IS&C

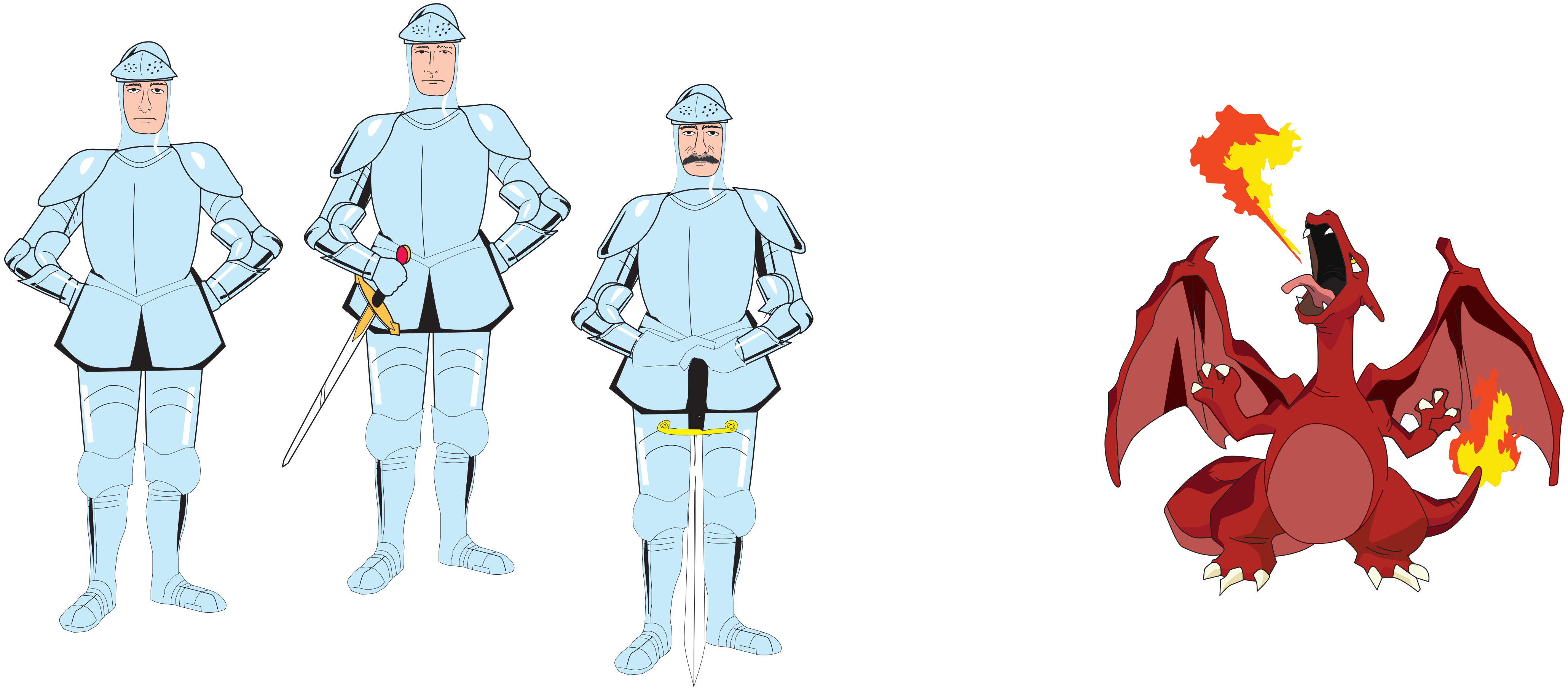
UNIVERSITÄT DES SAARLANDES

Achieving Security Despite Compromise Using Zero-knowledge

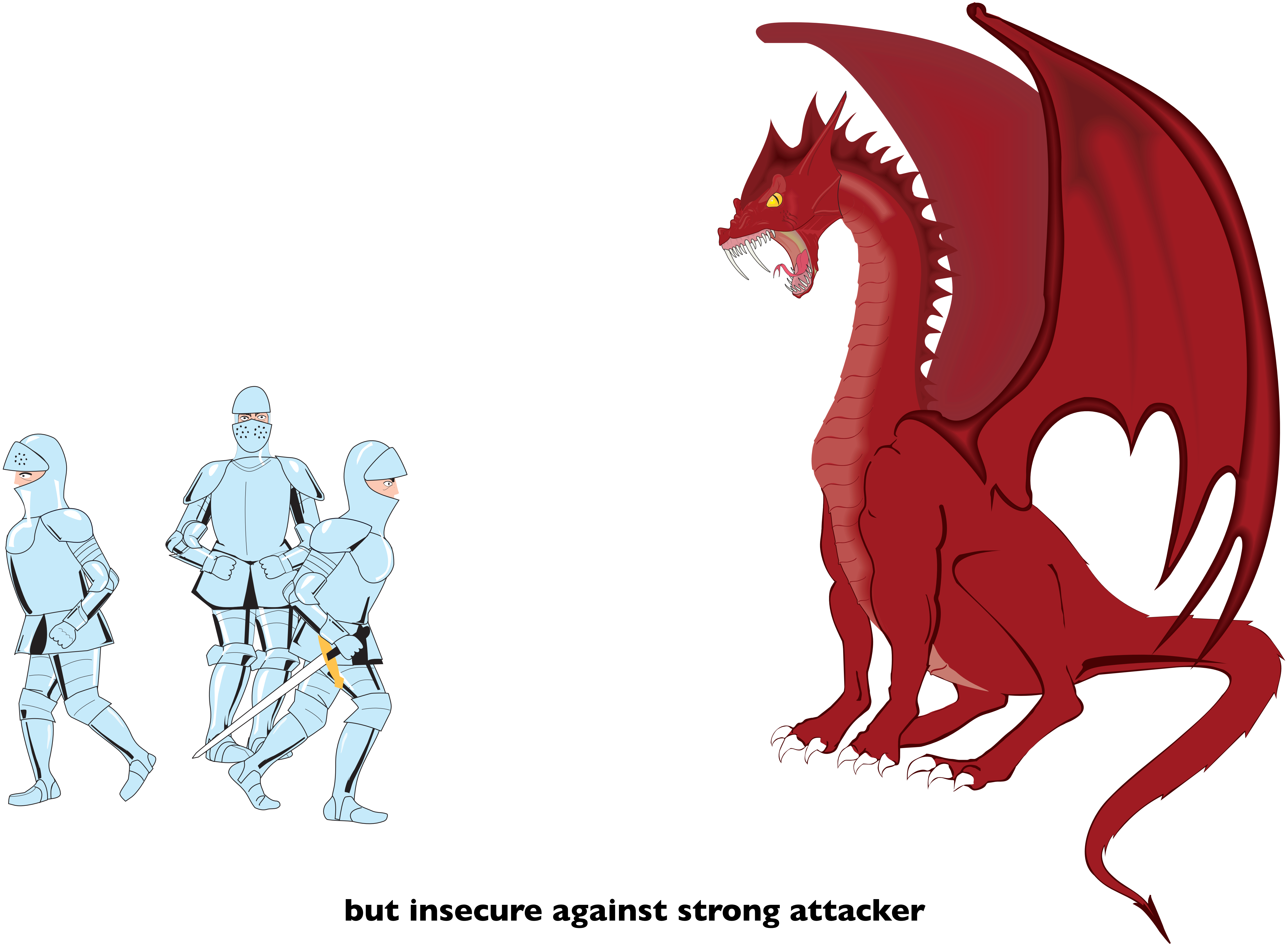
Michael Backes, Cătălin Hrițcu, Martin Grochulla and Matteo Maffei

to be presented at CSF 2009

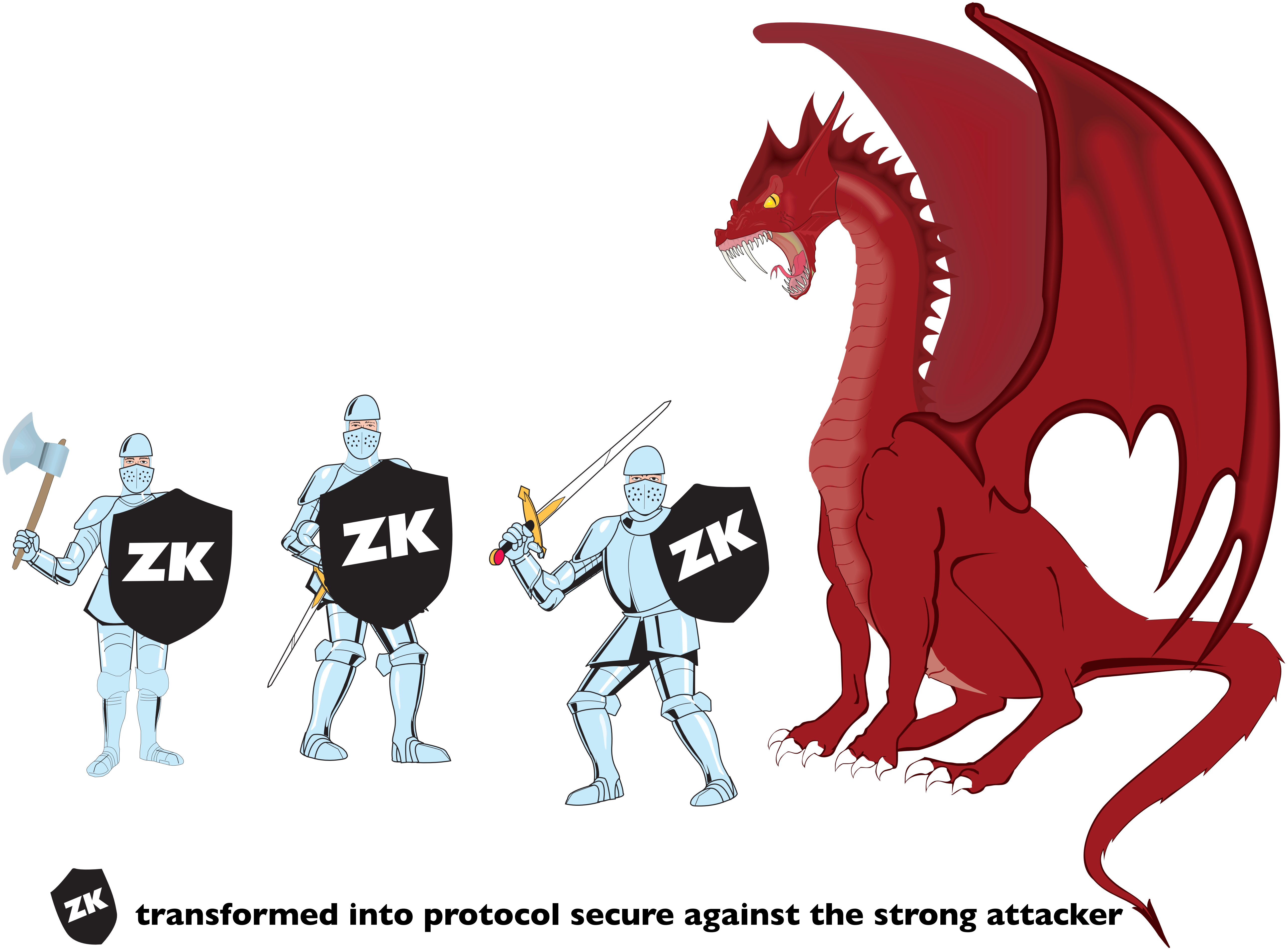
Reading Group Presentation, Saarbrücken, April 2009



protocol secure against a weak attacker



but insecure against strong attacker



ZK transformed into protocol secure against the strong attacker

- **General goal:** to aid secure protocol design
  - designer only needs to consider restricted security threats
- Automatic protocol transformation adding ZK proofs
  - Enforce any authorization policy even if some participants are compromised (security despite compromise)
  - Preserve secrecy if everybody is honest
- Automatic verification of the generated protocols (translation validation)
  - We use type system for ZK [Backes, Hritcu & Maffei, CCS ’08]
    - Now extended to handle security despite compromise
- Automatic code generation (from Spi to ML fragment)

What we did

- **General goal:** to aid secure protocol design
  - designer only needs to consider restricted security threats
- Automatic protocol transformation adding ZK proofs
  - Enforce any authorization policy even if some participants are compromised (security despite compromise)
  - Preserve secrecy if everybody is honest
- Automatic verification of the generated protocols (translation validation)
  - We use type system for ZK [Backes, Hritcu & Maffei, CCS ’08]
    - Now extended to handle security despite compromise
- Automatic code generation (from Spi to ML fragment)

What we did

- **General goal:** to aid secure protocol design
    - designer only needs to consider restricted security threats

- Automatic protocol transformation adding ZK proofs

    - Enforce any authorization policy even if some participants are compromised (security despite compromise)
    - Preserve secrecy if everybody is honest

What we did

- **General goal:** to aid secure protocol design
  - designer only needs to consider restricted security threats
    - can assume that all participants are honest
- Automatic protocol transformation adding ZK proofs

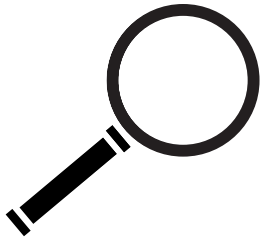
  - Enforce any authorization policy even if some participants are compromised (security despite compromise)

  - Preserve secrecy if everybody is honest

What we did

- **General goal:** to aid secure protocol design
  - designer only needs to consider restricted security threats
    - can assume that all participants are honest
- Automatic protocol transformation adding ZK proofs

  - Enforce any authorization policy even if some participants are compromised (security despite compromise)
  - Preserve secrecy if everybody is honest
- Automatic verification of the generated protocols (translation validation)

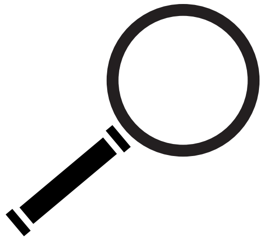


  - We use type system for ZK [Backes, Hritcu & Maffei, CCS ’08]
    - Now extended to handle security despite compromise

What we did

- **General goal:** to aid secure protocol design
  - designer only needs to consider restricted security threats
    - can assume that all participants are honest
- Automatic protocol transformation adding ZK proofs
  - Enforce any authorization policy even if some participants are compromised (security despite compromise)
  - Preserve secrecy if everybody is honest

- Automatic verification of the generated protocols (translation validation)
  - We use type system for ZK [Backes, Hritcu & Maffei, CCS ’08]
    - Now extended to handle security despite compromise

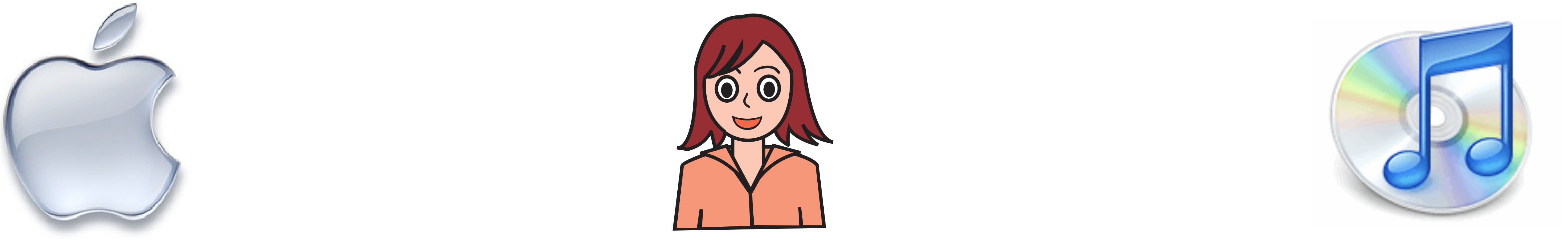


- Automatic code generation (from Spi to ML fragment)

Example

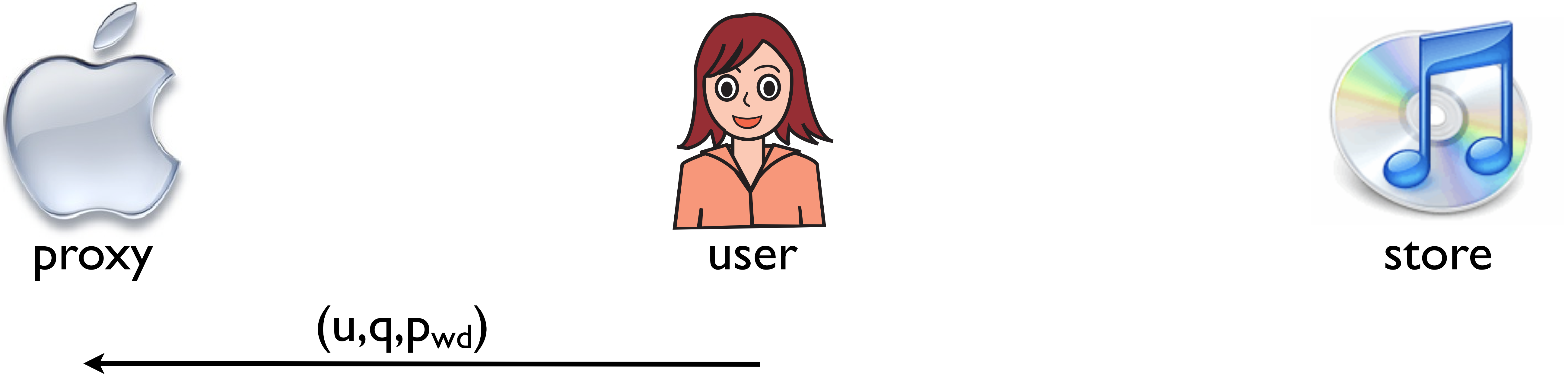
Adapted from [Fournet, Gordon & Maffeis, CSF ’07]

A simple protocol



proxy user store

A simple protocol



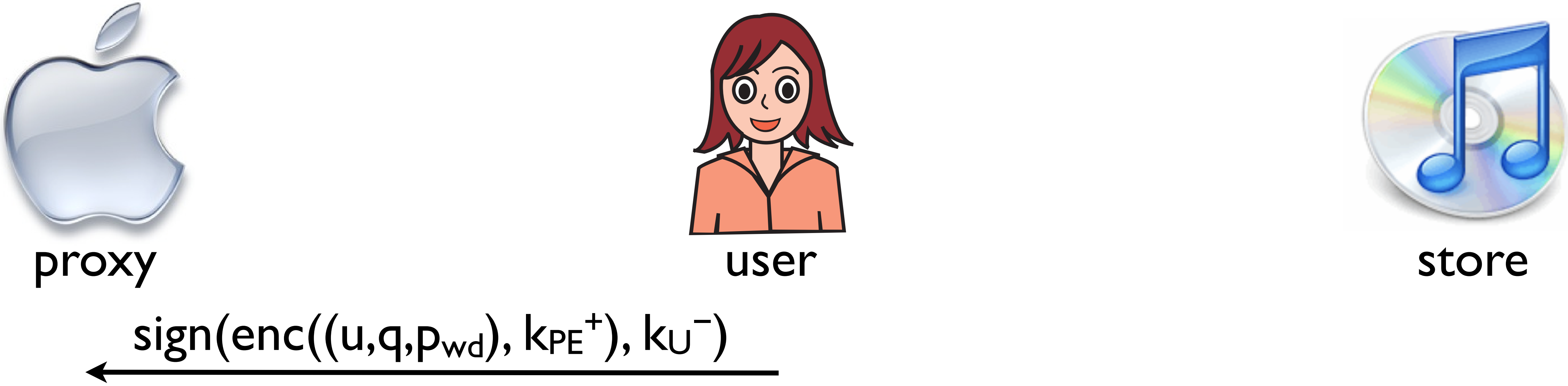
proxy

user

store

(u,q,p_{wd})

A simple protocol



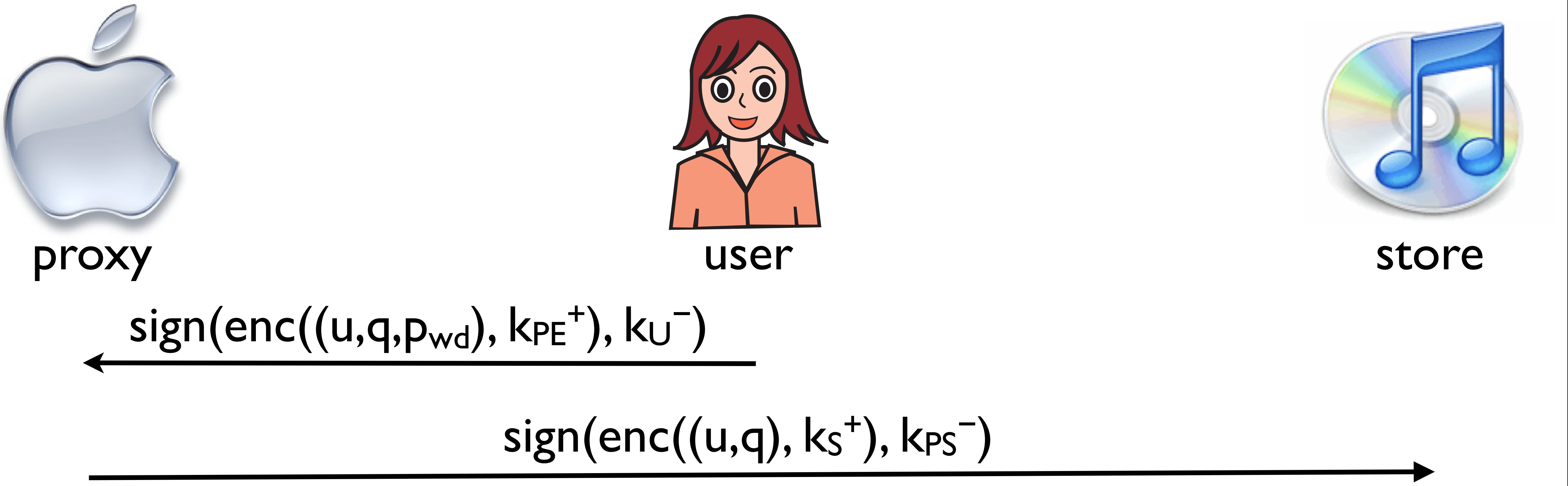
proxy

user

store

$\text{sign}(\text{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

A simple protocol



proxy

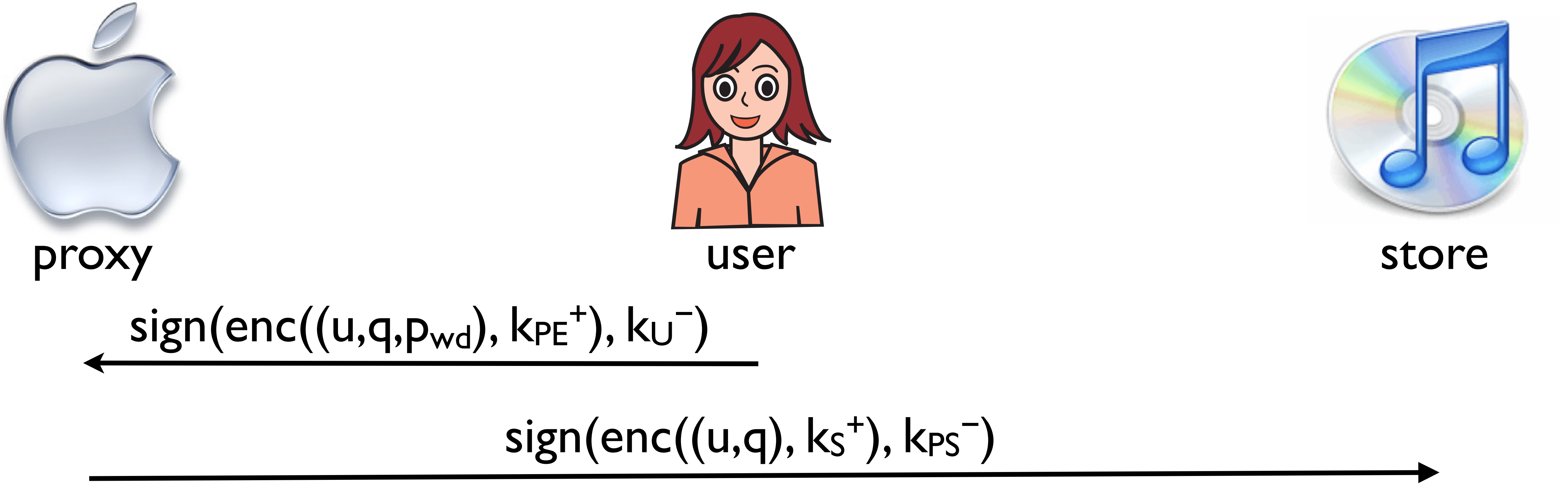
user

store

$\mathrm{sign}(\mathrm{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$\mathrm{sign}(\mathrm{enc}((u,q), k_{S}^{+}), k_{PS}^{-})$

