformation restores unforgeability
Security Awareness

- **r-RECOVER security**: cannot *receive* any genuine message after receiving a forgery
- **s-RECOVER security**: cannot *send* any genuine message after receiving a forgery
- **acknowledgement extractor**: each message carries and ACK of received messages
- **cleanliness extractor**: can figure out which message remains private from the history of queries

→ achieved with hybrid ARCAD\textsubscript{DV} + liteARCAD
Ratcheting

BARK

ARCAD

Comparison of Protocols
Implementations

ARCAD\textsubscript{DV} uses ECDSA and ECIES.
liteARCAD uses AES-GCM.
PR18 uses Gentry-Silverberg HIBE and ECDSA.
JS18 uses Gentry-Silverberg HIBE and Bellare-Miner forward-secure signature.
ACD19 uses ECDH and AES-GCM
JMM19 uses ECDSA and ECIES

Acknowledgement: implementations by Andrea Caforio

https://github.com/qantik/ratcheted
Performance

Runtime

Total amount of time (log scale) to send $n$ messages in alternating directions

![Graph showing alternating runtime](image)
Performance

Runtime

Total amount of time (log scale) to send $n$ messages

Unidirectional

Def. Unidirectional
Performance

State Size

Maximal state size (log scale) to send $n$ messages

Alternating

[Def.] Unidirectional

![Graph showing maximal state size](image-url)
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>PR18</th>
<th>JS18</th>
<th>BARK</th>
<th>JMM19</th>
<th>ACD19-PK</th>
<th>ARCAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security</td>
<td>optimal</td>
<td>optimal</td>
<td>sub-optimal</td>
<td>near-optimal</td>
<td>id-optimal</td>
<td>pragmatic</td>
</tr>
<tr>
<td>Complexity</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Corrupt coins resilience</td>
<td>no</td>
<td>pre-send $\Rightarrow$ exposure</td>
<td>no</td>
<td>post-send $\Rightarrow$ exposure</td>
<td>chosen coins $\Rightarrow$ exposure</td>
<td>no</td>
</tr>
<tr>
<td>Plain model</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>PKC or less</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Immediate decryption</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>r-RECOVER security</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>s-RECOVER security</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>ack. extractor</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cleanliness extractor</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

- **Security**: optimal $>$ near-optimal $>$ sub-optimal $>$ pragmatic $>$ id-optimal
- **Complexity to send** $n$ messages in total
- **Plain model**: some need random oracles
- **PKC or less**: some need HIBE
Conclusion

- better understanding on ratcheting security
- ratcheting security can be efficient
- new notions: on-demand ratcheting, security awareness