A simple protocol



proxy

user

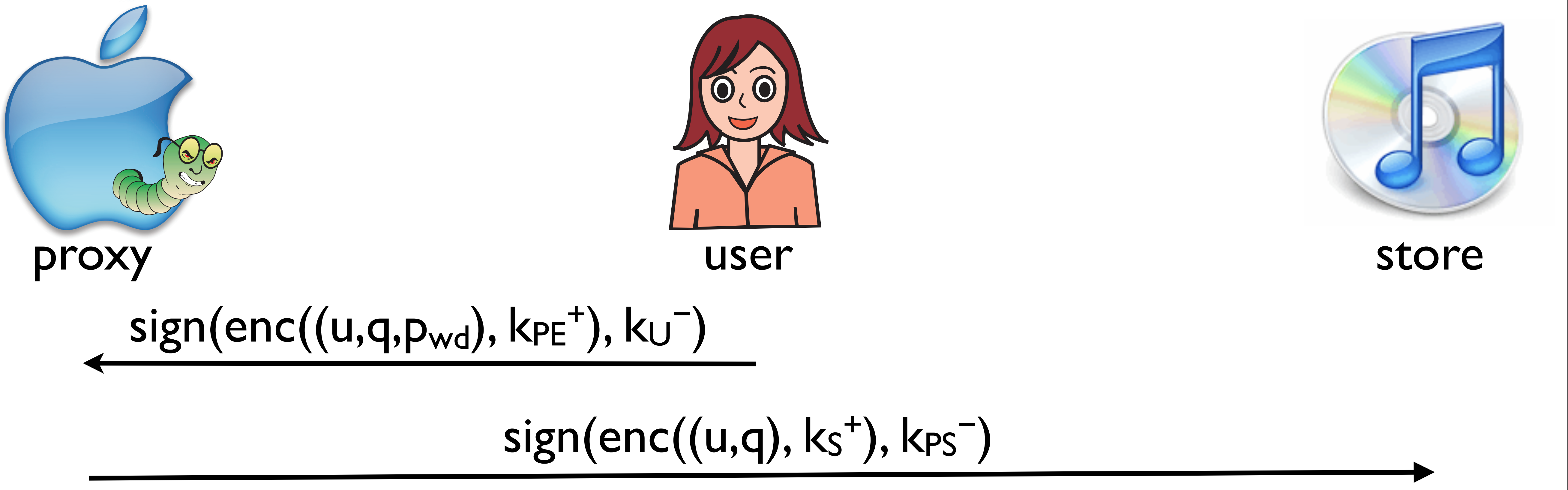
store

$\mathrm{sign}(\mathrm{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$\mathrm{sign}(\mathrm{enc}((u,q), k_{S}^{+}), k_{PS}^{-})$

- This protocol is secure if all participants are honest (q is secret and authentic)

A simple protocol



proxy

user

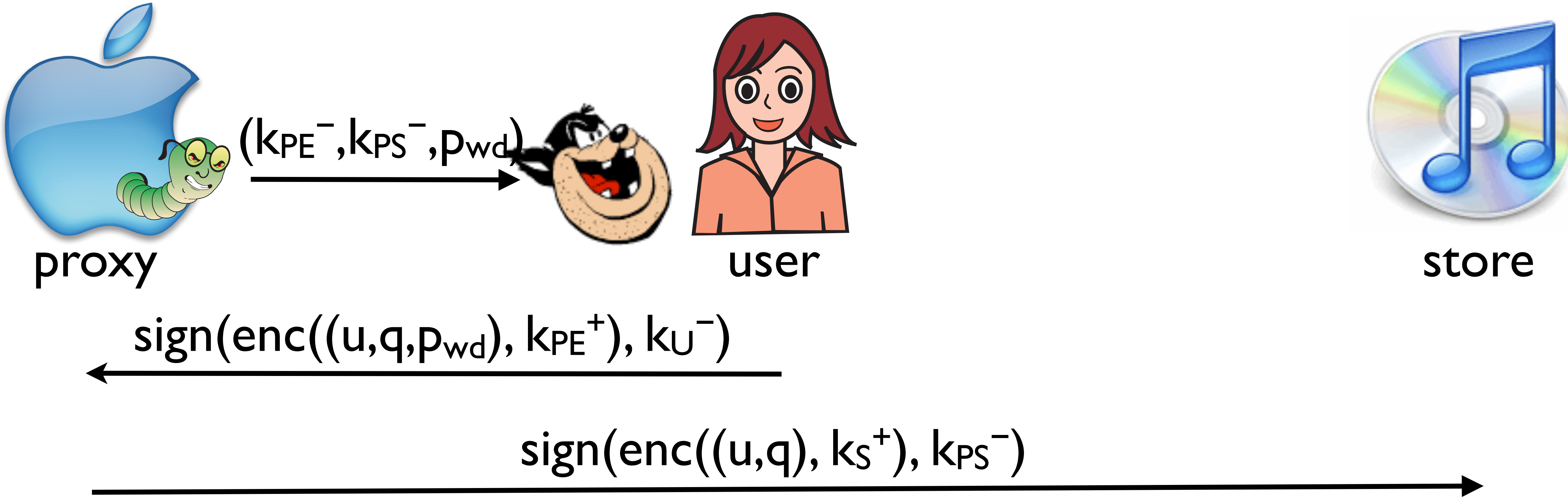
store

$\mathrm{sign}(\mathrm{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$\mathrm{sign}(\mathrm{enc}((u,q), k_{S}^{+}), k_{PS}^{-})$

- This protocol is secure if all participants are honest (q is secret and authentic)
- .... but insecure if the proxy is compromised

A simple protocol



$(k_{PE}^{-}, k_{PS}^{-}, p_{wd})$

proxy

user

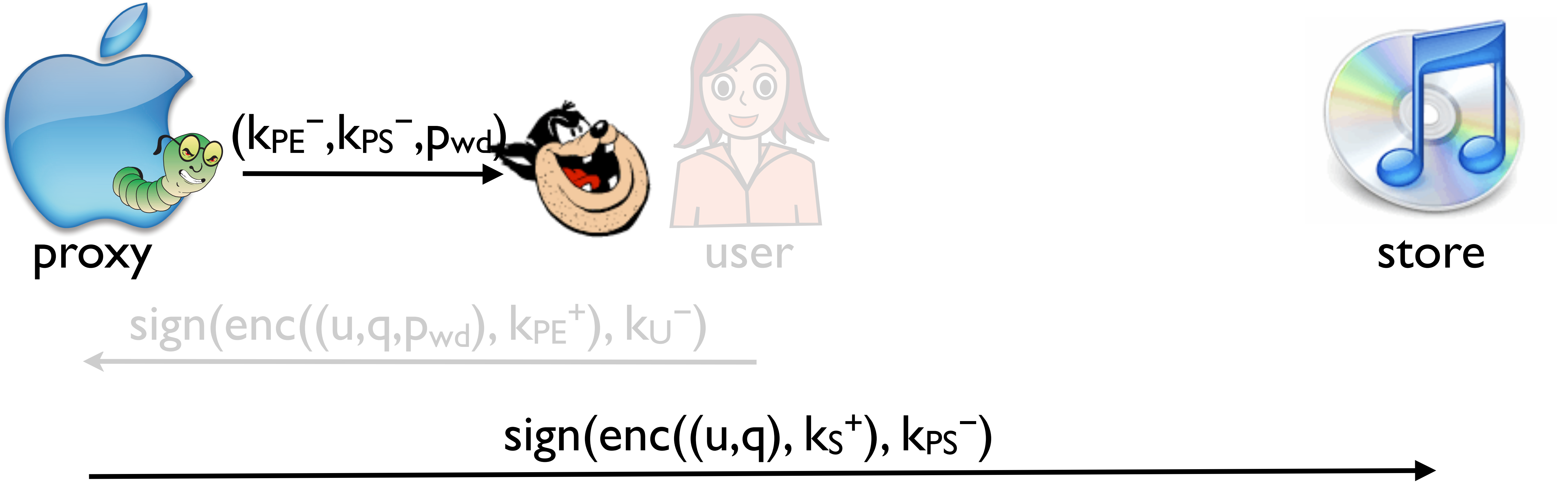
store

$\mathrm{sign}(\mathrm{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$\mathrm{sign}(\mathrm{enc}((u,q), k_{S}^{+}), k_{PS}^{-})$

- This protocol is secure if all participants are honest (q is secret and authentic)
- .... but insecure if the proxy is compromised
  - compromised proxy can leak q or p_{wd} (unavoidable)

A simple protocol



proxy

user

store

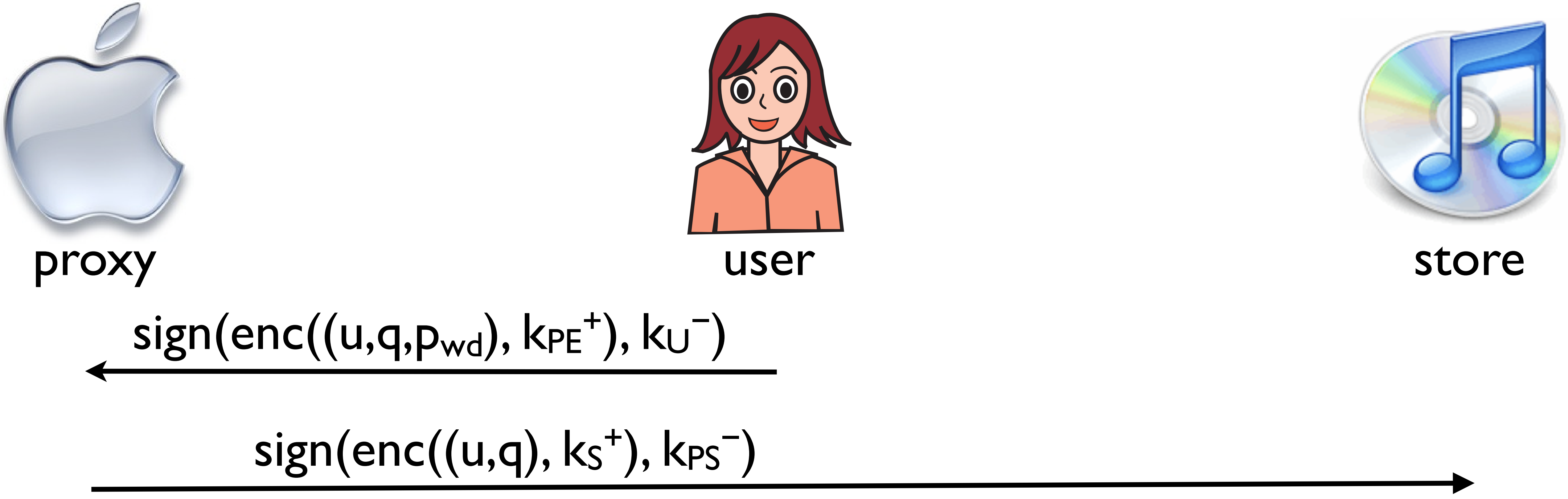
$(k_{PE}^{-}, k_{PS}^{-}, p_{wd})$

$sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$sign(enc((u,q), k_{S}^{+}), k_{PS}^{-})$

- This protocol is secure if all participants are honest (q is secret and authentic)
- .... but insecure if the proxy is compromised
    - compromised proxy can leak q or p_{wd} (unavoidable)
    - **compromised proxy can fake request from the user (break authenticity)**

Trying to strengthen the protocol



proxy

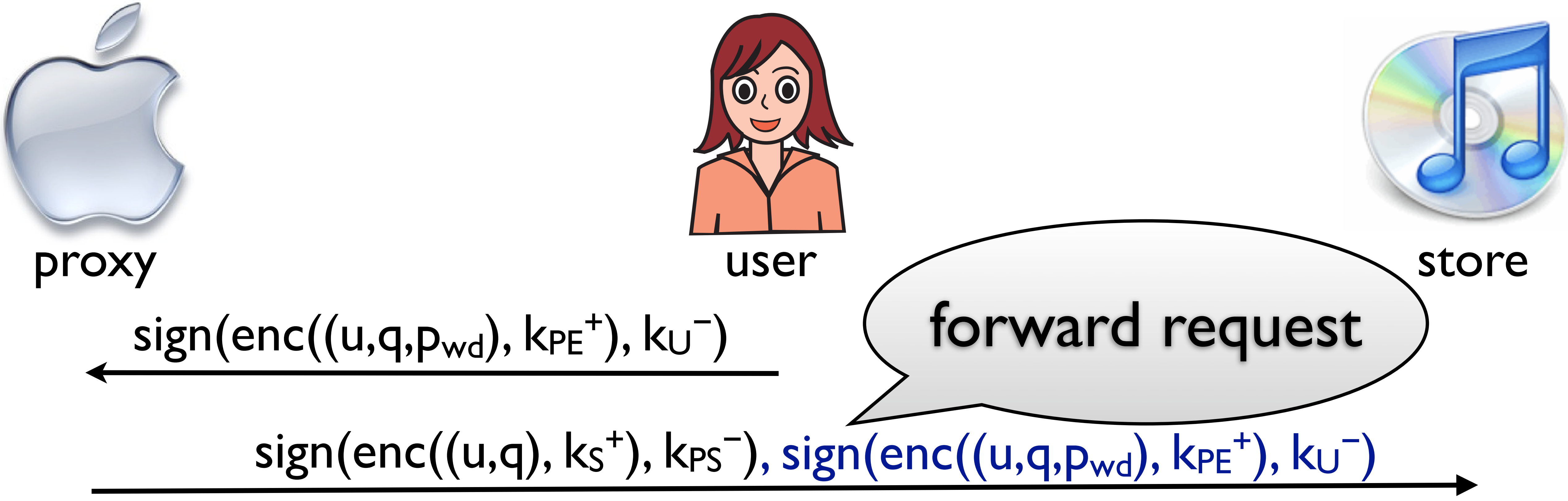
user

store

$\mathrm{sign}(\mathrm{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$\mathrm{sign}(\mathrm{enc}((u,q), k_{S}^{+}), k_{PS}^{-})$

Trying to strengthen the protocol



proxy

user

store

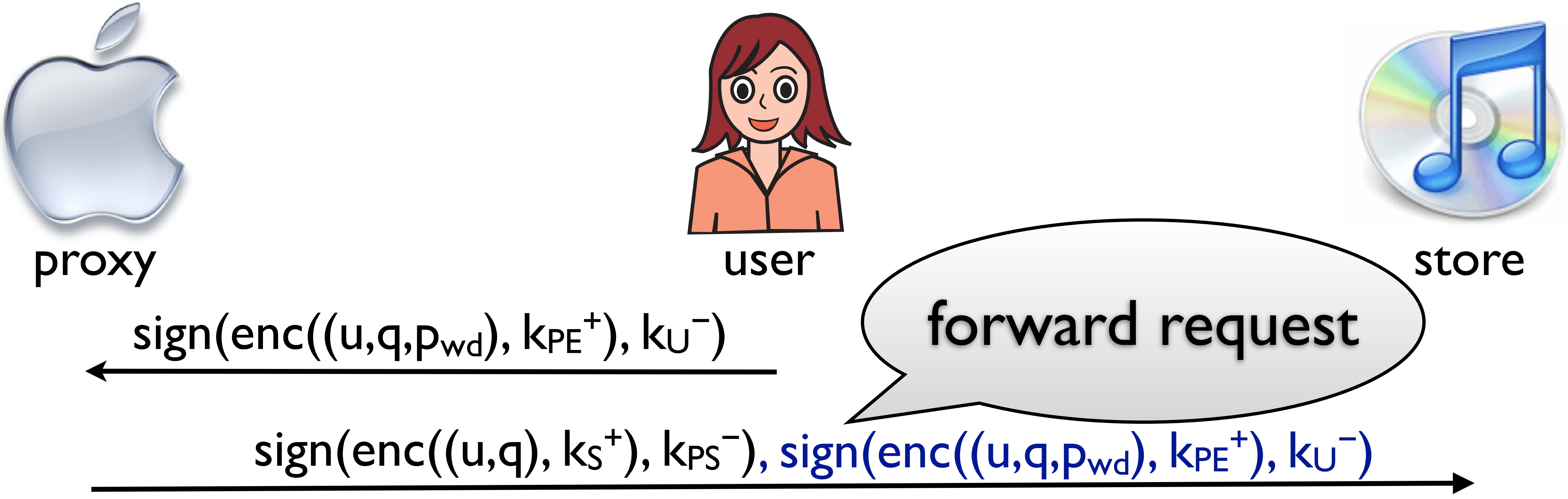
forward request

$\text{sign}(\text{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$\text{sign}(\text{enc}((u,q), k_{S}^{+}), k_{PS}^{-})$, $\text{sign}(\text{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

- Store can check user's signature on "$\text{enc}((q,p_{wd}),k_{PE}^{+})$"

Trying to strengthen the protocol

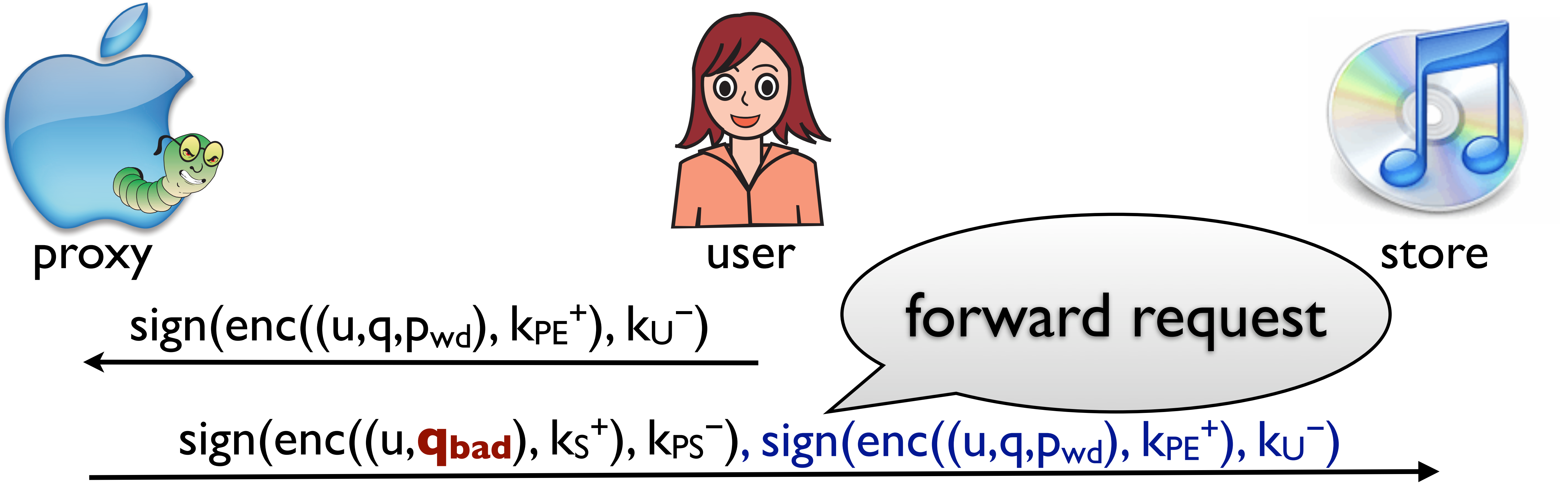


sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})

sign(enc((u,q), k_{S}^{+}), k_{PS}^{-}), sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})

- Store can check user's signature on "enc((q,p_{wd}),k_{PE}^{+})"
- Store cannot decrypt "enc((u,q,p_{wd}),k_{PE}^{+})" in order to check q

Trying to strengthen the protocol



proxy

user

store

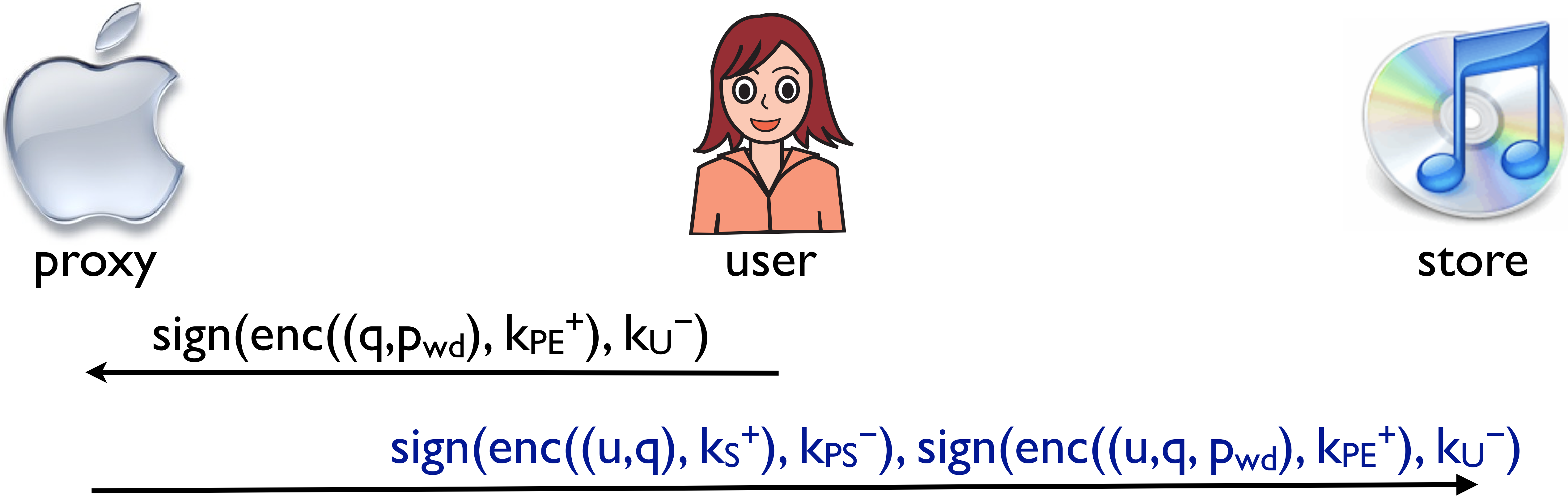
$\mathrm{sign}(\mathrm{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

forward request

$\mathrm{sign}(\mathrm{enc}((u,\mathbf{q_{bad}}), k_{S}^{+}), k_{PS}^{-}), \mathrm{sign}(\mathrm{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

- Store can check user's signature on "$\mathrm{enc}((q,p_{wd}),k_{PE}^{+})$"
- Store cannot decrypt "$\mathrm{enc}((u,q,p_{wd}),k_{PE}^{+})$" in order to check q
- **... still insecure if proxy comprised (message substitution attack)**

Using non-interactive ZK



proxy

user

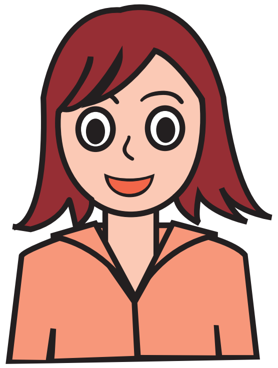
store

$\mathrm{sign}(\mathrm{enc}((q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$\mathrm{sign}(\mathrm{enc}((u,q), k_{S}^{+}), k_{PS}^{-}), \mathrm{sign}(\mathrm{enc}((u,q, p_{wd}), k_{PE}^{+}), k_{U}^{-})$

Using non-interactive ZK

proxy



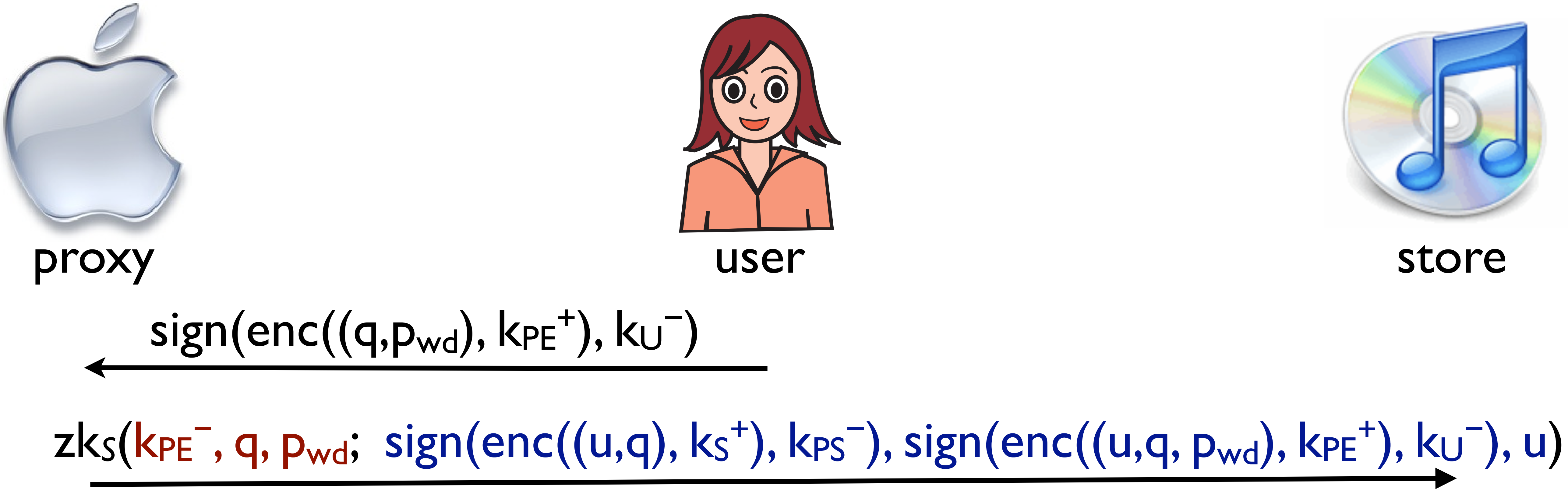
user

store

$\mathrm{sign}(\mathrm{enc}((q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$ ←

$\mathrm{zk}_{S}(k_{PE}^{-}, q, p_{wd};\ \mathrm{sign}(\mathrm{enc}((u,q), k_{S}^{+}), k_{PS}^{-}), \mathrm{sign}(\mathrm{enc}((u,q, p_{wd}), k_{PE}^{+}), k_{U}^{-}), u)$ →

Using non-interactive ZK

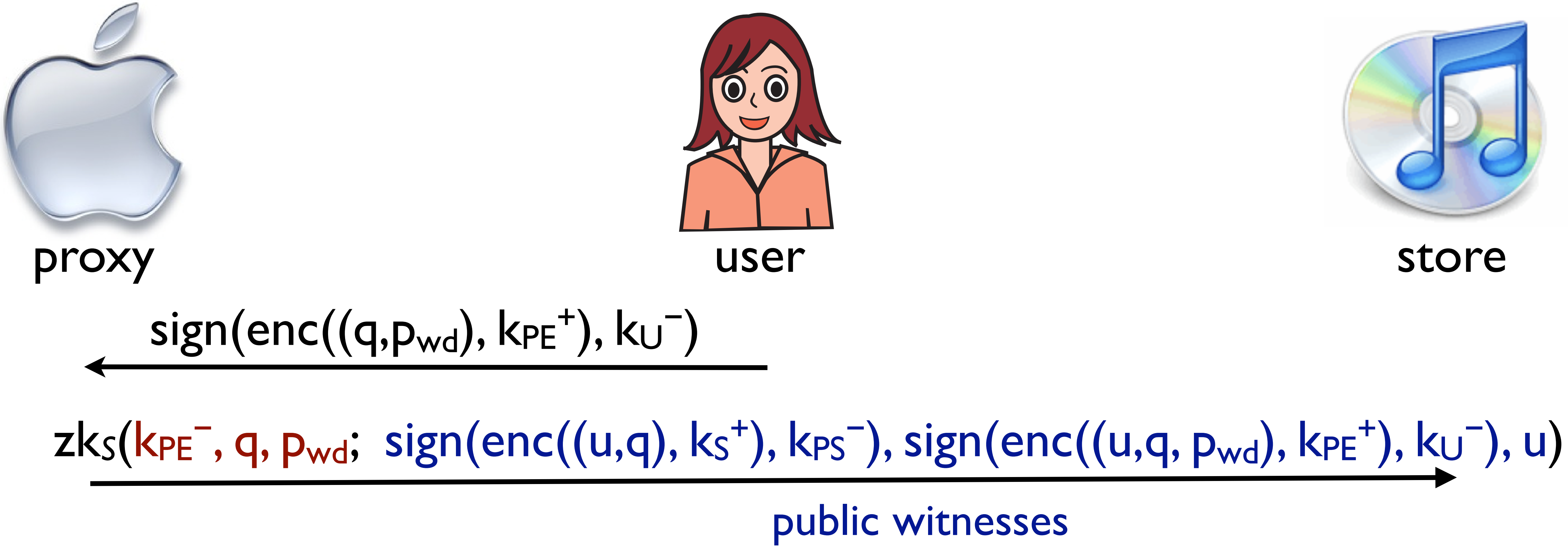


$\text{sign}(\text{enc}((q, p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$zk_{S}(k_{PE}^{-}, q, p_{wd};\ \text{sign}(\text{enc}((u,q), k_{S}^{+}), k_{PS}^{-}), \text{sign}(\text{enc}((u,q, p_{wd}), k_{PE}^{+}), k_{U}^{-}), u)$

secret
witnesses

Using non-interactive ZK



proxy

user

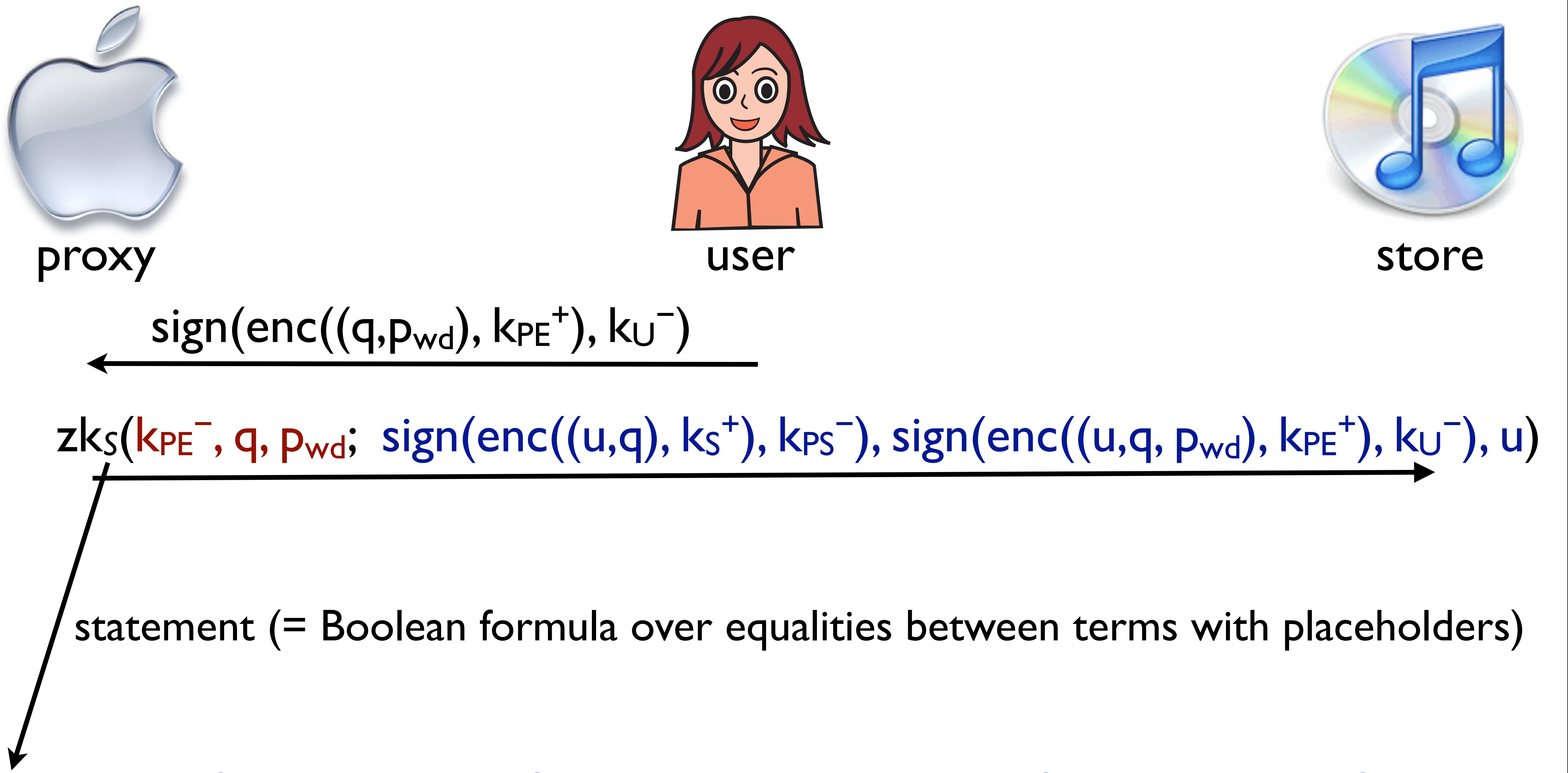
store

$\mathrm{sign}(\mathrm{enc}((q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$\mathrm{zk}_{S}(k_{PE}^{-}, q, p_{wd};\ \mathrm{sign}(\mathrm{enc}((u,q), k_{S}^{+}), k_{PS}^{-}), \mathrm{sign}(\mathrm{enc}((u,q, p_{wd}), k_{PE}^{+}), k_{U}^{-}), u)$

public witnesses

Using non-interactive ZK



proxy

user

store

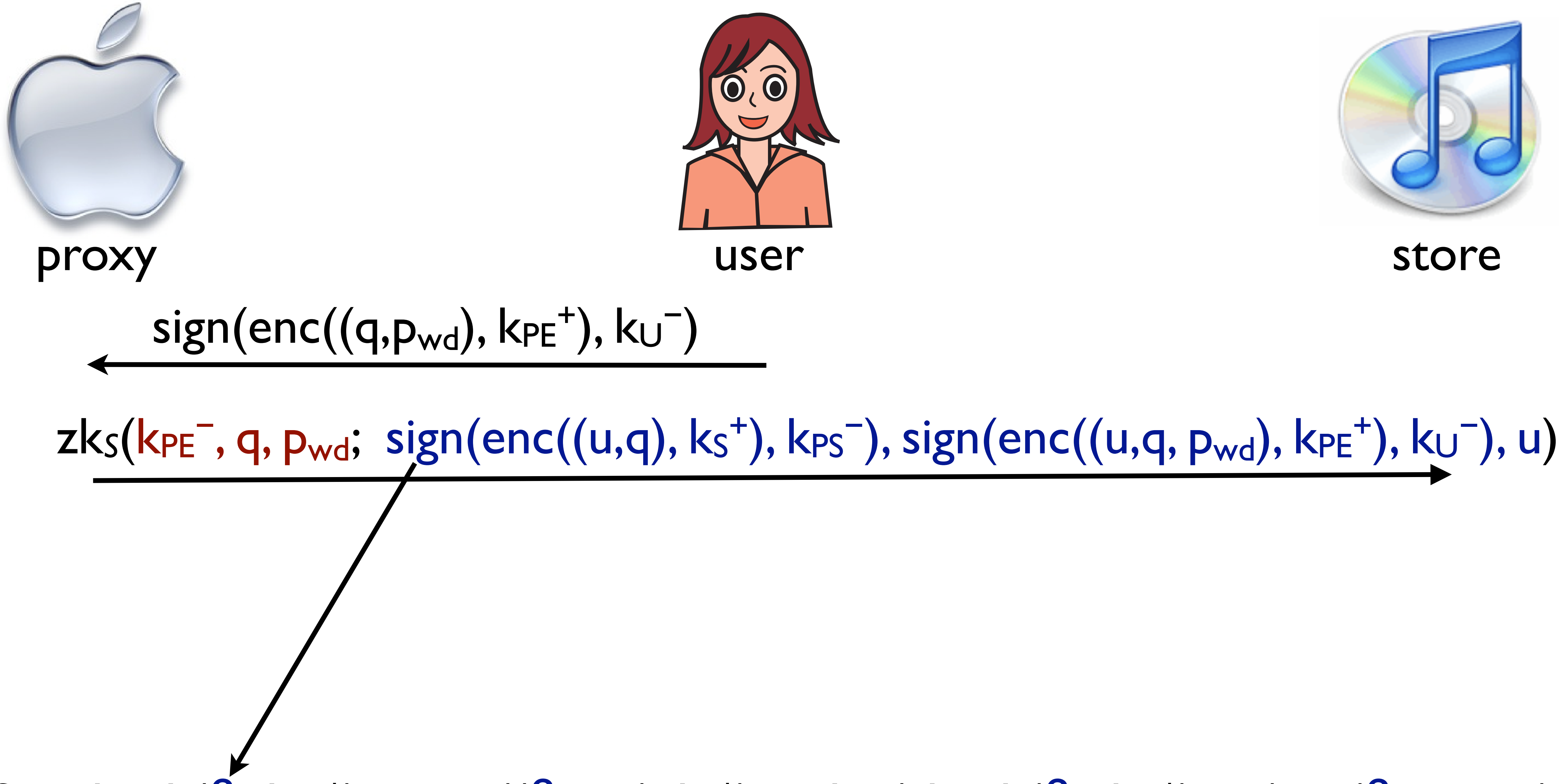
$\mathrm{sign}(\mathrm{enc}((q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$\mathrm{zk}_{S}(k_{PE}^{-}, q, p_{wd};\ \mathrm{sign}(\mathrm{enc}((u,q), k_{S}^{+}), k_{PS}^{-}), \mathrm{sign}(\mathrm{enc}((u,q, p_{wd}), k_{PE}^{+}), k_{U}^{-}), u)$

statement (= Boolean formula over equalities between terms with placeholders)

$S = \mathrm{check}(\beta_{1},k_{PS}^{+}) = \mathrm{enc}((\beta_{3},\alpha_{2}), k_{S}^{+}) \wedge \mathrm{dec}(\mathrm{check}(\beta_{2}, k_{U}^{+}), \alpha_{1}) = (\beta_{3},\alpha_{2},\alpha_{3})$

Using non-interactive ZK

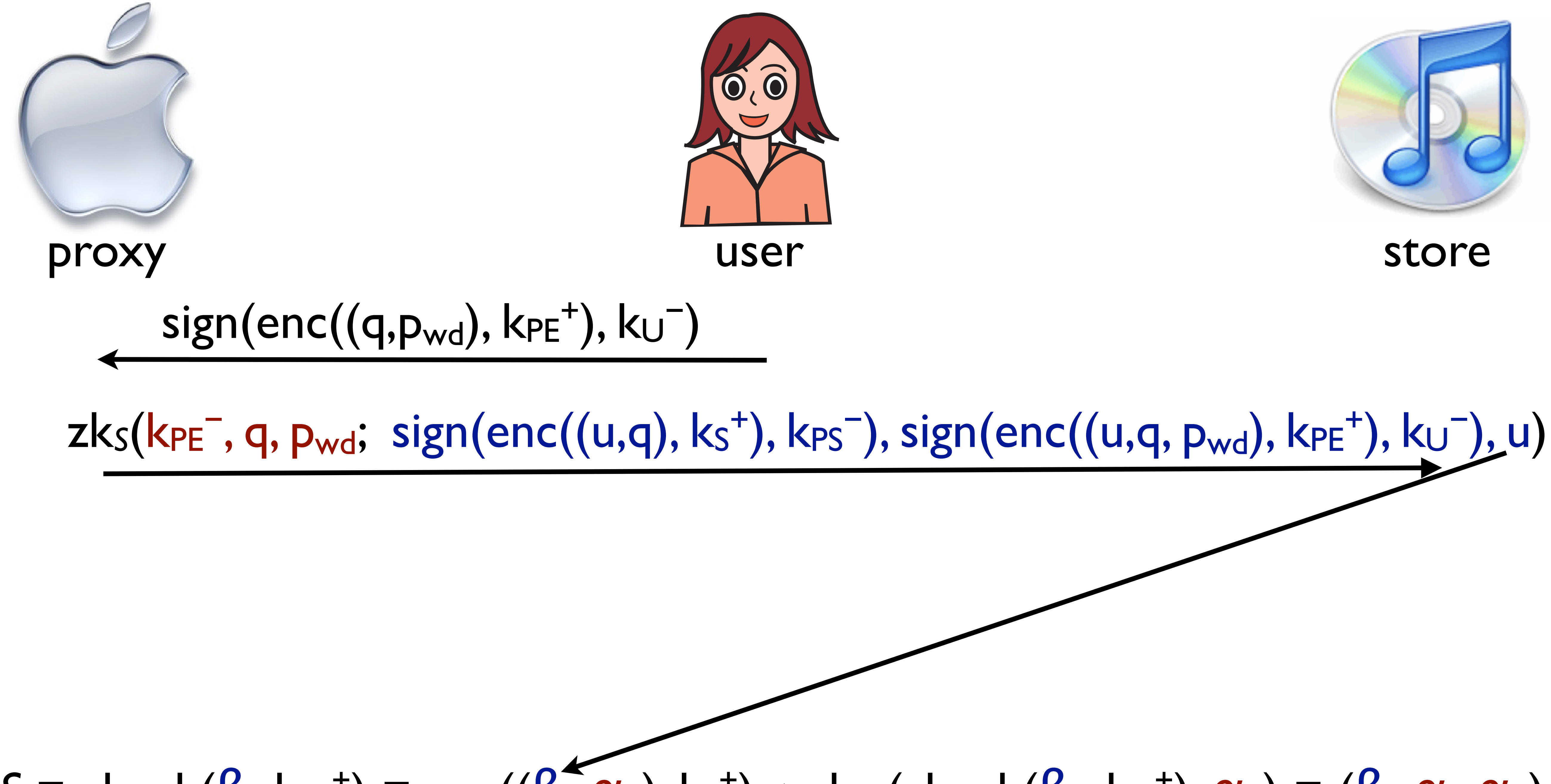


$\mathrm{sign}(\mathrm{enc}((q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$zk_{S}(k_{PE}^{-}, q, p_{wd};\ \mathrm{sign}(\mathrm{enc}((u,q), k_{S}^{+}), k_{PS}^{-}), \mathrm{sign}(\mathrm{enc}((u,q, p_{wd}), k_{PE}^{+}), k_{U}^{-}), u)$

$$S = \mathrm{check}(\beta_1,k_{PS}^{+}) = \mathrm{enc}((\beta_3,\alpha_2), k_{S}^{+}) \wedge \mathrm{dec}(\mathrm{check}(\beta_2, k_{U}^{+}), \alpha_1) = (\beta_3,\alpha_2,\alpha_3)$$

Using non-interactive ZK



proxy

user

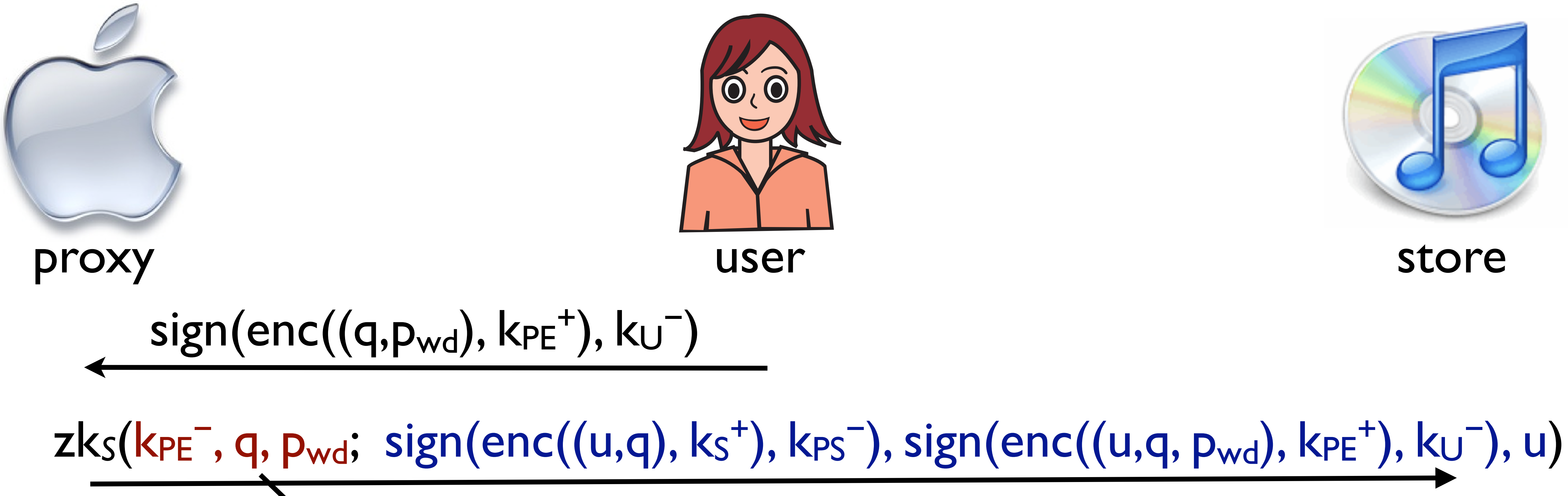
store

$\mathrm{sign}(\mathrm{enc}((q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$\mathrm{zk}_{S}(k_{PE}^{-}, q, p_{wd};\ \mathrm{sign}(\mathrm{enc}((u,q), k_{S}^{+}), k_{PS}^{-}), \mathrm{sign}(\mathrm{enc}((u,q, p_{wd}), k_{PE}^{+}), k_{U}^{-}), u)$

$$S = \mathrm{check}(\beta_1,k_{PS}^{+}) = \mathrm{enc}((\beta_3,\alpha_2), k_{S}^{+}) \wedge \mathrm{dec}(\mathrm{check}(\beta_2, k_{U}^{+}), \alpha_1) = (\beta_3,\alpha_2,\alpha_3)$$

Using non-interactive ZK

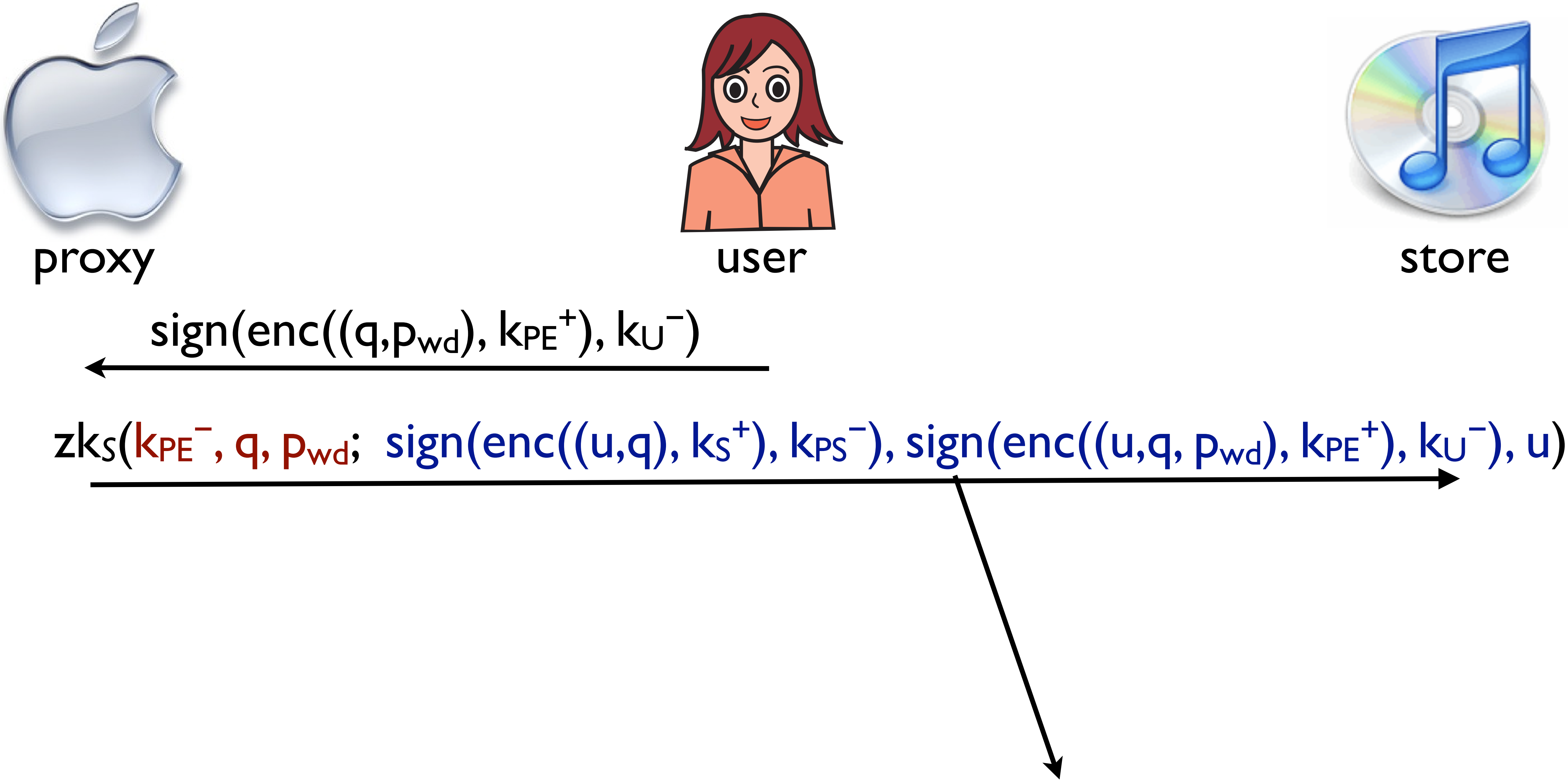


$\mathrm{sign}(\mathrm{enc}((q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$\mathrm{zk}_{S}(k_{PE}^{-}, q, p_{wd};\ \mathrm{sign}(\mathrm{enc}((u,q), k_{S}^{+}), k_{PS}^{-}), \mathrm{sign}(\mathrm{enc}((u,q, p_{wd}), k_{PE}^{+}), k_{U}^{-}), u)$

$$S = \mathrm{check}(\beta_1,k_{PS}^{+}) = \mathrm{enc}((\beta_3,\alpha_2), k_{S}^{+}) \wedge \mathrm{dec}(\mathrm{check}(\beta_2, k_{U}^{+}), \alpha_1) = (\beta_3,\alpha_2,\alpha_3)$$

Using non-interactive ZK



proxy

user

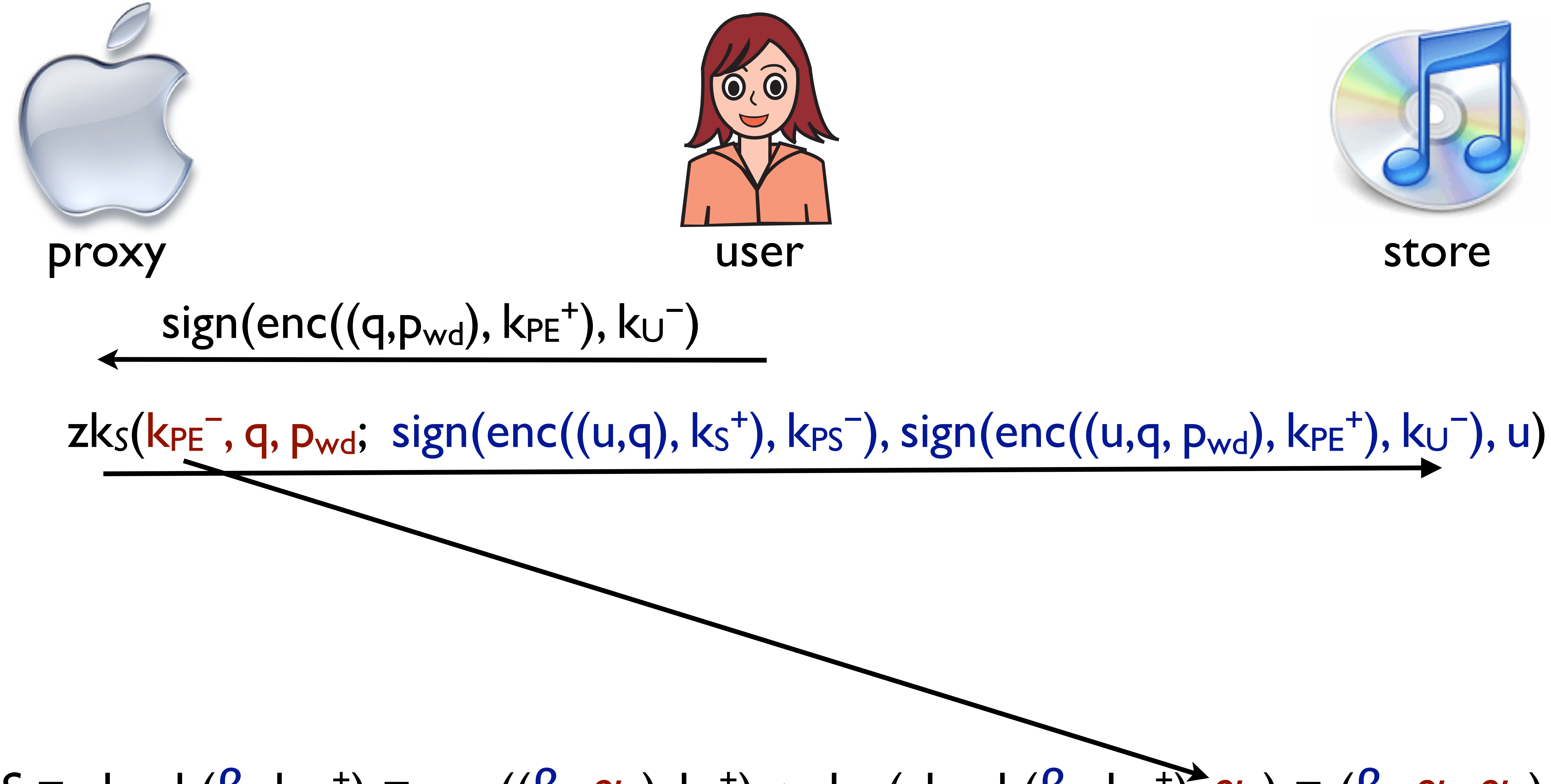
store

$\mathrm{sign}(\mathrm{enc}((q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$zk_{S}(k_{PE}^{-}, q, p_{wd};\ \mathrm{sign}(\mathrm{enc}((u,q), k_{S}^{+}), k_{PS}^{-}), \mathrm{sign}(\mathrm{enc}((u,q, p_{wd}), k_{PE}^{+}), k_{U}^{-}), u)$

$$S = \mathrm{check}(\beta_1,k_{PS}^{+}) = \mathrm{enc}((\beta_3,\alpha_2), k_S^{+}) \wedge \mathrm{dec}(\mathrm{check}(\beta_2, k_U^{+}), \alpha_1) = (\beta_3,\alpha_2,\alpha_3)$$

Using non-interactive ZK

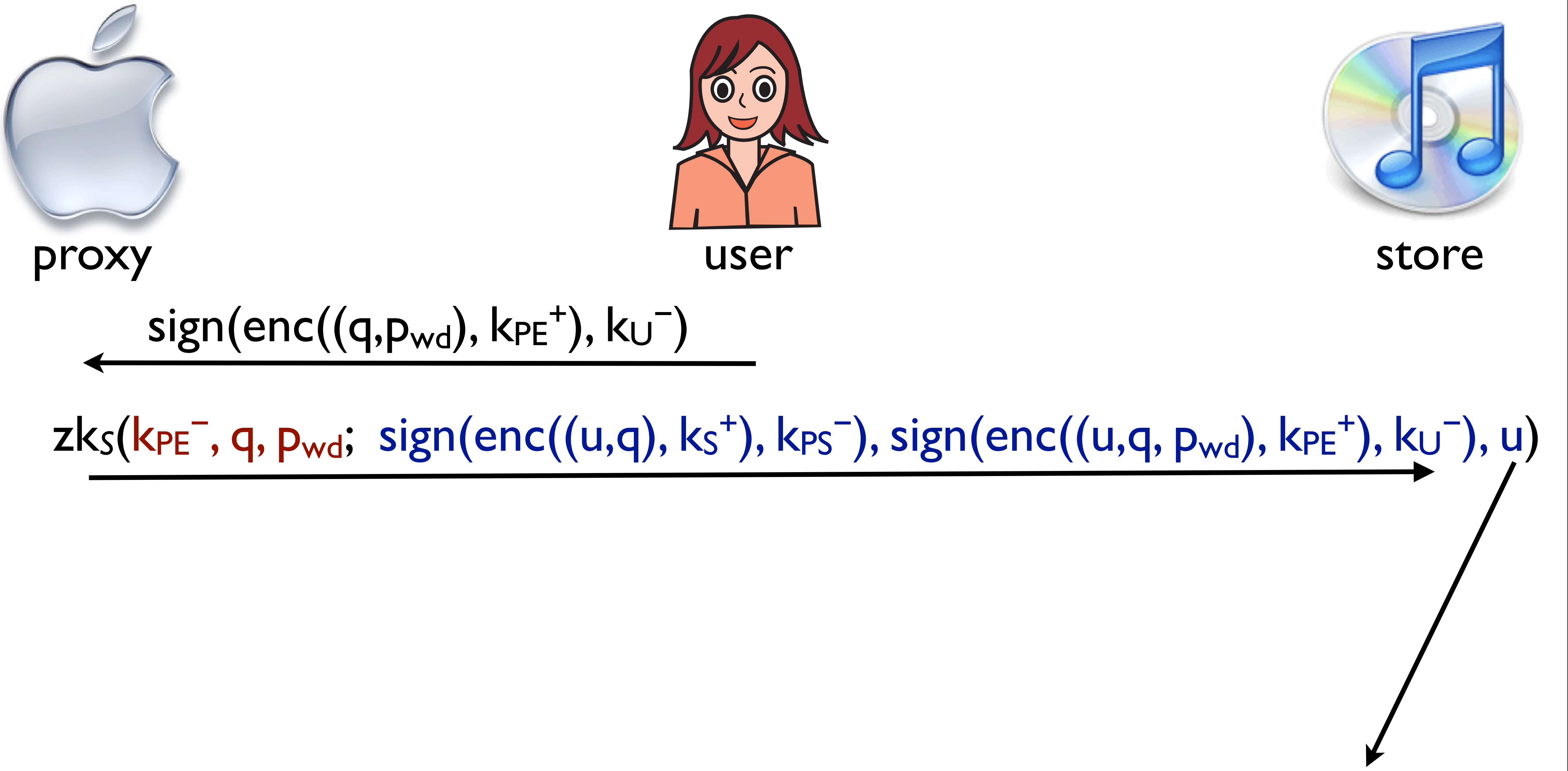


$\mathrm{sign}(\mathrm{enc}((q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$\mathrm{zk}_{S}(k_{PE}^{-}, q, p_{wd};\ \mathrm{sign}(\mathrm{enc}((u,q), k_{S}^{+}), k_{PS}^{-}), \mathrm{sign}(\mathrm{enc}((u,q, p_{wd}), k_{PE}^{+}), k_{U}^{-}), u)$

$$S = \mathrm{check}(\beta_1,k_{PS}^{+}) = \mathrm{enc}((\beta_3,\alpha_2), k_{S}^{+}) \wedge \mathrm{dec}(\mathrm{check}(\beta_2, k_{U}^{+}), \alpha_1) = (\beta_3,\alpha_2,\alpha_3)$$

Using non-interactive ZK



proxy

user

store

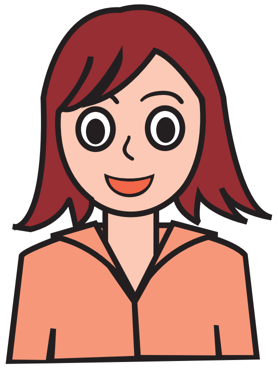
$\mathrm{sign}(\mathrm{enc}((q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$zk_{S}(k_{PE}^{-}, q, p_{wd};\ \mathrm{sign}(\mathrm{enc}((u,q), k_{S}^{+}), k_{PS}^{-}), \mathrm{sign}(\mathrm{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-}), u)$

$$S = \mathrm{check}(\beta_1, k_{PS}^{+}) = \mathrm{enc}((\beta_3,\alpha_2), k_S^{+}) \wedge \mathrm{dec}(\mathrm{check}(\beta_2, k_U^{+}), \alpha_1) = (\beta_3,\alpha_2,\alpha_3)$$

Using non-interactive ZK

proxy



user

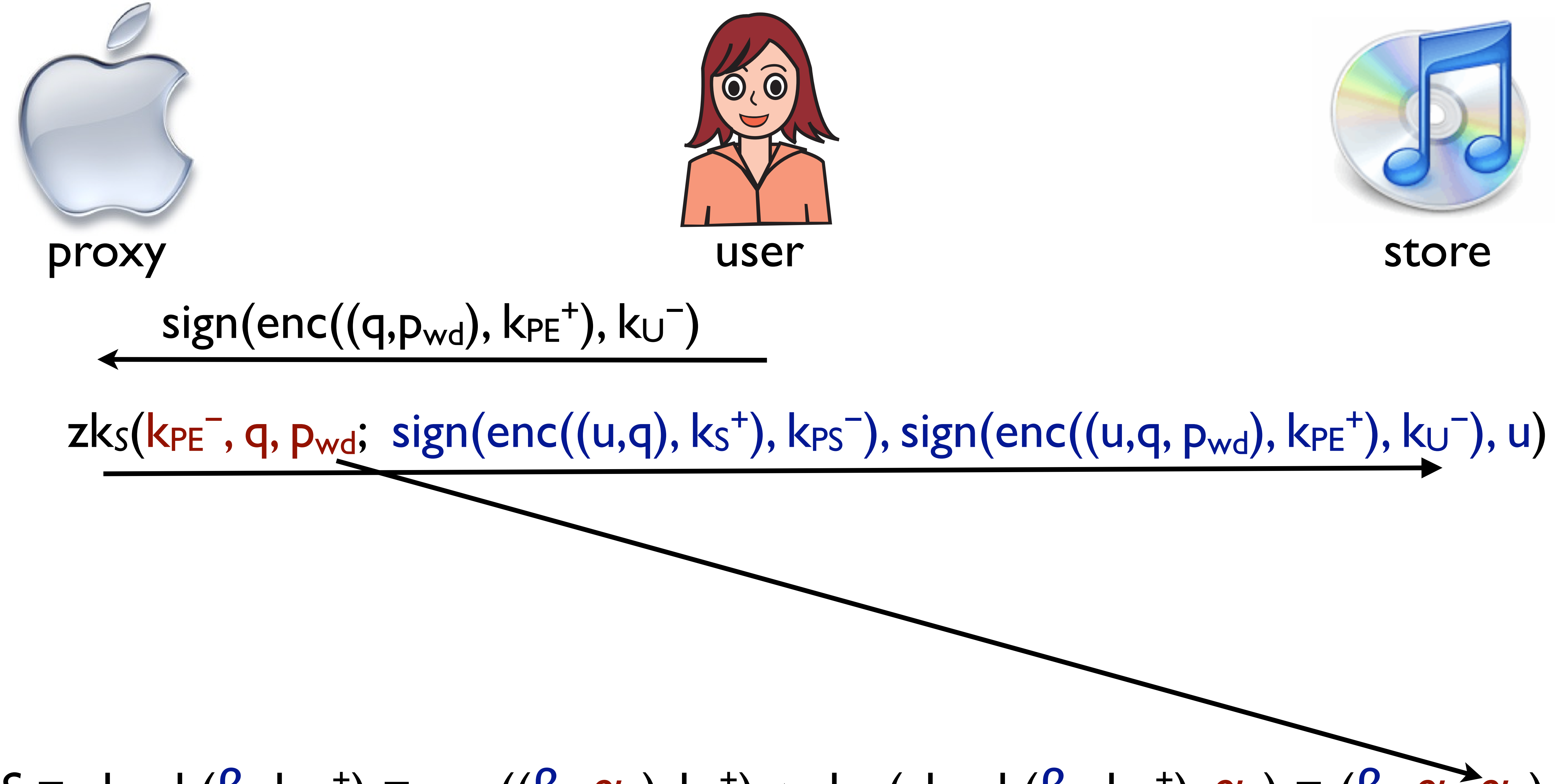
store

$$\xleftarrow{\mathrm{sign}(\mathrm{enc}((q,p_{wd}),\,k_{PE}^{+}),\,k_{U}^{-})}$$

$$\xrightarrow{zk_S(k_{PE}^{-},\,q,\,p_{wd};\;\mathrm{sign}(\mathrm{enc}((u,q),\,k_S^{+}),\,k_{PS}^{-}),\,\mathrm{sign}(\mathrm{enc}((u,q,\,p_{wd}),\,k_{PE}^{+}),\,k_U^{-}),\,u)}$$

$$S = \mathrm{check}(\beta_1,k_{PS}^{+}) = \mathrm{enc}((\beta_3,\alpha_2),\,k_S^{+}) \wedge \mathrm{dec}(\mathrm{check}(\beta_2,\,k_U^{+}),\,\alpha_1) = (\beta_3,\alpha_2,\alpha_3)$$

Using non-interactive ZK



proxy

user

store

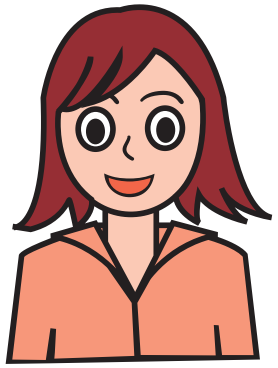
$\mathrm{sign}(\mathrm{enc}((q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$zk_{S}(k_{PE}^{-}, q, p_{wd};\ \mathrm{sign}(\mathrm{enc}((u,q), k_{S}^{+}), k_{PS}^{-}), \mathrm{sign}(\mathrm{enc}((u,q, p_{wd}), k_{PE}^{+}), k_{U}^{-}), u)$

$S = \mathrm{check}(\beta_1, k_{PS}^{+}) = \mathrm{enc}((\beta_3,\alpha_2), k_{S}^{+}) \wedge \mathrm{dec}(\mathrm{check}(\beta_2, k_{U}^{+}), \alpha_1) = (\beta_3,\alpha_2,\alpha_3)$

Using non-interactive ZK

proxy



user

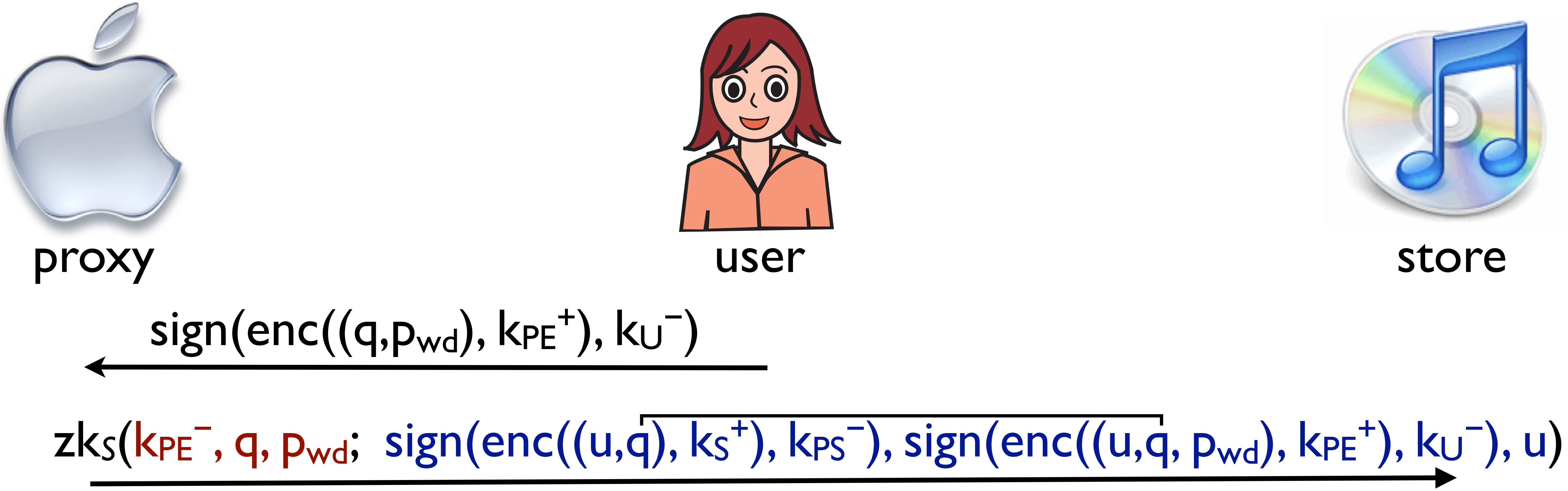
store

$$\xleftarrow{\mathrm{sign}(\mathrm{enc}((q,p_{wd}), k_{PE}^{+}), k_U^{-})}$$

$$\xrightarrow{zk_S(k_{PE}^{-}, q, p_{wd};\ \mathrm{sign}(\mathrm{enc}((u,\overbrace{q), k_S^{+}), k_{PS}^{-}), \mathrm{sign}(\mathrm{enc}((u,q}, p_{wd}), k_{PE}^{+}), k_U^{-}), u)}$$

$$S = \mathrm{check}(\beta_1, k_{PS}^{+}) = \mathrm{enc}((\beta_3, \underbrace{\alpha_2), k_S^{+}) \wedge \mathrm{dec}(\mathrm{check}(\beta_2, k_U^{+}), \alpha_1) = (\beta_3, \alpha_2}, \alpha_3)$$

Using non-interactive ZK



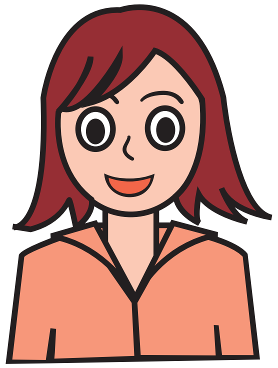
$sign(enc((q,p_{wd}), k_{PE}^+), k_U^-)$

$zk_S(k_{PE}^-, q, p_{wd};\ sign(enc((u,\overline{q), k_S^+), k_{PS}^-), sign(enc((u,q}, p_{wd}), k_{PE}^+), k_U^-), u)$

- The proxy has to prove that its message is correctly generated from a request he received from the user

Using non-interactive ZK

proxy



user

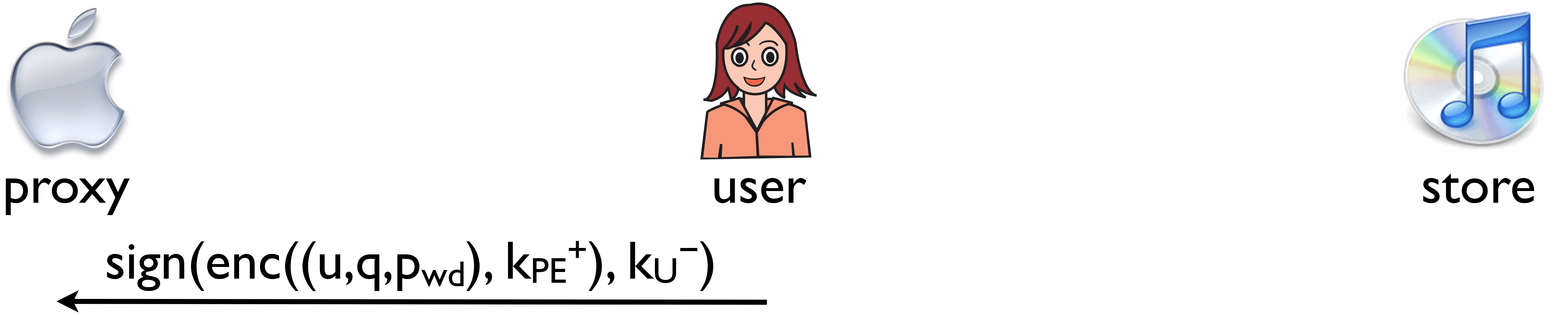
store

$\xleftarrow{\quad sign(enc((q,p_{wd}), k_{PE}^{+}), k_{U}^{-}) \quad}$

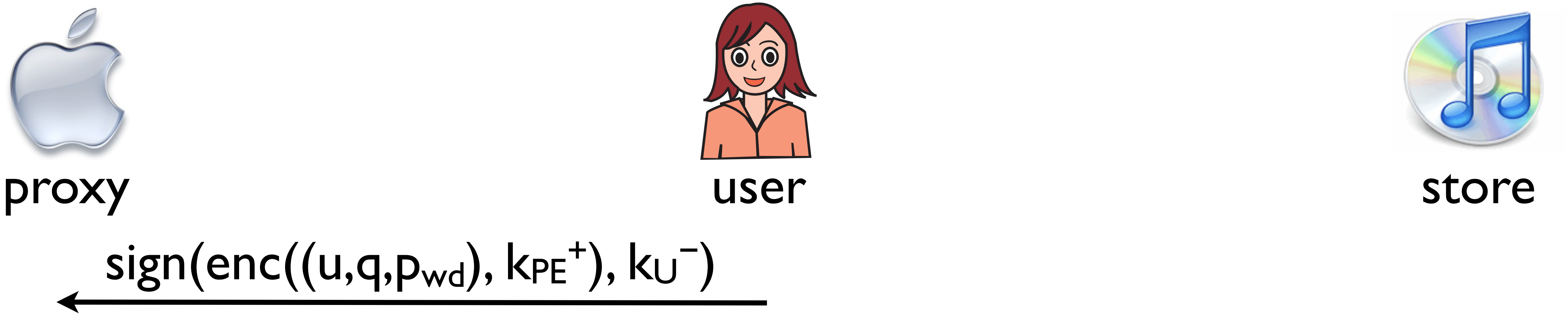
$zk_{S}(k_{PE}^{-}, q, p_{wd};\; sign(enc((u,q), k_{S}^{+}), k_{PS}^{-}), sign(enc((u,q, p_{wd}), k_{PE}^{+}), k_{U}^{-}), u) \longrightarrow$

- The proxy has to prove that its message is correctly generated from a request he received from the user
  - Compromised proxy can no longer cheat

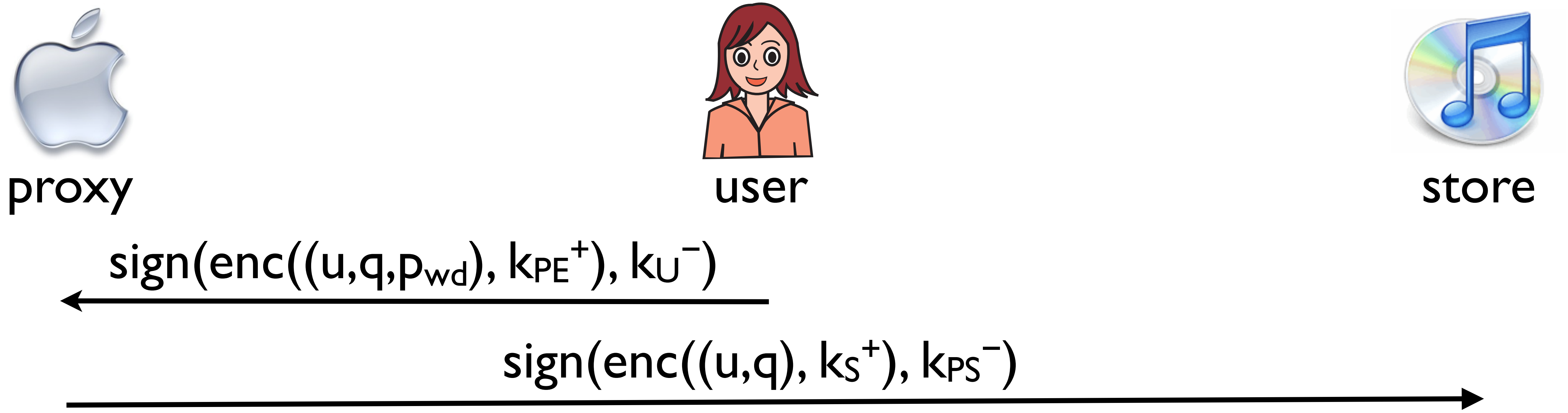
Transformation in slightly more detail



$\mathrm{sign}(\mathrm{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$


proxy
user
store
$\mathrm{sign}(\mathrm{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

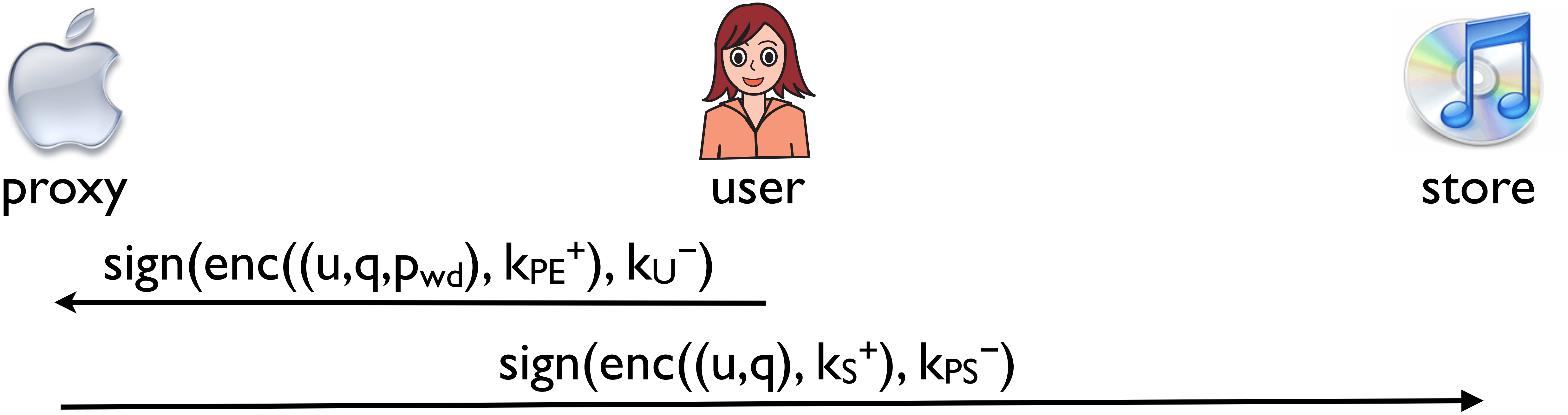
```
let user = new q;
        out(c_1, sign(enc((u,q,p_wd), k_PE^+), k_U^-)).

let proxy =
        in(c_1, x);
        let (=u, x_q, =p_wd) = dec(check(x, k_U^+), k_PE^-) in
        ...
```



$\text{sign}(\text{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$\text{sign}(\text{enc}((u,q), k_{S}^{+}), k_{PS}^{-})$

```
let user = new q;
          out(c_1, sign(enc((u,q,p_{wd}), k_{PE}^+), k_U^-)).

let proxy =
          in(c_1, x);
          let (=u, x_q, =p_{wd}) = dec(check(x, k_U^+), k_{PE}^-) in
          out(c_2, sign(enc((u,x_q), k_S^+), k_{PS}^-)).

let store = in(c_2, z);
          let (x_u,x_q) = dec(check(z, k_{PS}^+), k_S^-) in
          ...
```



proxy

user

store

sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})

sign(enc((u,q), k_{S}^{+}), k_{PS}^{-})

let user = **new** q;
    out(c_1, sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})).

let proxy =
    in(c_1, x);
    let (=u, x_q, =p_{wd}) = dec(check(x, k_{U}^{+}), k_{PE}^{-}) **in**
    out(c_2, sign(enc((u,x_q), k_{S}^{+}), k_{PS}^{-})).

let store = **in**(c_2, z);
    let (x_u,x_q) = dec(check(z, k_{PS}^{+}), k_{S}^{-}) **in**
    …

new k_{U}^{-}, k_{PE}^{-}, k_{PS}^{-}, k_{S}^{-}, p_{wd}; (user | proxy | store)

Transformation

1. **Static analysis**
2. **Process translation**

Transformation

1. **Static analysis**
2. **Process translation**

let user = ...

let proxy =
    in(c_1, x);
    let (=u, x_q, =p_{wd}) = dec(check(x, k_U^+), k_{PE}^-) **in**
    out(c_2, sign(enc((u,x_q), k_S^+), k_{PS}^-)).

let store = ...

new k_U^-, k_{PE}^-, k_{PS}^-, k_S^-, p_{wd}; (user | proxy | store)

Transformation

1. **Static analysis**
2. **Process translation**

public values: c_1, c_2, u, z, k_{PS}^{+}, k_S^{+}, k_U^{+}

secret values: x_q, k_{PE}^{-}, k_{PS}^{-}

let user = ...

let proxy =

$\quad\quad$ $\mathbf{in}(c_1, x);$

$\quad\quad$ $\mathbf{let}\ (=u, x_q, =p_{wd}) = \mathrm{dec}(\mathrm{check}(x, k_U^{+}), k_{PE}^{-})\ \mathbf{in}$

$\quad\quad$ $\underline{\mathbf{out}(c_2, \mathrm{sign}(\mathrm{enc}((u,x_q), k_S^{+}), k_{PS}^{-}))}.$

let store = ...

new k_U^{-}, k_{PE}^{-}, k_{PS}^{-}, k_S^{-}, p_{wd}; (user | proxy | store)

Transformation

1. **Static analysis**
2. **Process translation**

public values: c_1, c_2, u, z, k_{PS}^+, k_S^+, k_U^+

secret values: x_q, k_{PE}^-, k_{PS}^-

output-input data dependency:

let user = ...

let proxy =
  in(c_1, x);
  let (=u, x_q, =p_{wd}) = dec(check(x, k_U^+), k_{PE}^-) **in**
  $\underline{\textbf{out}(c_2, \text{sign}(\text{enc}((u,x_q), k_S^+), k_{PS}^-))}$.

let store = ...

new k_U^-, k_{PE}^-, k_{PS}^-, k_S^-, p_{wd}; (user | proxy | store)

Transformation

1. **Static analysis**
2. **Process translation**

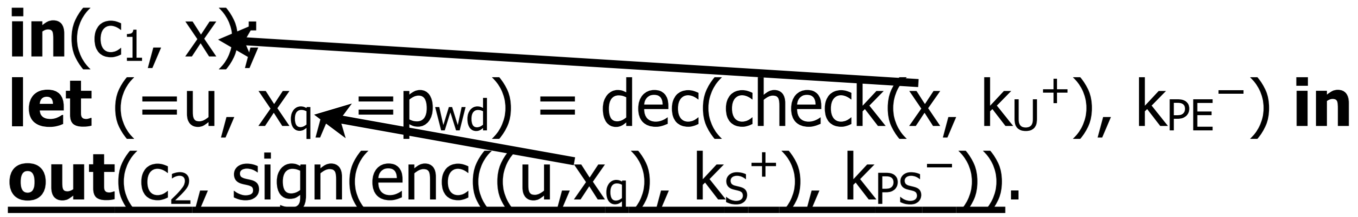
public values: c_1, c_2, u, z, k_{PS}^{+}, k_S^{+}, k_U^{+}

secret values: x_q, k_{PE}^{-}, k_{PS}^{-}

output-input data dependency:
$\mathrm{dec}(\mathrm{check}(x, k_U^{+}), k_{PE}^{-}) = (u, x_q, p_{wd})$

let user = ...

let proxy =



in(c_1, x);
let (=u, x_q, =p_{wd}) = dec(check(x, k_U^{+}), k_{PE}^{-}) **in**
$\underline{\textbf{out}(c_2, \mathrm{sign}(\mathrm{enc}((u,x_q), k_S^{+}), k_{PS}^{-}))}$.

let store = ...

new k_U^{-}, k_{PE}^{-}, k_{PS}^{-}, k_S^{-}, p_{wd}; (user | proxy | store)

Transformation

1. **Static analysis**
2. **Process translation**
   (incl. zk statement generation)

public values: c_1, c_2, u, z, k_{PS}^{+}, k_S^{+}, k_U^{+}

secret values: x_q, k_{PE}^{-}, k_{PS}^{-}

output-input data dependency:
$\mathrm{dec}(\mathrm{check}(x, k_U^{+}), k_{PE}^{-}) = (u, x_q, p_{wd})$

let user = …

let proxy′=
    in(c_1, x);
    let $(=u, x_q, =p_{wd}) = \mathrm{dec}(\mathrm{check}(x, k_U^{+}), k_{PE}^{-})$ **in**
    out(c_2, $\mathrm{sign}(\mathrm{enc}((u,x_q), k_S^{+}), k_{PS}^{-})$).

let store = …

new k_U^{-}, k_{PE}^{-}, k_{PS}^{-}, k_S^{-}, p_{wd}; (user | proxy′| store)

Transformation

1. **Static analysis**
2. **Process translation**
   (incl. zk statement generation)

public values: c_1, c_2, u, z, k_{PS}^+, k_S^+, k_U^+

secret values: x_q, k_{PE}^-, k_{PS}^-

output-input data dependency:
$dec(check(x, k_U^+), k_{PE}^-) = (u, x_q, p_{wd})$

let user = ...

let proxy′=
    in(c_1, x);
    let (=u, x_q, =p_{wd}) = dec(check(x, k_U^+), k_{PE}^-) **in**
    out(c_2, zk_S(,   ,   ; sign(enc((u,x_q), k_S^+), k_{PS}^-), x,   ).

let store = ...

new k_U^-, k_{PE}^-, k_{PS}^-, k_S^-, p_{wd}; (user | proxy′| store)

Transformation

1. **Static analysis**
2. **Process translation**
   (incl. zk statement generation)

public values: c_1, c_2, u, z, k_{PS}^+, k_S^+, k_U^+

secret values: x_q, k_{PE}^-, k_{PS}^-

output-input data dependency:
$\mathrm{dec}(\mathrm{check}(x, k_U^+), k_{PE}^-) = (u, x_q, p_{wd})$

```
stmt S = true

let user = ...

let proxy'=
        in(c₁, x);
        let (=u, xq, =pwd) = dec(check(x, kU⁺), kPE⁻) in
        out(c₂, zkS(      ,    ,      ; sign(enc((u,xq), kS⁺), kPS⁻), x,   ).

let store = ...




new kU⁻, kPE⁻, kPS⁻, kS⁻, pwd; (user | proxy'| store )
```

Transformation

1. **Static analysis**
2. **Process translation**
   (incl. zk statement generation)

public values: $c_1, c_2, u, z, k_{PS}^+, k_S^+, k_U^+$

secret values: x_q, k_{PE}^-, k_{PS}^-

output-input data dependency:
$dec(check(x, k_U^+), k_{PE}^-) = (u, x_q, p_{wd})$

stmt S = true

let user = ...

let proxy′=

  in(c_1, x);

  let (=u, x_q, =p_{wd}) = dec(check(x, k_U^+), k_{PE}^-) **in**

  out(c_2, $zk_S(\quad,\quad,\quad;\ \underline{sign(enc((u,x_q), k_S^+), k_{PS}^-)}, x,\quad)$).

let store = ...

new $k_U^-, k_{PE}^-, k_{PS}^-, k_S^-, p_{wd}$; (user | proxy′| store)

Transformation

1. **Static analysis**
2. **Process translation** (incl. zk statement generation)

public values: c_1, c_2, u, z, k_{PS}^{+}, k_S^{+}, k_U^{+}

secret values: x_q, k_{PE}^{-}, k_{PS}^{-}

output-input data dependency:
$\mathrm{dec}(\mathrm{check}(x, k_U^{+}), k_{PE}^{-}) = (u, x_q, p_{wd})$

stmt S = $\mathrm{check}(\beta_1, k_{PS}^{+}) = \mathrm{enc}((\beta_3, \alpha_2), k_S^{+})$

```
let user = ...

let proxy'=
        in($c_1$, x);
        let (=u, $x_q$, =$p_{wd}$) = $\mathrm{dec}(\mathrm{check}(x, k_U^{+}), k_{PE}^{-})$ in
        out($c_2$, $zk_S$(      ,    ,      ; $\underline{\mathrm{sign}(\mathrm{enc}((u, x_q), k_S^{+}), k_{PS}^{-})}$, x,    ).

let store = ...
```

new k_U^{-}, k_{PE}^{-}, k_{PS}^{-}, k_S^{-}, p_{wd}; (user | proxy'| store)

Transformation

1. **Static analysis**
2. **Process translation**
   (incl. zk statement generation)

public values: c_1, c_2, u, z, k_{PS}^+, k_S^+, k_U^+

secret values: x_q, k_{PE}^-, k_{PS}^-

output-input data dependency:
$dec(check(x, k_U^+), k_{PE}^-) = (u, x_q, p_{wd})$

stmt *S* = $check(\beta_1,k_{PS}^+) = enc((\beta_3,\alpha_2), k_S^+)$

let user = ...

let proxy′=
  in(c_1, x);
  let (=u, x_q, =p_{wd}) = dec(check(x, k_U^+), k_{PE}^-) **in**
  out(c_2, zk_S(,   ,   ; $\underline{sign(enc((u,x_q), k_S^+), k_{PS}^-)}$, x,   ).

let store = ...

new k_U^-, k_{PE}^-, k_{PS}^-, k_S^-, p_{wd}; (user | proxy′| store)

Transformation

1. **Static analysis**
2. **Process translation**
   (incl. zk statement generation)

public values: c_1, c_2, u, z, k_{PS}^+, k_S^+, k_U^+

secret values: x_q, k_{PE}^-, k_{PS}^-

output-input data dependency:
$dec(check(x, k_U^+), k_{PE}^-) = (u, x_q, p_{wd})$

stmt S = $check(\beta_1, k_{PS}^+) = enc((\beta_3, \alpha_2), k_S^+)$

let user = ...

let proxy'=
  in(c_1, x);
  let (=u, x_q, =p_{wd}) = dec(check(x, k_U^+), k_{PE}^-) **in**
  out(c_2, zk_S(,   ,   ; $\underline{sign(enc((u,x_q), k_S^+), k_{PS}^-)}$, x, u).

let store = ...

new k_U^-, k_{PE}^-, k_{PS}^-, k_S^-, p_{wd}; (user | proxy'| store)

Transformation

1. **Static analysis**
2. **Process translation** (incl. zk statement generation)

public values: c_1, c_2, u, z, k_{PS}^+, k_S^+, k_U^+

secret values: x_q, k_{PE}^-, k_{PS}^-

output-input data dependency:
$dec(check(x, k_U^+), k_{PE}^-) = (u, x_q, p_{wd})$

stmt S = $check(\beta_1,k_{PS}^+) = enc((\beta_3,\alpha_2), k_S^+)$

let user = ...

let proxy′=
- **in**(c_1, x);
- **let** (=u, x_q, =p_{wd}) = $dec(check(x, k_U^+), k_{PE}^-)$ **in**
- **out**(c_2, $zk_S($, x_q, ; $\underline{sign(enc((u,x_q), k_S^+), k_{PS}^-)}$, x, u).

let store = ...

new k_U^-, k_{PE}^-, k_{PS}^-, k_S^-, p_{wd}; (user | proxy′| store)

Transformation

1. **Static analysis**
2. **Process translation** (incl. zk statement generation)

public values: c_1, c_2, u, z, k_{PS}^+, k_S^+, k_U^+

secret values: x_q, k_{PE}^-, k_{PS}^-

output-input data dependency:
$dec(check(x, k_U^+), k_{PE}^-) = (u, x_q, p_{wd})$

stmt S = $check(\beta_1, k_{PS}^+) = enc((\beta_3, \alpha_2), k_S^+)$

let user = ...

let proxy′=
  in(c_1, x);
  let (=u, x_q, =p_{wd}) = dec(check(x, k_U^+), k_{PE}^-) **in**
  out(c_2, zk_S(, x_q,   ; sign(enc((u,x_q), k_S^+), k_{PS}^-)$\underline{, x,}$ u).

let store = ...

new k_U^-, k_{PE}^-, k_{PS}^-, k_S^-, p_{wd}; (user | proxy′| store)

Transformation

1. **Static analysis**
2. **Process translation**
   (incl. zk statement generation)

public values: c_1, c_2, u, z, k_{PS}^+, k_S^+, k_U^+

secret values: x_q, k_{PE}^-, k_{PS}^-

output-input data dependency:
$\underline{dec(check(x, k_U^+), k_{PE}^-) = (u, x_q, p_{wd})}$

stmt *S* = $check(\beta_1,k_{PS}^+) = enc((\beta_3,\alpha_2), k_S^+)$

let user = ...

let proxy′=
  in(c_1, x);
  let (=u, x_q, =p_{wd}) = dec(check(x, k_U^+), k_{PE}^-) **in**
  out(c_2, zk_S(, x_q, ; sign(enc((u,x_q), k_S^+), k_{PS}^-), $\underline{x,}$ u).

let store = ...

new k_U^-, k_{PE}^-, k_{PS}^-, k_S^-, p_{wd}; (user | proxy′| store)

Transformation

1. **Static analysis**
2. **Process translation**
   (incl. zk statement generation)

public values: c_1, c_2, u, z, k_{PS}^+, k_S^+, k_U^+

secret values: x_q, k_{PE}^-, k_{PS}^-

output-input data dependency:
$\underline{\text{dec(check}(x, k_U^+), k_{PE}^-) = (u, x_q, p_{wd})}$

stmt S = check(β_1,k_{PS}^+) = enc((β_3,α_2), k_S^+) ∧ dec(check(β_2, k_U^+), α_1) = (β_3,α_2,α_3)

let user = …

let proxy′=
  in(c_1, x);
  let (=u, x_q, =p_{wd}) = dec(check(x, k_U^+), k_{PE}^-) **in**
  out(c_2, zk_S(, x_q, ; sign(enc((u,x_q), k_S^+), k_{PS}^-), $\underline{x,}$ u).

let store = …

new k_U^-, k_{PE}^-, k_{PS}^-, k_S^-, p_{wd}; (user | proxy′| store)

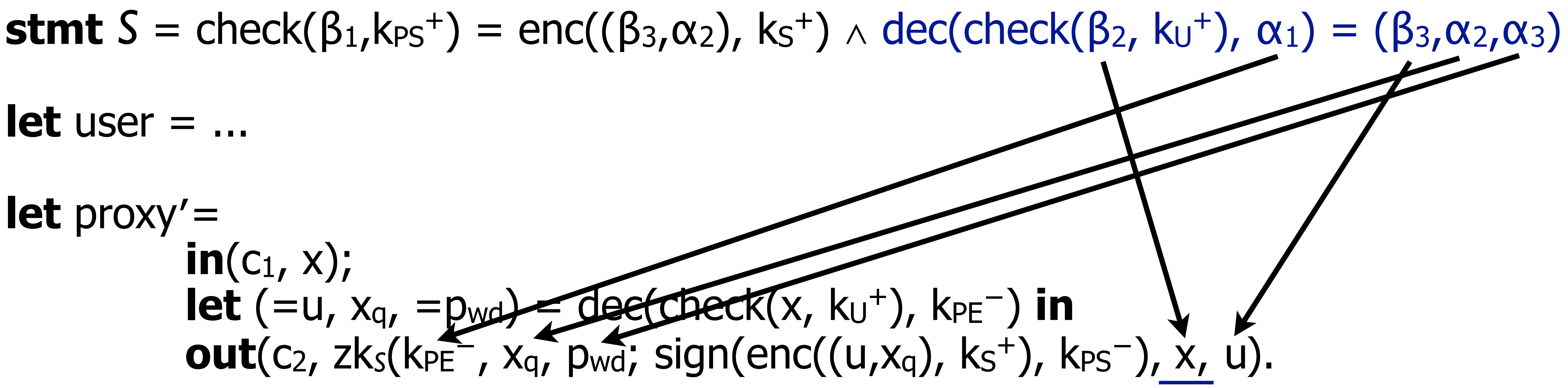
Transformation

1. **Static analysis**
2. **Process translation** (incl. zk statement generation)

public values: $c_1, c_2, u, z, k_{PS}^{+}, k_S^{+}, k_U^{+}$

secret values: $x_q, k_{PE}^{-}, k_{PS}^{-}$

output-input data dependency:
$\mathrm{dec}(\mathrm{check}(x, k_U^{+}), k_{PE}^{-}) = (u, x_q, p_{wd})$



stmt S = $\mathrm{check}(\beta_1, k_{PS}^{+}) = \mathrm{enc}((\beta_3,\alpha_2), k_S^{+}) \wedge \mathrm{dec}(\mathrm{check}(\beta_2, k_U^{+}), \alpha_1) = (\beta_3,\alpha_2,\alpha_3)$

let user = ...

let proxy′=
    in(c_1, x);
    let $(=u, x_q, =p_{wd}) = \mathrm{dec}(\mathrm{check}(x, k_U^{+}), k_{PE}^{-})$ **in**
    out(c_2, $\mathrm{zk}_S(k_{PE}^{-}, x_q, p_{wd}; \mathrm{sign}(\mathrm{enc}((u,x_q), k_S^{+}), k_{PS}^{-}), \underline{x,}\ u)$.

let store = ...

new $k_U^{-}, k_{PE}^{-}, k_{PS}^{-}, k_S^{-}, p_{wd}$; (user | proxy′| store)

Transformation

1. **Static analysis**
2. **Process translation**
   (incl. zk statement generation)

public values: $c_1, c_2, u, z, k_{PS}^+, k_S^+, k_U^+$

secret values: x_q, k_{PE}^-, k_{PS}^-

output-input data dependency:
$\mathrm{dec}(\mathrm{check}(x, k_U^+), k_{PE}^-) = (u, x_q, p_{wd})$

stmt $S = \mathrm{check}(\beta_1,k_{PS}^+) = \mathrm{enc}((\beta_3,\alpha_2), k_S^+) \wedge \mathrm{dec}(\mathrm{check}(\beta_2, k_U^+), \alpha_1) = (\beta_3,\alpha_2,\alpha_3)$

Asymmetry caused by k_S^+ being unknown to the proxy

let user = …

let proxy′=
  in(c_1, x);
  let $(=u, x_q, =p_{wd}) = \mathrm{dec}(\mathrm{check}(x, k_U^+), k_{PE}^-)$ **in**
  out$(c_2, \mathrm{zk}_S(k_{PE}^-, x_q, p_{wd}; \mathrm{sign}(\mathrm{enc}((u,x_q), k_S^+), k_{PS}^-)\underline{, x,}\ u)$.

let store = …

new $k_U^-, k_{PE}^-, k_{PS}^-, k_S^-, p_{wd}$; (user | proxy′| store)

Transformation

1. **Static analysis**
2. **Process translation**
   (incl. zk statement generation)

public values: c_1, c_2, u, z, k_{PS}^{+}, k_S^{+}, k_U^{+}

secret values: x_q, k_{PE}^{-}, k_{PS}^{-}

output-input data dependency:
$\mathrm{dec}(\mathrm{check}(x, k_U^{+}), k_{PE}^{-}) = (u, x_q, p_{wd})$

stmt $S = \mathrm{check}(\beta_1, k_{PS}^{+}) = \mathrm{enc}((\beta_3, \alpha_2), k_S^{+}) \wedge \mathrm{dec}(\mathrm{check}(\beta_2, k_U^{+}), \alpha_1) = (\beta_3, \alpha_2, \alpha_3)$

let user = ...

let proxy′=
  in(c_1, x);
  let $(=u, x_q, =p_{wd}) = \mathrm{dec}(\mathrm{check}(x, k_U^{+}), k_{PE}^{-})$ **in**
  out(c_2, $\mathrm{zk}_S(k_{PE}^{-}, x_q, p_{wd}; \mathrm{sign}(\mathrm{enc}((u, x_q), k_S^{+}), k_{PS}^{-}), x, u)$.

let store′= **in**(c_2, z);

  let $(x_u, x_q) = \mathrm{dec}(\mathrm{check}(z, k_{PS}^{+}), k_S^{-})$ **in**
  ...

new k_U^{-}, k_{PE}^{-}, k_{PS}^{-}, k_S^{-}, p_{wd}; (user | proxy′| store′)

Transformation

1. **Static analysis**
2. **Process translation** (incl. zk statement generation)

public values: $c_1, c_2, u, z, k_{PS}^{+}, k_S^{+}, k_U^{+}$

secret values: $x_q, k_{PE}^{-}, k_{PS}^{-}$

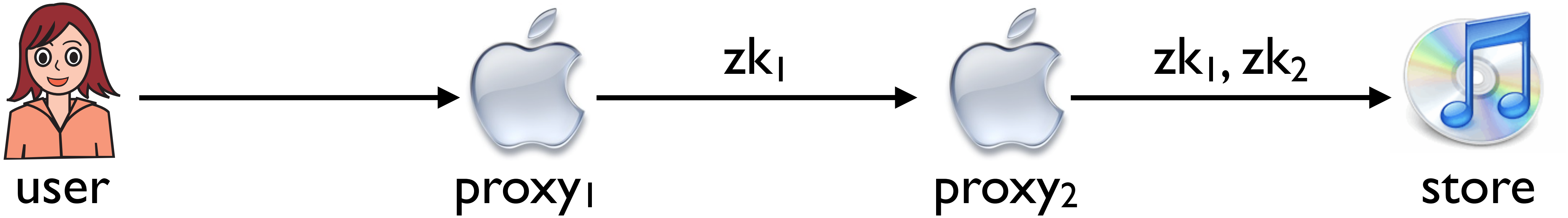
output-input data dependency:
$dec(check(x, k_U^{+}), k_{PE}^{-}) = (u, x_q, p_{wd})$

stmt $S = check(\beta_1, k_{PS}^{+}) = enc((\beta_3, \alpha_2), k_S^{+}) \wedge dec(check(\beta_2, k_U^{+}), \alpha_1) = (\beta_3, \alpha_2, \alpha_3)$

```
let user = ...

let proxy'=
        in($c_1$, x);
        let (=u, $x_q$, =$p_{wd}$) = dec(check(x, $k_U^{+}$), $k_{PE}^{-}$) in
        out($c_2$, $zk_S(k_{PE}^{-}, x_q, p_{wd}; sign(enc((u,x_q), k_S^{+}), k_{PS}^{-}), x, u)$).

let store'= in($c_2$, z);
        let ($\beta_1,\beta_2,\beta_3$) = $ver_S$(z) in
        let ($x_u$,$x_q$) = dec(check($\beta_1$, $k_{PS}^{+}$), $k_S^{-}$) in
        ...


new $k_U^{-}$, $k_{PE}^{-}$, $k_{PS}^{-}$, $k_S^{-}$, $p_{wd}$; (user | proxy'| store')
```

Further complications

- Forwarding zero-knowledge proofs
  - Ensure correct behavior of all protocol participants



- Symmetric encryption (prove identity using ZK)
- Transforming types (more on types later)

Enhanced type system for zero-knowledge

Translation validation

[Pnueli et al., TACAS ’98]

- Accepted technique for increasing user’s confidence in complex transformations (e.g. compiler)
  - + prevents incorrect code from being run
  - + strong guarantees if validation succeeds
  - + without the need to prove transformation always correct

Translation validation

[Pnueli et al., TACAS '98]

- Accepted technique for increasing user's confidence in complex transformations (e.g. compiler)
  - + prevents incorrect code from being run
  - + strong guarantees if validation succeeds
  - + without the need to prove transformation always correct
  - + changing transformation is very easy (e.g. optimizing)

Translation validation

[Pnueli et al., TACAS '98]

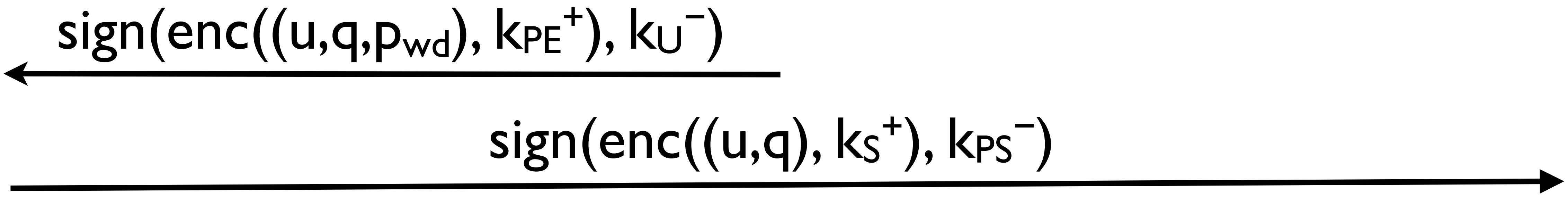
- Accepted technique for increasing user's confidence in complex transformations (e.g. compiler)
  - + prevents incorrect code from being run
  - + strong guarantees if validation succeeds
  - + without the need to prove transformation always correct
  - + changing transformation is very easy (e.g. optimizing)
  - – guarantees only for a specific policy (e.g. authorization policy)
  - – no guarantees if validation fails

Translation validation

[Pnueli et al., TACAS '98]

- Accepted technique for increasing user's confidence in complex transformations (e.g. compiler)
  - + prevents incorrect code from being run
  - + strong guarantees if validation succeeds
  - + without the need to prove transformation always correct
  - + changing transformation is very easy (e.g. optimizing)
  - – guarantees only for a specific policy (e.g. authorization policy)
  - – no guarantees if validation fails
- We use type system for validation [Backes, Hrițcu & Maffei, CCS '08] [Fournet, Gordon & Maffeis, CSF '07]
  - Now extended to handle security despite compromise (added union and intersection types and a logical charact. of kinding)

Authorization policy



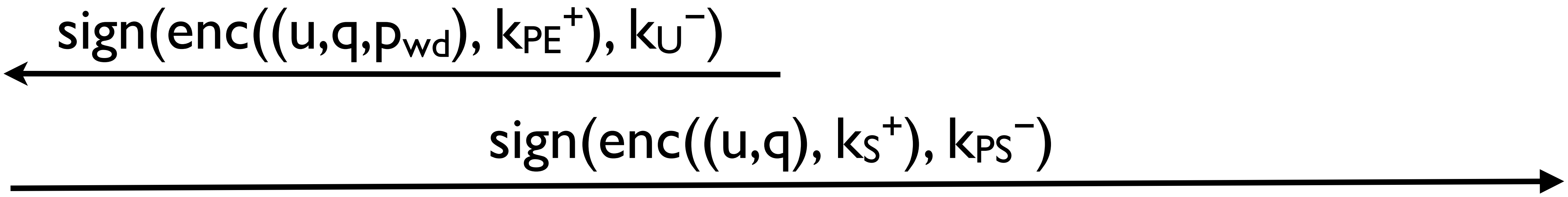
$\text{sign}(\text{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$\text{sign}(\text{enc}((u,q), k_{S}^{+}), k_{PS}^{-})$

```
let user = new q;
            out(c1, sign(enc((u,q,p_wd), k_PE^+), k_U^-)).

let proxy =
            in(c1, x);
            let (=u, x_q, =p_wd) = dec(check(x, k_U^+), k_PE^-) in
            out(c2, sign(enc((u,x_q), k_S^+), k_PS^-)).

let store = in(c2, z);
            let (x_u,x_q) = dec(check(z, k_PS^+), k_S^-) in
            ...
```

```
new k_U^-, k_PE^-, k_PS^-, k_S^-, p_wd; (user | proxy | store)
```

Authorization policy

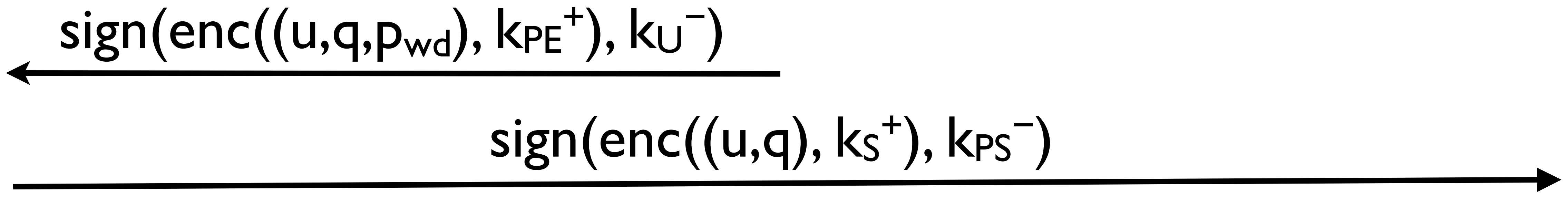


$\mathrm{sign}(\mathrm{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$\mathrm{sign}(\mathrm{enc}((u,q), k_{S}^{+}), k_{PS}^{-})$

```
let user = new q; assume Request(u, q) |
          out($c_1$, sign(enc((u,q,$p_{wd}$), $k_{PE}^{+}$), $k_U^{-}$)).

let proxy = assume Registered(u) |
          in($c_1$, x);
          let (=u, $x_q$, =$p_{wd}$) = dec(check(x, $k_U^{+}$), $k_{PE}^{-}$) in
          out($c_2$, sign(enc((u,$x_q$), $k_S^{+}$), $k_{PS}^{-}$)).

let store = in($c_2$, z);
          let ($x_u$,$x_q$) = dec(check(z, $k_{PS}^{+}$), $k_S^{-}$) in
          ...

new $k_U^{-}$, $k_{PE}^{-}$, $k_{PS}^{-}$, $k_S^{-}$, $p_{wd}$; (user | proxy | store)
```

Authorization policy



$\mathrm{sign}(\mathrm{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$\mathrm{sign}(\mathrm{enc}((u,q), k_{S}^{+}), k_{PS}^{-})$

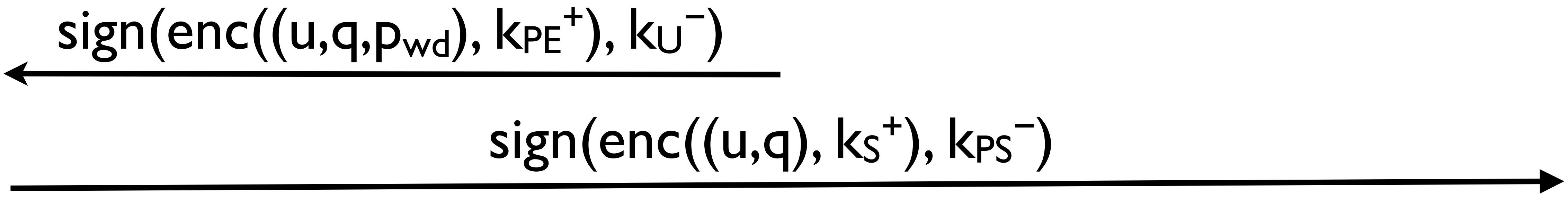
let user = **new** q; **assume** Request(u, q) |
    out(c_1, $\mathrm{sign}(\mathrm{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$).

let proxy = **assume** Registered(u) |
    in(c_1, x);
    let (=u, x_q, =p_{wd}) = $\mathrm{dec}(\mathrm{check}(x, k_{U}^{+}), k_{PE}^{-})$ **in**
    out(c_2, $\mathrm{sign}(\mathrm{enc}((u,x_q), k_{S}^{+}), k_{PS}^{-})$).

let store = **in**(c_2, z);
    let (x_u,x_q) = $\mathrm{dec}(\mathrm{check}(z, k_{PS}^{+}), k_{S}^{-})$ **in**
    assert Authenticate(x_u,x_q).

new k_{U}^{-}, k_{PE}^{-}, k_{PS}^{-}, k_{S}^{-}, p_{wd}; (user | proxy | store)

Authorization policy


$\text{sign}(\text{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$
$\text{sign}(\text{enc}((u,q), k_{S}^{+}), k_{PS}^{-})$

let user = **new** q; **assume** Request(u, q) |
out(c_1, sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})).

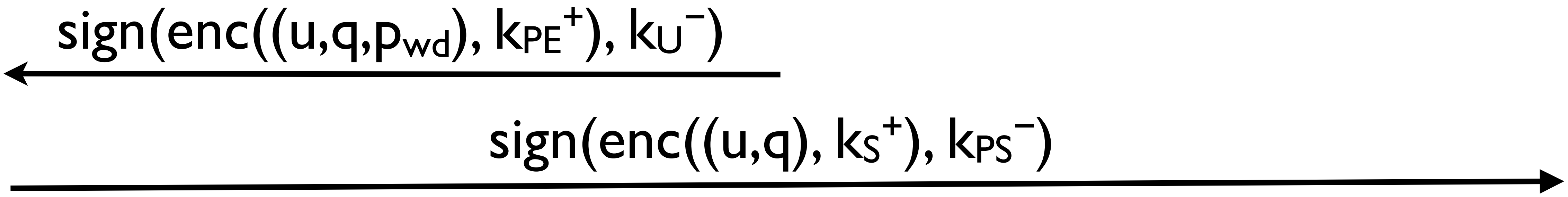
let proxy = **assume** Registered(u) |
in(c_1, x);
let (=u, x_q, =p_{wd}) = dec(check(x, k
out(c_2, sign(enc((u,x_q), k_S^{+}), k

let store = **in**(c_2, z);
let (x_u,x_q) = dec(check(z, k_{PS}^{+}
assert Authenticate(x_u,x_q).

assert succeeds only if
Authenticate(x_u,x_q) holds

new k_U^{-}, k_{PE}^{-}, k_{PS}^{-}, k_S^{-}, p_{wd}; (user | proxy | store)

Authorization policy



$\mathrm{sign}(\mathrm{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$\mathrm{sign}(\mathrm{enc}((u,q), k_{S}^{+}), k_{PS}^{-})$

let user = **new** q; **assume** Request(u, q) |
    out(c_1, $\mathrm{sign}(\mathrm{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$).

let proxy = **assume** Registered(u) |
    in(c_1, x);
    let (=u, x_q, =p_{wd}) = $\mathrm{dec}(\mathrm{check}(x, k_{U}^{+}), k_{PE}^{-})$ **in**
    out(c_2, $\mathrm{sign}(\mathrm{enc}((u,x_q), k_{S}^{+}), k_{PS}^{-})$).

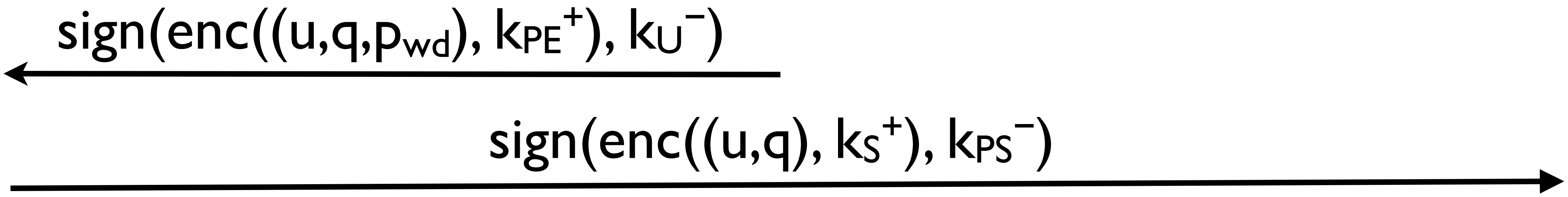
let store = **in**(c_2, z);
    let (x_u,x_q) = $\mathrm{dec}(\mathrm{check}(z, k_{PS}^{+}), k_{S}^{-})$ **in**
    assert Authenticate(x_u,x_q).

formula in some authorization logic (here FOL)

let policy = **assume** $\forall$ u, q. (Request(u, q) $\wedge$ Registered(u) $\Rightarrow$ Authenticate(u, q))

new k_{U}^{-}, k_{PE}^{-}, k_{PS}^{-}, k_{S}^{-}, p_{wd}; (user | proxy | store | policy)

Authorization policy


$\mathrm{sign}(\mathrm{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$
$\mathrm{sign}(\mathrm{enc}((u,q), k_{S}^{+}), k_{PS}^{-})$

let user = **new** q; **assume** Request(u, q) |
    out(c_1, sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})).

let proxy = **assume** Registered(u) |
    in(c_1, x);
    let (=u, x_q, =p_{wd}) = dec(check(x, k_U^{+}), k
    out(c_2, sign(enc((u,x_q), k_S^{+}), k_P

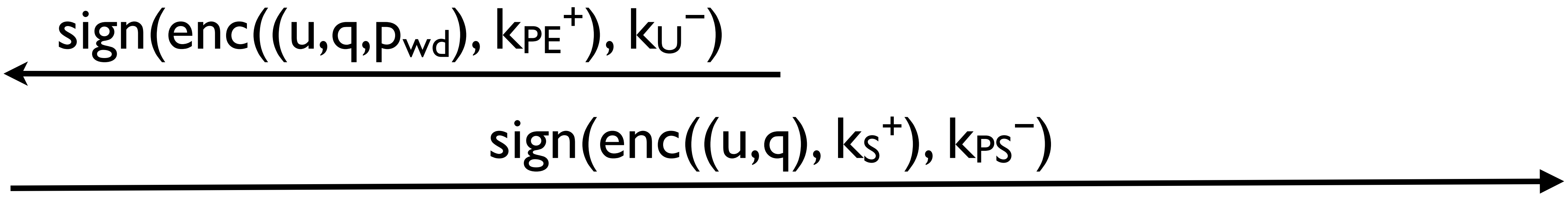
let store = **in**(c_2, z);
    let (x_u,x_q) = dec(check(z, k_{PS}^{+})
    assert Authenticate(x_u,x_q).

assert succeeds only if
Authenticate(x_u,x_q) holds

let policy = **assume** ∀ u, q. (Request(u, q) ∧ Registered(u) ⇒ Authenticate(u, q))

new k_U^{-}, k_{PE}^{-}, k_{PS}^{-}, k_S^{-}, p_{wd}; (user | proxy | store | policy)

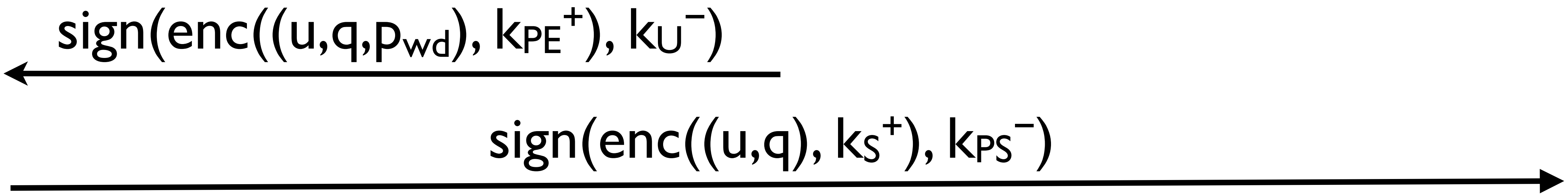
Authorization policy


$sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$
$sign(enc((u,q), k_{S}^{+}), k_{PS}^{-})$

```
let user = new q; assume Request(u, q) |
        out($c_1$, sign(enc((u,q,$p_{wd}$), $k_{PE}^{+}$), $k_U^{-}$)).

let proxy = assume Registered(u) |
        in($c_1$, x);
        let (=u, $x_q$, =$p_{wd}$) = dec(check(x, $k_U^{+}$), $k_{PE}^{-}$) in
        out($c_2$, sign(enc((u,$x_q$), $k_S^{+}$), $k_{PS}^{-}$)).

let store = in($c_2$, z);
        let ($x_u$,$x_q$) = dec(check(z, $k_{PS}^{+}$), $k_S^{-}$) in
        assert Authenticate($x_u$,$x_q$).

let policy = assume $\forall$ u, q. (Request(u, q) $\wedge$ Registered(u) $\Rightarrow$ Authenticate(u, q))

new $k_U^{-}$, $k_{PE}^{-}$, $k_{PS}^{-}$, $k_S^{-}$, $p_{wd}$; (user | proxy | store | policy)
```

Authenticate(x_u,x_q) holds only if Request(x_u, x_q) $\wedge$ Registered(x_u) holds (since Authenticate only appears here)

Authorization policy

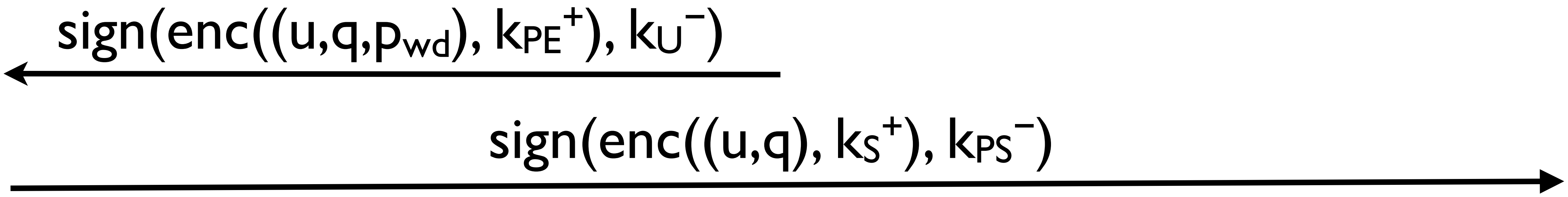


$\mathrm{sign}(\mathrm{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$
$\mathrm{sign}(\mathrm{enc}((u,q), k_{S}^{+}), k_{PS}^{-})$

```
let user = new q; assume Request(u, q) |
           out(c_1, sign(enc((u,q,p_wd), k_PE^+), k_U^-)).

let proxy = assume Registered(u) |
           in(c_1, x);
           let (=u, x_q, =p_wd) = dec(check(x, k_U^+), k_PE^-) in
           out(c_2, sign(enc((u,x_q), k_S^+), k_PS^-)).

let store = in(c_2, z);
           let (x_u,x_q) = dec(check(z, k_PS^+), k_S^-) in
           assert Authenticate(x_u,x_q).

let policy = assume ∀ u, q. (Request(u, q) ∧ Registered(u) ⇒ Authenticate(u, q))

new k_U^-, k_PE^-, k_PS^-, k_S^-, p_wd; (user | proxy | store | policy)
```

$\mathrm{Request}(x_u, x_q)$ holds only if the user has indeed issued a request

Authorization policy



$\mathrm{sign}(\mathrm{enc}((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

$\mathrm{sign}(\mathrm{enc}((u,q), k_{S}^{+}), k_{PS}^{-})$

let user = **new** q; **assume** Request(u, q) |
out(c_1, sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})).

let proxy = **assume** Registered(u) |
in(c_1, x);
let (=u, x_q, =p_{wd}) = dec(check(x, k
out(c_2, sign(enc((u,x_q), k_{S}^{+}), k_{PS}^{-}

let store = **in**(c_2, z);
let (x_u,x_q) = dec(check(z, k_{PS}^{+}), k_{S}^{-}
assert Authenticate(x_u,x_q).

This policy enforces that the store authenticates the user only if a registered user has indeed issued a request

let policy = **assume** ∀ u, q. (Request(u, q) ∧ Registered(u) ⇒ Authenticate(u, q))

new k_{U}^{-}, k_{PE}^{-}, k_{PS}^{-}, k_{S}^{-}, p_{wd}; (user | proxy | store | policy)

Security properties (informal)

- **Robust safety:** in all executions all asserts succeed (i.e. asserts are logically entailed by the active assumes)

Security properties (informal)

- **Robust safety:** in all executions all asserts succeed (i.e. asserts are logically entailed by the active assumes)
    - in the presence of arbitrary DY attacker

Security properties (informal)

- **Robust safety:** in all executions all asserts succeed (i.e. asserts are logically entailed by the active assumes)
    - in the presence of arbitrary DY attacker
    - but where all participants are assumed honest

Security properties (informal)

- **Robust safety:** in all executions all asserts succeed (i.e. asserts are logically entailed by the active assumes)
  - in the presence of arbitrary DY attacker
  - but where all participants are assumed honest

- **Safety despite compromise:** "An invalid authorization decision [...] should only arise if participants on which the decision logically depends are compromised."

"Hence, the impact of partial compromise should be apparent from the policy, without study of the code"

[Fournet, Gordon & Maffeis, CSF '07]

proxy

```
let user = new q; assume Request(u, q) |
           out(c_1, sign(enc((u,q,p_wd), k_PE^+), k_U^-)).

let proxy = assume Registered(u) |
           in(c_1, x);
           let (=u, x_q, =p_wd) = dec(check(x, k_U^+), k_PE^-) in
           out(c_2, sign(enc((u,x_q), k_S^+), k_PS^-)).
let store = ...



let policy = assume ∀ u, q. (Request(u, q) ∧ Registered(u) ⇒ Authenticate(u, q))
```

proxy

```
let user = new q; assume Request(u, q) |
           out(c$_1$, sign(enc((u,q,p$_{wd}$), k$_{PE}^{+}$), k$_{U}^{-}$)).

let proxy = assume Registered(u) |
           in(c$_1$, x);
           let (=u, x$_q$, =p$_{wd}$) = dec(check(x, k$_{U}^{+}$), k$_{PE}^{-}$) in
           out(c$_2$, sign(enc((u,x$_q$), k$_{S}^{+}$), k$_{PS}^{-}$)).
let store = ...


let policy = assume $\forall$ u, q. (Request(u, q) $\wedge$ Registered(u) $\Rightarrow$ Authenticate(u, q)) |
             assume Compromised(u) $\Rightarrow$ $\forall$ q. Request(u, q) |
             assume Compromised(p) $\Rightarrow$ $\forall$ u. Registered(u)
```

security
despite compromise

Compromising the proxy

proxy

```
let user = new q; assume Request(u, q) |
        out($c_1$, sign(enc((u,q,$p_{wd}$), $k_{PE}^{+}$), $k_U^{-}$)).

let proxy = assume Registered(u) |
        in($c_1$, x);
        let (=u, $x_q$, =$p_{wd}$) = dec(check(x, $k_U^{+}$), $k_{PE}^{-}$) in
        out($c_2$, sign(enc((u,$x_q$), $k_S^{+}$), $k_{PS}^{-}$)).

let store = ...


let policy = assume $\forall$ u, q. (Request(u, q) $\wedge$ Registered(u) $\Rightarrow$ Authenticate(u, q)) |
        assume Compromised(u) $\Rightarrow$ $\forall$ q. Request(u, q) |
        assume Compromised(p) $\Rightarrow$ $\forall$ u. Registered(u)
        assume $\neg$Compromised(u) $\wedge$ Compromised(p) $\wedge$ $\neg$Compromised(s)
```

Compromising the proxy

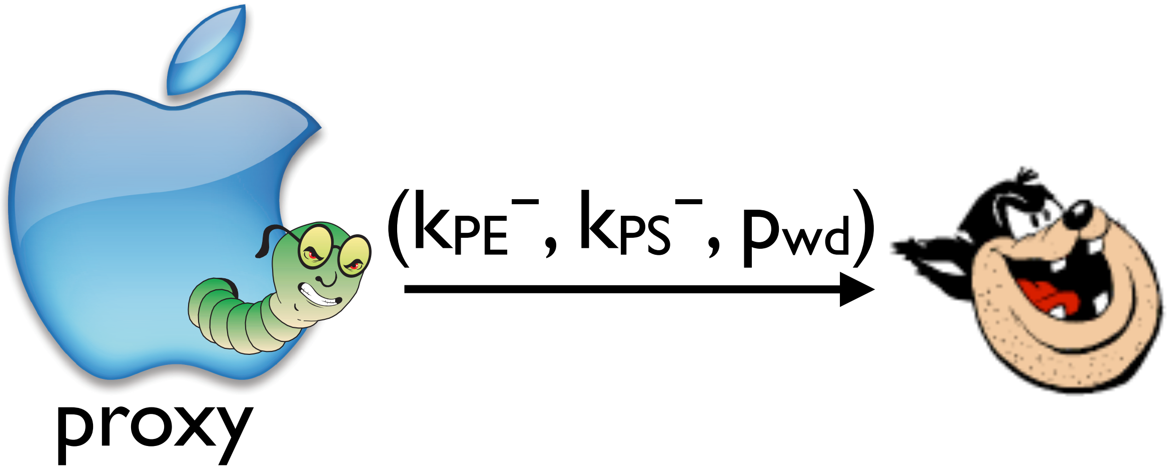
let user = **new** q; **assume** Request(u, q) |
    out(c_1, sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_U^{-})).

let proxy = **assume** Registered(u) |
    in(c_1, x);
    let (=u, x_q, =p_{wd}) = dec(check(x, k_U^{+}), k_{PE}^{-}) **in**
    out(c_2, sign(enc((u,x_q), k_S^{+}), k_{PS}^{-})).

let store = ...

let policy = **assume** ∀ u, q. (Request(u, q) ~~∧ Registered(u)~~ ⇒ Authenticate(u, q)) |
    assume Compromised(u) ⇒ ∀ q. Request(u, q) |
    assume Compromised(p) ⇒ ∀ u. Registered(u)
    assume ¬Compromised(u) ∧ Compromised(p) ∧ ¬Compromised(s)

Compromising the proxy



$(k_{PE}^{-}, k_{PS}^{-}, p_{wd})$

proxy

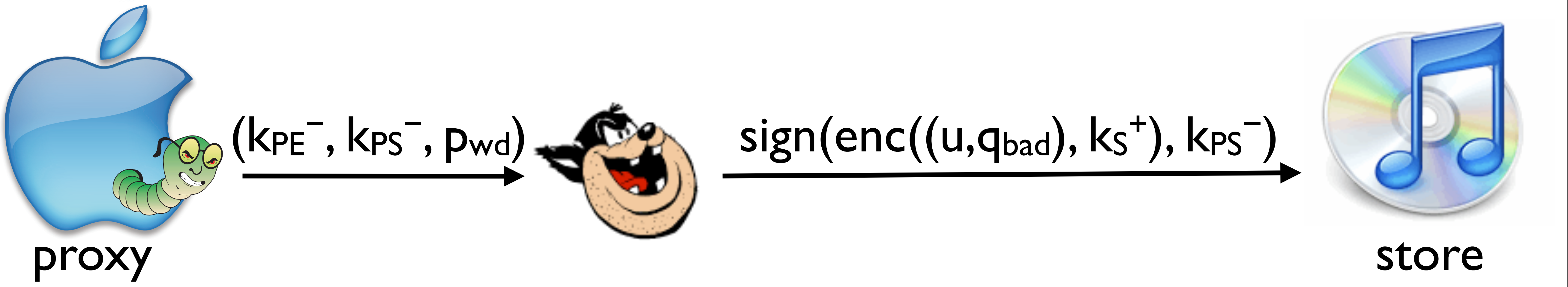
let user = **new** q; **assume** Request(u, q) |
out(c_1, sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_U^{-})).

let bad_proxy = **out**(c_{pub}, (k_{PE}^{-}, k_{PS}^{-}, p_{wd})).

let store = ...

let policy = **assume** ∀ u, q. (Request(u, q) ~~∧ Registered(u)~~ ⇒ Authenticate(u, q)) |
assume Compromised(u) ⇒ ∀ q. Request(u, q) |
assume Compromised(p) ⇒ ∀ u. Registered(u)
assume ¬Compromised(u) ∧ Compromised(p) ∧ ¬Compromised(s)

Compromising the proxy



$(k_{PE}^{-}, k_{PS}^{-}, p_{wd})$

$\text{sign}(\text{enc}((u,q_{bad}), k_S^{+}), k_{PS}^{-})$

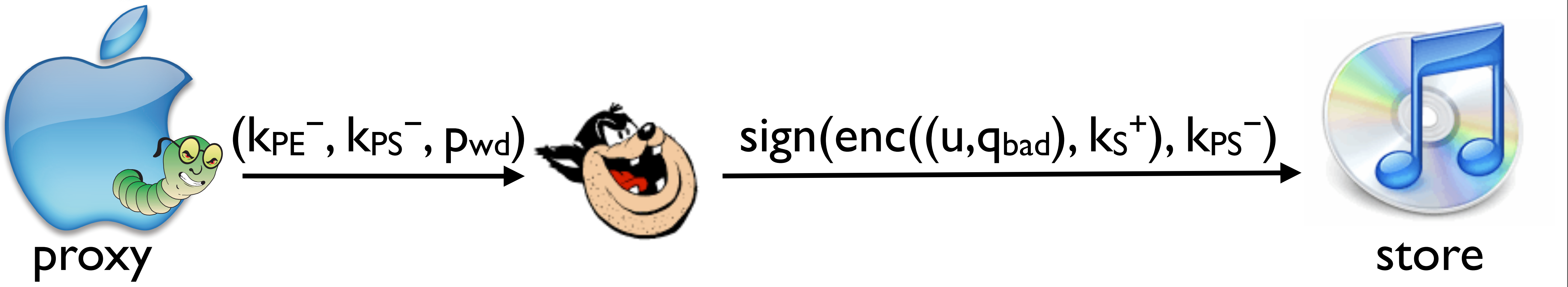
let user = **new** q; **assume** Request(u, q) |
out(c_1, sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_U^{-})).

let bad_proxy = **out**(c_{pub}, (k_{PE}^{-}, k_{PS}^{-}, p_{wd})).

let store = ...

let policy = **assume** ∀ u, q. (Request(u, q) ~~∧ Registered(u)~~ ⇒ Authenticate(u, q)) |
assume Compromised(u) ⇒ ∀ q. Request(u, q) |
assume Compromised(p) ⇒ ∀ u. Registered(u)
assume ¬Compromised(u) ∧ Compromised(p) ∧ ¬Compromised(s)

Compromising the proxy



(<i></i>k_{PE}^-, k_{PS}^-, p_{wd})

sign(enc((u,q_{bad}), k_S^+), k_{PS}^-)

proxy

store

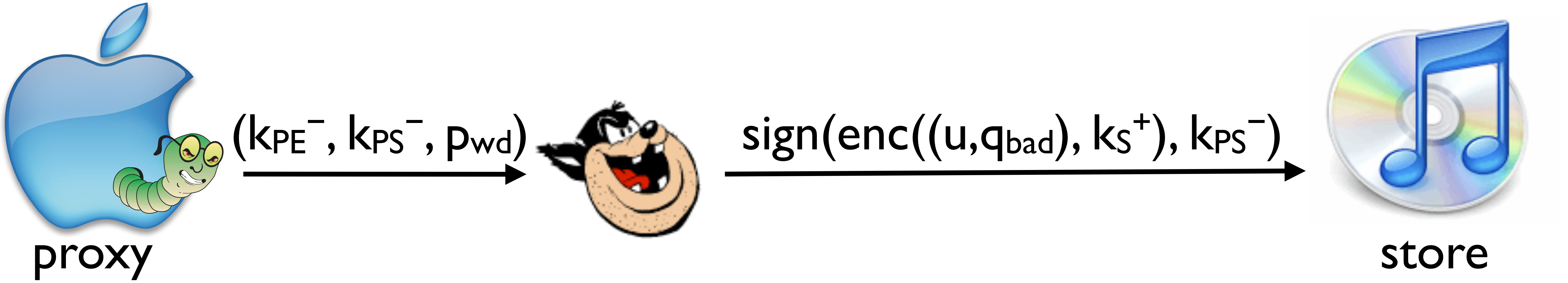
let user = **new** q; **assume** Request(u, q) |
out(c_1, sign(enc((u,q,p_{wd}), k_{PE}^+), k_U^-)).

let bad_proxy = **out**(c_{pub}, (k_{PE}^-, k_{PS}^-, p_{wd})).

let store = **in**(c_2, z);
let (x_u,x_q) = dec(check(z, k_{PS}^+), k_S^-) **in**
assert Authenticate(x_u,x_q).

let policy = **assume** ∀ u, q. (Request(u, q) ~~∧ Registered(u)~~ ⇒ Authenticate(u, q)) |
assume Compromised(u) ⇒ ∀ q. Request(u, q) |
assume Compromised(p) ⇒ ∀ u. Registered(u)
assume ¬Compromised(u) ∧ Compromised(p) ∧ ¬Compromised(s)

Compromising the proxy


$(k_{PE}^-, k_{PS}^-, p_{wd})$
$\mathrm{sign}(\mathrm{enc}((u,q_{bad}), k_S^+), k_{PS}^-)$
proxy
store

let user = **new** q; **assume** Request(u, q) |
out(c_1, sign(enc((u,q,p_{wd}), k_{PE}^+), k_U^-)).

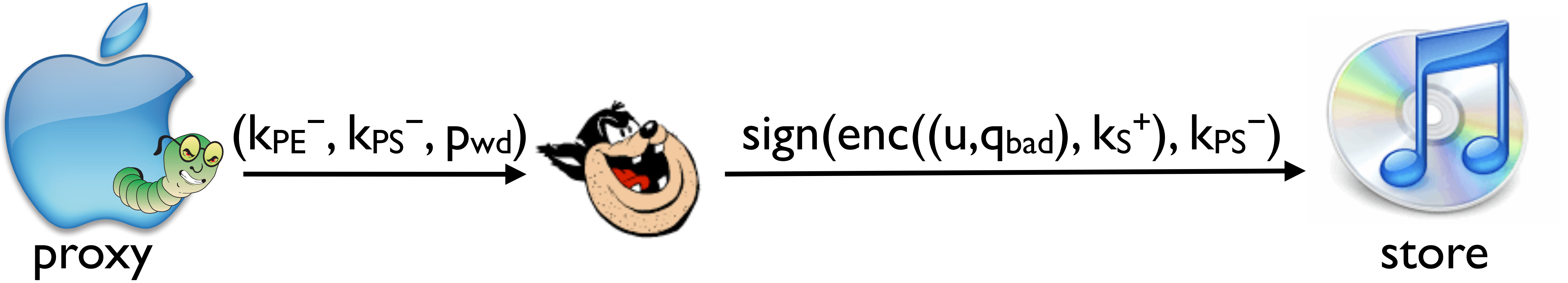
let bad_proxy = **out**(c_{pub}, (k_{PE}^-, k_{PS}^-, p_{wd})).

let store = **in**(c_2, z);
let (x_u,x_q) = dec(check(z, k_{PS}^+));
assert Authenticate(x_u,x_q).

Authenticate(u, q_{bad}) is not entailed since user never requested anything

let policy = **assume** ∀ u, q. (Request(u, q) ~~∧ Registered(u)~~ ⇒ Authenticate(u, q)) |
assume Compromised(u) ⇒ ∀ q. Request(u, q) |
assume Compromised(p) ⇒ ∀ u. Registered(u)
assume ¬Compromised(u) ∧ Compromised(p) ∧ ¬Compromised(s)

Compromising the proxy



$(k_{PE}^-, k_{PS}^-, p_{wd})$

$\mathrm{sign}(\mathrm{enc}((u,q_{bad}), k_S^+), k_{PS}^-)$

proxy

store

let user = **new** q; **assume** Request(u, q) |
  out(c_1, sign(enc((u,q,p_{wd}), k_{PE}^+), k_U^-)).

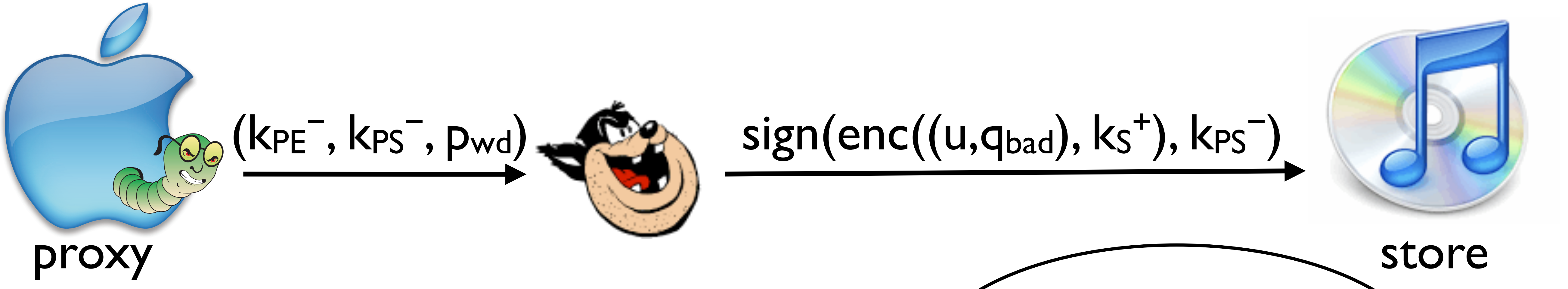
let bad_proxy = **out**(c_{pub}, (k_{PE}^-, k_{PS}^-, p_{wd})).

let store = **in**(c_2, z);
  let (x_u,x_q) = dec(check(z, k_{PS}^+
  assert Authenticate(x_u,x_q).

assert fails, so protocol is not secure if the proxy is compromised

let policy = **assume** ∀ u, q. (Request(u, q) ~~∧ Registered(u)~~ ⇒ Authenticate(u, q)) |
  assume Compromised(u) ⇒ ∀ q. Request(u, q) |
  assume Compromised(p) ⇒ ∀ u. Registered(u)
  assume ¬Compromised(u) ∧ Compromised(p) ∧ ¬Compromised(s)

Compromising the proxy


$(k_{PE}^{-}, k_{PS}^{-}, p_{wd})$
sign(enc((u,q_{bad}), k_S^{+}), k_{PS}^{-})
proxy
store

let user = **new** q; **assume** Request(u, q) |
out(c_1, sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_U^{-})).

let bad_proxy = **out**(c_{pub}, (k_{PE}^{-}, k_{PS}^{-}, p_{wd})).

Our transformation fixes this

assert fails, so protocol is not secure if the proxy is compromised

let store = **in**(c_2, z);
let (x_u,x_q) = dec(check(z, k_{PS}^{+}
assert Authenticate(x_u,x_q).

let policy = **assume** ∀ u, q. (Request(u, q) ~~∧ Registered(u)~~ ⇒ Authenticate(u, q)) |

assume Compromised(u) ⇒ ∀ q. Request(u, q) |

assume Compromised(p) ⇒ ∀ u. Registered(u)

assume ¬Compromised(u) ∧ Compromised(p) ∧ ¬Compromised(s)

```
let user = new q          ; assume Request(u, q) |
           out($c_1$, sign(enc((u,q,$p_{wd}$), $k_{PE}^{+}$), $k_U^{-}$)).

let proxy' = assume Registered(u) |
           in($c_1$, x);
           let (=u, $x_q$, =$p_{wd}$) = dec(check(x, $k_U^{+}$), $k_{PE}^{-}$) in
           out($c_2$, $zk_S$($k_{PE}^{-}$, $x_q$, $p_{wd}$; sign(enc((u,$x_q$), $k_S^{+}$), $k_{PS}^{-}$), x, u).

let store' = in($c_2$, z);
           let ($\beta_1$,$\beta_2$,$\beta_3$) = $ver_S$(z) in
           let ($x_u$,$x_q$) = dec(check($\beta_1$, $k_{PS}^{+}$), $k_S^{-}$) in
           assert Authenticate($x_u$,$x_q$).

let policy = assume $\forall$ u, q. (Request(u, q) $\wedge$ Registered(u) $\Rightarrow$ Authenticate(u, q)) ...


new $k_U^{-}$                    ;        new $k_{PS}^{-}$                         ;
new $k_{PE}^{-}$              ;           new $k_S^{-}$                       ;
new $p_{wd}$                  ;
(user | proxy'       | store' | policy)
```

let user = **new** q ; **assume** Request(u, q) |
  out(c_1, sign(enc($\underline{(u,q,p_{wd})}$, k_{PE}^{+}), k_U^{-})).

let proxy′ = **assume** Registered(u) |
  in(c_1, x);
  let ($=u$, x_q, $=p_{wd}$) = dec(check(x, k_U^{+}), k_{PE}^{-}) **in**
  out(c_2, zk$_S$(k_{PE}^{-}, x_q, p_{wd}; sign(enc((u,x_q), k_S^{+}), k_{PS}^{-}), x, u).

let store′ = **in**(c_2, z);
  let (β_1,β_2,β_3) = ver$_S$(z) **in**
  let (x_u,x_q) = dec(check(β_1, k_{PS}^{+}), k_S^{-}) **in**
  assert Authenticate(x_u,x_q).

let policy = **assume** $\forall$ u, q. (Request(u, q) $\wedge$ Registered(u) $\Rightarrow$ Authenticate(u, q)) ...

new k_U^{-} ; **new** k_{PS}^{-} ;
new k_{PE}^{-} ; **new** k_S^{-} ;
new p_{wd} ;
(user | proxy′ | store′ | policy)

Public message - can be sent to the attacker

let user = **new** q : Un; **assume** Request(u, q) |
out(c_1, sign(enc($\underline{(u,q,p_{wd})}$, k_{PE}^{+}), k_U^{-})).

let proxy′ = **assume** Registered(u) |
in(c_1, x);
let (=u, x_q, =p_{wd}) = dec(check(x, k_U^{+}), k_{PE}^{-}) **in**
out(c_2, zk_S(k_{PE}^{-}, x_q, p_{wd}; sign(enc((u,x_q), k_S^{+}), k_{PS}^{-}), x, u).

let store′ = **in**(c_2, z);
let (β_1,β_2,β_3) = ver_S(z) **in**
let (x_u,x_q) = dec(check(β_1, k_{PS}^{+}), k_S^{-}) **in**
assert Authenticate(x_u,x_q).

let policy = **assume** $\forall$ u, q. (Request(u, q) $\wedge$ Registered(u) $\Rightarrow$ Authenticate(u, q)) ...

new k_U^{-} ; **new** k_{PS}^{-} ;
new k_{PE}^{-} ; **new** k_S^{-} ;
new p_{wd} ;
(user | proxy′ | store′ | policy)

let user = **new** q : Un; **assume** Request(u, q) |
    out(c_1, sign(enc($\underline{(u,q,p_{wd})}$, k_{PE}^+), k_U^-)).

let proxy′ = **assume** Registered(u) |
    in(c_1, x);
    let (=u, x_q, =p_{wd}) = dec(check(x, k_U^+), k_{PE}^-) **in**
    out(c_2, zk_S(k_{PE}^-, x_q, p_{wd}; sign(enc((u,x_q), k_S^+), k_{PS}^-), x, u).

let store′ = **in**(c_2, z);
    let (β_1,β_2,β_3) = ver_S(z) **in**
    let (x_u,x_q) = dec(check(β_1, k_{PS}^+), k_S^-) **in**
    assert Authenticate(x_u,x_q).

let policy = **assume**) ∧ Registered(u) ⇒ Authenticate(u, q)) ...

Secret message - not known to the attacker

new k_U^- $_{PS}^-$;
new k_{PE}^- , **ew** k_S^- ;
new p_{wd} : Private ;
(user | proxy′ | store′ | policy)

typedef T_1 = Triple(x_u : Un, {x_q : Un | Request(x_u, x_q)}, x_p : Private)

let user = **new** q : Un; **assume** Request(u, q) |
out(c_1, sign(enc($\underline{(u,q,p_{wd})}$, k_{PE}^+), k_U^-)).

let proxy′ = **assume** Registered(u) |
in(c_1, x);
let (=u, x_q, =p_{wd}) = dec(check(x, k_U^+), k_{PE}^-) **in**
out(c_2, zk_S(k_{PE}^-, x_q, p_{wd}; sign(enc((u,x_q), k_S^+), k_{PS}^-), x, u).

let store′ = **in**(c_2, z);
let (β_1,β_2,β_3) = ver_S(z) **in**
let (x_u,x_q) = dec(check(β_1, k_{PS}^+), k_S^-) **in**
assert Authenticate(x_u,x_q).

let policy = **assume** $\forall$ u, q. (Request(u, q) $\wedge$ Registered(u) $\Rightarrow$ Authenticate(u, q)) ...

new k_U^- ;
new k_{PE}^- ;
new p_{wd} : Private ;

new k_{PS}^- ;
new k_S^- ;

(user | proxy′ | store′ | policy)

typedef T_1 = Triple($\underline{x_u : \text{Un}, \{x_q : \text{Un} \mid \text{Request}(x_u, x_q)\}}$, x_p : Private)

Refinement type - conveys logical formula

let user = **new** q : Un; **assume** Request(u, q) |
out(c_1, sign(enc($\underline{(u,q,p_{wd})}$, k_{PE}^+), k_U^-)).

let proxy' = **assume** Registered(u) |
in(c_1, x);
let (=u, x_q, =p_{wd}) = dec(check(x, k_U^+), k_{PE}^-) **in**
out(c_2, zk_S(k_{PE}^-, x_q, p_{wd}; sign(enc((u,x_q), k_S^+), k_{PS}^-), x, u).

let store' = **in**(c_2, z);
let (β_1,β_2,β_3) = ver_S(z) **in**
let (x_u,x_q) = dec(check(β_1, k_{PS}^+), k_S^-) **in**
assert Authenticate(x_u,x_q).

let policy = **assume** $\forall$ u, q. (Request(u, q) $\wedge$ Registered(u) $\Rightarrow$ Authenticate(u, q)) ...

new k_U^- ; **new** k_{PS}^- ;
new k_{PE}^- ; **new** k_S^- ;
new p_{wd} : Private ;
(user | proxy' | store' | policy)

typedef T_1 = Triple(x_u : Un, {x_q : Un | Request(x_u, x_q)}, x_p : Private)

let user = **new** q : Un; **assume** Request(u, q) |
out(c_1, sign(enc((u,q,p_{wd}), $\underline{k_{PE}^{+}}$), k_U^{-})).

let proxy′ = **assume** Registered(u) |
in(c_1, x);
let (=u, x_q, =p_{wd}) = dec(check(x, k_U^{+}), k_{PE}^{-}) **in**
out(c_2, zk_S(k_{PE}^{-}, x_q, p_{wd}; sign(enc((u,x_q), k_S^{+}), k_{PS}^{-}), x, u).

let store′ = **in**(c_2, z);
let (β_1,β_2,β_3) = ver_S(z) **in**
let (x_u,x_q) = dec(check(β_1, k_{PS}^{+}), k_S^{-}) **in**
assert Authenticate(x_u,x_q).

let policy = **assume** ∀ u, q. (Request(u, q) ∧ Registered(u) ⇒ Authenticate(u, q)) ...

new k_U^{-} ;   **new** k_{PS}^{-} ;
new k_{PE}^{-} : DecKey(T_1);   **new** k_S^{-} ;
new p_{wd} : Private ;
(user | proxy′ | store′ | policy)

typedef T_1 = Triple(x_u : Un, {x_q : Un | Request(x_u, x_q)}, x_p : Private)

let user = **new** q : Un; **assume** Request(u, q) |
    out(c_1, sign(enc((u,q,p_{wd}), k_{PE}^{+}), $\underline{k_U^{-}}$)).

let proxy' = **assume** Registered(u) |
    in(c_1, x);
    let (=u, x_q, =p_{wd}) = dec(check(x, k_U^{+}), k_{PE}^{-}) **in**
    out(c_2, zk_S(k_{PE}^{-}, x_q, p_{wd}; sign(enc((u,x_q), k_S^{+}), k_{PS}^{-}), x, u).

let store' = **in**(c_2, z);
    let (β_1,β_2,β_3) = ver_S(z) **in**
    let (x_u,x_q) = dec(check(β_1, k_{PS}^{+}), k_S^{-}) **in**
    assert Authenticate(x_u,x_q).

let policy = **assume** $\forall$ u, q. (Request(u, q) $\wedge$ Registered(u) $\Rightarrow$ Authenticate(u, q)) ...

new k_U^{-} : SigKey(PubEnc(T_1));
new k_{PE}^{-} : DecKey(T_1);
new p_{wd} : Private ;
(user | proxy' | store' | policy)

new k_{PS}^{-} ;
new k_S^{-} ;

```
typedef $T_1$ = Triple($x_u$ : Un, {$x_q$ : Un | Request($x_u$, $x_q$)}, $x_p$ : Private)

let user = new q : Un; assume Request(u, q) |
           out($c_1$, sign(enc((u,q,$p_{wd}$), $k_{PE}^+$), $k_U^-$)).

let proxy' = assume Registered(u) |
           in($c_1$, x);
           let (=u, $x_q$, =$p_{wd}$) = dec(check(x, $k_U^+$), $k_{PE}^-$) in
           out($c_2$, $zk_S$($k_{PE}^-$, $x_q$, $p_{wd}$; sign(enc($\underline{(u,x_q)}$, $k_S^+$), $k_{PS}^-$), x, u).

let store' = in($c_2$, z);
           let ($\beta_1$,$\beta_2$,$\beta_3$) = $ver_S$(z) in
           let ($x_u$,$x_q$) = dec(check($\beta_1$, $k_{PS}^+$), $k_S^-$) in
           assert Authenticate($x_u$,$x_q$).

let policy = assume $\forall$ u, q. (Request(u, q) $\wedge$ Registered(u) $\Rightarrow$ Authenticate(u, q)) ...


new $k_U^-$ : SigKey(PubEnc($T_1$));          new $k_{PS}^-$                              ;
new $k_{PE}^-$ : DecKey($T_1$);                 new $k_S^-$                          ;
new $p_{wd}$ : Private          ;
(user | proxy'        | store' | policy)
```

typedef T_1 = Triple(x_u : Un, {x_q : Un | Request(x_u, x_q)}, x_p : Private)
typedef T_2 = Pair(x_u : Un, {x_q : Un | Request(x_u, x_q) $\wedge$ Registered(x_u)})

let user = **new** q : Un; **assume** Request(u, q) |
        out(c_1, sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_U^{-})).

let proxy' = **assume** Registered(u) |
        in(c_1, x);
        let (=u, x_q, =p_{wd}) = dec(check(x, k_U^{+}), k_{PE}^{-}) **in**
        out(c_2, zk_S(k_{PE}^{-}, x_q, p_{wd}; sign(enc($\underline{(u,x_q)}$, k_S^{+}), k_{PS}^{-}), x, u).

let store' = **in**(c_2, z);
        let (β_1,β_2,β_3) = ver_S(z) **in**
        let (x_u,x_q) = dec(check(β_1, k_{PS}^{+}), k_S^{-}) **in**
        assert Authenticate(x_u,x_q).

let policy = **assume** $\forall$ u, q. (Request(u, q) $\wedge$ Registered(u) $\Rightarrow$ Authenticate(u, q)) ...

new k_U^{-} : SigKey(PubEnc(T_1));   **new** k_{PS}^{-} ;
new k_{PE}^{-} : DecKey(T_1);   **new** k_S^{-} ;
new p_{wd} : Private   ;
(user | proxy' | store' | policy)

typedef T_1 = Triple(x_u : Un, {x_q : Un | Request(x_u, x_q)}, x_p : Private)
typedef T_2 = Pair(x_u : Un, {x_q : Un | Request(x_u, x_q) $\wedge$ Registered(x_u)})

let user = **new** q : Un; **assume** Request(u, q) |
    out(c_1, sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_U^{-})).

let proxy′ = **assume** Registered(u) |
    in(c_1, x);
    let (=u, x_q, =p_{wd}) = dec(check(x, k_U^{+}), k_{PE}^{-}) **in**
    out(c_2, zk_S(k_{PE}^{-}, x_q, p_{wd}; sign(enc((u,x_q), k_S^{+}), k_{PS}^{-}), x, u).

let store′ = **in**(c_2, z);
    let (β_1,β_2,β_3) = ver_S(z) **in**
    let (x_u,x_q) = dec(check(β_1, k_{PS}^{+}), k_S^{-}) **in**
    assert Authenticate(x_u,x_q).

let policy = **assume** $\forall$ u, q. (Request(u, q) $\wedge$ Registered(u) $\Rightarrow$ Authenticate(u, q)) ...

new k_U^{-} : SigKey(PubEnc(T_1));
new k_{PE}^{-} : DecKey(T_1);
new p_{wd} : Private ;
new k_{PS}^{-} : SigKey(PubEnc(T_2)) ;
new k_S^{-} : DecKey(T_2) ;
(user | proxy′ | store′ | policy)

```
typedef $T_1$ = Triple($x_u$ : Un, {$x_q$ : Un | Request($x_u$, $x_q$)}, $x_p$ : Private)
typedef $T_2$ = Pair($x_u$ : Un, {$x_q$ : Un | Request($x_u$, $x_q$) $\wedge$ Registered($x_u$)})


let user = new q : Un; assume Request(u, q) |
            out($c_1$, sign(enc((u,q,$p_{wd}$), $k_{PE}^{+}$), $k_U^{-}$)).

let proxy' = assume Registered(u) |
            in($c_1$, x);
            let (=u, $x_q$, =$p_{wd}$) = dec(check(x, $k_U^{+}$), $k_{PE}^{-}$) in
            out($c_2$, $zk_S$($k_{PE}^{-}$, $x_q$, $p_{wd}$; sign(enc((u,$x_q$), $k_S^{+}$), $k_{PS}^{-}$), x, u).

let store' = in($c_2$, z);
            let ($\beta_1$,$\beta_2$,$\beta_3$) = $ver_S$(z) in
            let ($x_u$,$x_q$) = dec(check($\beta_1$, $k_{PS}^{+}$), $k_S^{-}$) in
            assert Authenticate($x_u$,$x_q$).

let policy = assume $\forall$ u, q. (Request(u, q) $\wedge$ Registered(u) $\Rightarrow$ Authenticate(u, q)) ...


new $k_U^{-}$ : SigKey(PubEnc($T_1$));          new $k_{PS}^{-}$ : SigKey(PubEnc($T_2$))        ;
new $k_{PE}^{-}$ : DecKey($T_1$);               new $k_S^{-}$ : DecKey($T_2$)        ;
new $p_{wd}$ : Private          ;
(user | proxy'        | store' | policy)
```

✓ Transformed protocol type-checks when all participants are honest

typedef T_1 = Triple(x_u : Un, {x_q : Un | Request(x_u, x_q)}, x_p : Private)
typedef T_2 = Pair(x_u : Un, {x_q : Un | Request(x_u, x_q) $\wedge$ Registered(x_u)})

let user = **new** q : Un; **assume** Request(u, q) |
    out(c_1, sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_U^{-})).

let proxy' = **assume** Registered(u) |
    in(c_1, x);
    let (=u, x_q, =p_{wd}) = dec(check(x, k_U^{+}), k_{PE}^{-}) **in**
    out(c_2, zk_S(k_{PE}^{-}, x_q, p_{wd}; sign(enc((u,x_q), k_S^{+}), k_{PS}^{-}), x, u)).

But these annotations
are <u>not</u> appropriate when
proxy is compromised

let store' = **in**(c_2, z);
    let (β_1,β_2,β_3) = ver_S(z) **in**
    let (x_u,x_q) = dec(check(β_1, k_{PS}^{+}), k_S^{-}) **in**
    assert Authenticate(x_u,x_q).

let policy = **assume** $\forall$ u, q. (Request(u, q) $\wedge$ Registered(u) $\Rightarrow$ Authenticate(u, q)) ...

new k_U^{-} : SigKey(PubEnc(T_1)); **new** k_{PS}^{-} : SigKey(PubEnc(T_2)) ;
new k_{PE}^{-} : DecKey(T_1); **new** k_S^{-} : DecKey(T_2) ;
new p_{wd} : Private ;
(user | proxy' | store' | policy)

typedef T_1 = Triple(x_u : Un, {x_q : Un | Request(x_u, x_q)}, x_p : Private)
typedef T_2 = Pair(x_u : Un, {x_q : Un | Request(x_u, x_q) $\wedge$ Registered(x_u)})

let user = **new** q : Un; **assume** Request(u, q) |
  out(c_1, sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_U^{-})).

let bad_proxy = **out**(c_{pub}, (k_{PE}^{-}, k_{PS}^{-}, p_{wd})).

let store’ = **in**(c_2, z);
  let (β_1,β_2,β_3) = ver_S(z) **in**
  let (x_u,x_q) = dec(check(β_1, k_{PS}^{+}), k_S^{-}) **in**
  assert Authenticate(x_u,x_q).

let policy = **assume** $\forall$ u, q. (Request(u, q) ~~$\wedge$ Registered(u)~~ $\Rightarrow$ Authenticate(u, q)) ...

  assume Compromised(p).

new k_U^{-} : SigKey(PubEnc(T_1)); **new** k_{PS}^{-} : SigKey(PubEnc(T_2)) ;
new k_{PE}^{-} : DecKey(T_1); **new** k_S^{-} : DecKey(T_2) ;
new p_{wd} : Private ;
(user | bad_proxy | store’ | policy)

typedef PrivateUnlessP = {Private | ¬Compromised(p)} ∨ {Un | Compromised(p)}
typedef T_1 = Triple(x_u : Un, {x_q : U , x_p : PrivateUnlessP)
typedef T_2 = Pair(x_u : Un, {x_q : U Registered(x_u)})

Union type

let user = **new** q : Un; **assume** Request(u, q) |
out(c_1, sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_U^{-})).

let bad_proxy = **out**(c_{pub}, (k_{PE}^{-}, k_{PS}^{-}, p_{wd})).

let store’ = **in**(c_2, z);
let (β_1,β_2,β_3) = ver_S(z) **in**
let (x_u,x_q) = dec(check(β_1, k_{PS}^{+}), k_S^{-}) **in**
assert Authenticate(x_u,x_q).

let policy = **assume** ∀ u, q. (Request(u, q) ~~∧ Registered(u)~~ ⇒ Authenticate(u, q)) ...
assume Compromised(p).

new k_U^{-} : SigKey(PubEnc(T_1)); **new** k_{PS}^{-} : SigKey(PubEnc(T_2)) ;
new k_{PE}^{-} : DecKey(T_1); **new** k_S^{-} : DecKey(T_2) ;
new p_{wd} : PrivateUnlessP;
(user | bad_proxy | store’ | policy)

typedef PrivateUnlessP = {Private | ¬Compromised(p)} ∨ {Un | Compromised(p)}
typedef T_1 = Triple(x_u : Un, {x_q : Un | Request(x_u, x_q)}, x_p : PrivateUnlessP)
typedef T_2 = Pair(x_u : Un, {x_q : Un | Request(x_u, x_q) ∧ Registered(x_u)})
typedef T_2unlessP = {T_2 | ¬Compromised(p)} ∨ {Un | Compromised(p)}

let user = **new** q : Un; **assume** Request(u, q) |
out(c_1, sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_U^{-})).

let bad_proxy = **out**(c_{pub}, (k_{PE}^{-}, k_{PS}^{-}, p_{wd})).

let store’ = **in**(c_2, z);
let (β_1,β_2,β_3) = ver_S(z) **in**
let (x_u,x_q) = dec(check(β_1, k_{PS}^{+}), k_S^{-}) **in**
assert Authenticate(x_u,x_q).

let policy = **assume** ∀ u, q. (Request(u, q) ~~∧ Registered(u)~~ ⇒ Authenticate(u, q)) …
assume Compromised(p).

new k_U^{-} : SigKey(PubEnc(T_1));
new k_{PE}^{-} : DecKey(T_1);
new p_{wd} : PrivateUnlessP;
(user | bad_proxy | store’ | policy)

new k_{PS}^{-} : SigKey(PubEnc(T_2unlessP));
new k_S^{-} : DecKey(T_2unlessP);

...

typedef T_1 = Triple(x_u : Un, {x_q : Un | Request(x_u, x_q)}, x_p : PrivateUnlessP)

...

let store' = **in**(c_2, z);
    let (β_1,β_2,β_3) = **ver**$_S$**(z)** **in**
    let (x_u,x_q) = dec(check(β_1, k_{PS}^{+}), k_S^{-}) **in**
    assert Authenticate(x_u,x_q).

let policy = **assume** ∀ u, q. (Request(u, q) ~~∧ Registered(u)~~ ⇒ Authenticate(u, q)) ...
    assume Compromised(p).

new k_U^{-} : SigKey(PubEnc(T_1));
new k_{PE}^{-} : DecKey(T_1);
new p_{wd} : PrivateUnlessP;
(user | bad_proxy | store' | policy)

new k_{PS}^{-} : SigKey(PubEnc(T_2unlessP));
new k_S^{-} : DecKey(T_2unlessP);

…

typedef T_1 = Triple(x_u : Un, {x_q : Un | Request(x_u, x_q)}, x_p : PrivateUnlessP)

…

stmt S = check(β_1,k_{PS}^+) = enc((β_3,α_2), k_S^+) ∧ dec(check(β_2, k_U^+), α_1) = (β_3,α_2,α_3)

let store’ = **in**(c_2, z);
    let (β_1,β_2,β_3) = **ver$_S$(z)** **in**
    let (x_u,x_q) = dec(check(β_1, k_{PS}^+), k_S^-) **in**
    assert Authenticate(x_u,x_q).

let policy = **assume** ∀ u, q. (Request(u, q) ~~∧ Registered(u)~~ ⇒ Authenticate(u, q)) …

assume Compromised(p).

new k_U^- : SigKey(PubEnc(T_1));
new k_{PE}^- : DecKey(T_1);
new p_{wd} : PrivateUnlessP;
(user | bad_proxy | store’ | policy)

new k_{PS}^- : SigKey(PubEnc(T_2unlessP));
new k_S^- : DecKey(T_2unlessP);

…
typedef T_1 = Triple(x_u : Un, {x_q : Un | Request(x_u, x_q)}, x_p : PrivateUnlessP)
…

stmt S = $\exists \alpha_1,\alpha_2,\alpha_3$.check($\beta_1$,$k_{PS}^+$) = enc(($\beta_3$,$\alpha_2$), k_S^+) $\wedge$ dec(check(β_2, k_U^+), α_1) = (β_3,α_2,α_3)

let store’ = **in**(c_2, z);
    let (β_1,β_2,β_3) = $\mathbf{vers}_S$**(z) in**
    let (x_u,x_q) = dec(check(β_1, k_{PS}^+), k_S^-) **in**
    assert Authenticate(x_u,x_q).

let policy = **assume** $\forall$ u, q. (Request(u, q) ~~$\wedge$ Registered(u)~~ $\Rightarrow$ Authenticate(u, q)) …
    assume Compromised(p).

new k_U^- : SigKey(PubEnc(T_1));
new k_{PE}^- : DecKey(T_1);
new p_{wd} : PrivateUnlessP;
new k_{PS}^- : SigKey(PubEnc(T_2unlessP));
new k_S^- : DecKey(T_2unlessP);
(user | bad_proxy | store’ | policy)

…

typedef T_1 = Triple(x_u : Un, {x_q : Un | Request(x_u, x_q)}, x_p : PrivateUnlessP)

…

stmt $S = \exists\alpha_1,\alpha_2,\alpha_3.\mathrm{check}(\beta_1,k_{PS}^+) = \mathrm{enc}((\beta_3,\alpha_2), k_S^+) \wedge \mathrm{dec}(\mathrm{check}(\beta_2, \underbrace{k_U^+}_{:\mathrm{VerKey}(\mathrm{PubEnc}(T_1))}), \alpha_1) = (\beta_3,\alpha_2,\alpha_3)$

let store’ = **in**(c_2, z);
  let (β_1,β_2,β_3) = **ver$_S$(z) in**
  let (x_u,x_q) = dec(check(β_1, k_{PS}^+), k_S^-) **in**
  assert Authenticate(x_u,x_q).

let policy = **assume** $\forall$ u, q. (Request(u, q) ~~$\wedge$ Registered(u)~~ $\Rightarrow$ Authenticate(u, q)) …

assume Compromised(p).

new k_U^- : SigKey(PubEnc(T_1));
new k_{PE}^- : DecKey(T_1);
new p_{wd} : PrivateUnlessP;
(user | bad_proxy | store’ | policy)

new k_{PS}^- : SigKey(PubEnc(T_2unlessP));
new k_S^- : DecKey(T_2unlessP);

…

typedef T_1 = Triple(x_u : Un, {x_q : Un | Request(x_u, x_q)}, x_p : PrivateUnlessP)

…

stmt $S = \exists \alpha_1,\alpha_2,\alpha_3.\mathrm{check}(\beta_1,k_{PS}^{+}) = \mathrm{enc}((\beta_3,\alpha_2), k_S^{+}) \wedge \mathrm{dec}(\underbrace{\mathrm{check}(\beta_2, k_U^{+})}_{:\mathrm{PubEnc}(T_1)}, \alpha_1) = (\beta_3,\alpha_2,\alpha_3)$

let store’ = **in**(c_2, z);
let (β_1,β_2,β_3) = **vers(z)** **in**
let (x_u,x_q) = dec(check(β_1, k_{PS}^{+}), k_S^{-}) **in**
assert Authenticate(x_u,x_q).

let policy = **assume** ∀ u, q. (Request(u, q) ~~∧ Registered(u)~~ ⇒ Authenticate(u, q)) …
assume Compromised(p).

new k_U^{-} : SigKey(PubEnc(T_1));
new k_{PE}^{-} : DecKey(T_1);
new p_{wd} : PrivateUnlessP;
(user | bad_proxy | store’ | policy)

new k_{PS}^{-} : SigKey(PubEnc(T_2unlessP));
new k_S^{-} : DecKey(T_2unlessP);

```
...
typedef $T_1$ = Triple($x_u$ : Un, {$x_q$ : Un | Request($x_u$, $x_q$)}, $x_p$ : PrivateUnlessP)
...
```

stmt $S = \exists\alpha_1,\alpha_2,\alpha_3.\text{check}(\beta_1,k_{PS}^+) = \text{enc}((\beta_3,\alpha_2), k_S^+) \wedge \text{dec}(\text{check}(\beta_2, k_U^+), \alpha_1) = \underbrace{(\beta_3,\alpha_2,\alpha_3)}_{:T_1}$

```
let store' = in($c_2$, z);
             let ($\beta_1$,$\beta_2$,$\beta_3$) = vers$_S$(z) in
             let ($x_u$,$x_q$) = dec(check($\beta_1$, $k_{PS}^+$), $k_S^-$) in
             assert Authenticate($x_u$,$x_q$).

let policy = assume $\forall$ u, q. (Request(u, q) ~~$\wedge$ Registered(u)~~ $\Rightarrow$ Authenticate(u, q)) ...

                assume Compromised(p).

new $k_U^-$ : SigKey(PubEnc($T_1$));            new $k_{PS}^-$ : SigKey(PubEnc($T_2$unlessP));
new $k_{PE}^-$ : DecKey($T_1$);                 new $k_S^-$ : DecKey($T_2$unlessP);
new $p_{wd}$ : PrivateUnlessP;
(user | bad_proxy | store' | policy)
```

...

typedef T_1 = Triple(x_u : Un, {x_q : Un | Request(x_u, x_q)}, x_p : PrivateUnlessP)

...

stmt $S = \exists\alpha_1,\alpha_2,\alpha_3.\mathrm{check}(\beta_1,k_{PS}^{+}) = \mathrm{enc}((\beta_3,\alpha_2), k_S^{+}) \wedge \mathrm{dec}(\mathrm{check}(\beta_2, k_U^{+}), \alpha_1) = \underbrace{(\beta_3,\alpha_2,\alpha_3)}_{:T_1}$

```
let store' = in($c_2$, z);
             let ($\beta_1,\beta_2,\beta_3$) = $\mathrm{ver}_S$(z) in
             let ($x_u$,$x_q$) = dec(check($\beta_1$, $k_{PS}^{+}$), $k_S^{-}$) in
             assert Authenticate($x_u$,$x_q$).
```

let policy = **assume** ∀ u, q. (Request(u, q) ~~∧ Registered(u)~~ ⇒ Authenticate(u, q)) ...

assume Compromised(p).

new k_U^{-} : SigKey(PubEnc(T_1));

new k_{PE}^{-} : DecKey(T_1);

new p_{wd} : PrivateUnlessP;

new k_{PS}^{-} : SigKey(PubEnc(T_2unlessP));

new k_S^{-} : DecKey(T_2unlessP);

(user | bad_proxy | store' | policy)

…

typedef T_1 = Triple(x_u : Un, {x_q : Un | Request(x_u, x_q)}, x_p : PrivateUnlessP)

…

stmt $S = \exists \alpha_1,\alpha_2,\alpha_3.\mathrm{check}(\beta_1,k_{PS}^{+}) = \mathrm{enc}((\beta_3,\alpha_2), k_S^{+}) \wedge \mathrm{dec}(\mathrm{check}(\beta_2, k_U^{+}), \alpha_1) = (\beta_3,\alpha_2,\alpha_3)$
$\wedge$ Request(β_3, α_2)

let store’ = **in**(c_2, z);
  let (β_1,β_2,β_3) = **vers(z)** **in**
  let (x_u,x_q) = dec(check(β_1, k_{PS}^{+}), k_S^{-}) **in**
  assert Authenticate(x_u,x_q).

let policy = **assume** $\forall$ u, q. (Request(u, q) ~~$\wedge$ Registered(u)~~ $\Rightarrow$ Authenticate(u, q)) …
  assume Compromised(p).

new k_U^{-} : SigKey(PubEnc(T_1));
new k_{PE}^{-} : DecKey(T_1);
new p_{wd} : PrivateUnlessP;
(user | bad_proxy | store’ | policy)

new k_{PS}^{-} : SigKey(PubEnc(T_2unlessP));
new k_S^{-} : DecKey(T_2unlessP);

…

typedef T_1 = Triple(x_u : Un, {x_q : Un | Request(x_u, x_q)}, x_p : PrivateUnlessP)

…

stmt $S = \exists \alpha_1,\alpha_2,\alpha_3.\underline{\mathrm{check}(\beta_1,k_{PS}^{+})} = \mathrm{enc}((\beta_3,\alpha_2), k_S^{+}) \wedge \mathrm{dec}(\mathrm{check}(\beta_2, k_U^{+}), \alpha_1) = (\beta_3,\alpha_2,\alpha_3)$
$\wedge\ \mathrm{Request}(\beta_3, \alpha_2)$

let store′ = **in**(c_2, z);
    let (β_1,β_2,β_3) = **ver_{s}(z) in**
    let (x_u,x_q) = dec($\underline{\mathrm{check}(\beta_1, k_{PS}^{+})}$, k_S^{-}) **in**
    assert Authenticate(x_u,x_q).

let policy = **assume** ∀ u, q. (Request(u, q) ~~∧ Registered(u)~~ ⇒ Authenticate(u, q)) …
    assume Compromised(p).

new k_U^{-} : SigKey(PubEnc(T_1));
new k_{PE}^{-} : DecKey(T_1);
new p_{wd} : PrivateUnlessP;
(user | bad_proxy | store′ | policy)

new k_{PS}^{-} : SigKey(PubEnc(T_2unlessP));
new k_S^{-} : DecKey(T_2unlessP);

…

typedef T_1 = Triple(x_u : Un, {x_q : Un | Request(x_u, x_q)}, x_p : PrivateUnlessP)

…

stmt $S = \exists \alpha_1,\alpha_2,\alpha_3.\underline{\mathrm{check}(\beta_1,k_{PS}^+)} = \mathrm{enc}((\beta_3,\alpha_2), \underline{k_S^+}) \wedge \mathrm{dec}(\mathrm{check}(\beta_2, k_U^+), \alpha_1) = (\beta_3,\alpha_2,\alpha_3)$

$\wedge\ \mathrm{Request}(\beta_3, \alpha_2)$

let store’ = **in**(c_2, z);

  let (β_1,β_2,β_3) = **ver$_S$(z) in**

  let (x_u,x_q) = dec($\underline{\mathrm{check}(\beta_1, k_{PS}^+)}$), $\underline{k_S^-}$) **in**

  assert Authenticate(x_u,x_q).

let policy = **assume** ∀ u, q. (Request(u, q) ~~∧ Registered(u)~~ ⇒ Authenticate(u, q)) …

  assume Compromised(p).

new k_U^- : SigKey(PubEnc(T_1));

new k_{PE}^- : DecKey(T_1);

new p_{wd} : PrivateUnlessP;

(user | bad_proxy | store’ | policy)

new k_{PS}^- : SigKey(PubEnc(T_2unlessP));

new k_S^- : DecKey(T_2unlessP);

…

typedef T_1 = Triple(x_u : Un, {x_q : Un | Request(x_u, x_q)}, x_p : PrivateUnlessP)

…

stmt S = $\exists\alpha_1,\alpha_2,\alpha_3.\mathrm{check}(\beta_1,k_{PS}^{+}) = \mathrm{enc}((\underline{\beta_3,\alpha_2}), k_S^{+}) \wedge \mathrm{dec}(\mathrm{check}(\beta_2, k_U^{+}), \alpha_1) = (\beta_3,\alpha_2,\alpha_3)$
$\wedge\ \mathrm{Request}(\beta_3, \alpha_2)$

let store’ = **in**(c_2, z);
    let (β_1,β_2,β_3) = **ver$_S$(z)** **in**
    let $\underline{(x_u,x_q)}$ = dec(check(β_1, k_{PS}^{+}), k_S^{-}) **in**
    assert Authenticate(x_u,x_q).

let policy = **assume** ∀ u, q. (Request(u, q) ~~∧ Registered(u)~~ ⇒ Authenticate(u, q)) …
    assume Compromised(p).

new k_U^{-} : SigKey(PubEnc(T_1));
new k_{PE}^{-} : DecKey(T_1);
new p_{wd} : PrivateUnlessP;
(user | bad_proxy | store’ | policy)

new k_{PS}^{-} : SigKey(PubEnc(T_2unlessP));
new k_S^{-} : DecKey(T_2unlessP);

…

typedef T_1 = Triple(x_u : Un, {x_q : Un | Request(x_u, x_q)}, x_p : PrivateUnlessP)

…

$$\exists \alpha_1,\alpha_2,\alpha_3.\mathrm{check}(\beta_1,k_{PS}^{+}) = \mathrm{enc}((\beta_3,\alpha_2), k_S^{+}) \wedge \mathrm{dec}(\mathrm{check}(\beta_2, k_U^{+}), \alpha_1) = (\beta_3,\alpha_2,\alpha_3)$$

$$\wedge\ \mathrm{Request}(\beta_3, \alpha_2)$$

$$\wedge\ \beta_3 = x_u \wedge \alpha_2 = x_q$$

let store’ = **in**(c_2, z);
    let (β_1,β_2,β_3) = **ver_S(z) in**
    let (x_u,x_q) = dec(check(β_1, k_{PS}^{+}), k_S^{-}) **in**
    assert Authenticate(x_u,x_q).

let policy = **assume** $\forall$ u, q. (Request(u, q) ~~$\wedge$ Registered(u)~~ $\Rightarrow$ Authenticate(u, q)) …

assume Compromised(p).

new k_U^{-} : SigKey(PubEnc(T_1));
new k_{PE}^{-} : DecKey(T_1);
new p_{wd} : PrivateUnlessP;
(user | bad_proxy | store’ | policy)

new k_{PS}^{-} : SigKey(PubEnc(T_2unlessP));
new k_S^{-} : DecKey(T_2unlessP);

…

typedef T_1 = Triple(x_u : Un, {x_q : Un | Request(x_u, x_q)}, x_p : PrivateUnlessP)

…

$$\exists \alpha_1,\alpha_2,\alpha_3.\mathrm{check}(\beta_1,k_{PS}^+) = \mathrm{enc}((\beta_3,\alpha_2), k_S^+) \wedge \mathrm{dec}(\mathrm{check}(\beta_2, k_U^+), \alpha_1) = (\beta_3,\alpha_2,\alpha_3) \wedge \mathrm{Request}(x_u, x_q)$$

let store′ = **in**(c_2, z);
    let (β_1,β_2,β_3) = **ver_S(z)** **in**
    let (x_u,x_q) = dec(check(β_1, k_{PS}^+), k_S^-) **in**
    assert Authenticate(x_u,x_q).

let policy = **assume** ∀ u, q. (Request(u, q) ~~∧ Registered(u)~~ ⇒ Authenticate(u, q)) …

assume Compromised(p).

new k_U^- : SigKey(PubEnc(T_1));
new k_{PE}^- : DecKey(T_1);
new p_{wd} : PrivateUnlessP;

new k_{PS}^- : SigKey(PubEnc(T_2unlessP));
new k_S^- : DecKey(T_2unlessP);

(user | bad_proxy | store′ | policy)

...

typedef T_1 = Triple(x_u : Un, {x_q : Un | Request(x_u, x_q)}, x_p : PrivateUnlessP)

...

$$\exists \alpha_1,\alpha_2,\alpha_3.\mathrm{check}(\beta_1,k_{PS}^{+}) = \mathrm{enc}((\beta_3,\alpha_2), k_S^{+}) \wedge \mathrm{dec}(\mathrm{check}(\beta_2, k_U^{+}), \alpha_1) = (\beta_3,\alpha_2,\alpha_3)$$
$$\wedge\ \mathrm{Request}(x_u, x_q)$$

✓ Transformed protocol type-checks even when proxy is compromised ⇒ secure despite compromise

let store' = **in**(c_2, z);
    let (β_1,β_2,β_3) = **ver$_S$(z) in**
    let (x_u,x_q) = dec(check(β_1, k_{PS}^{+}), k_S^{-}) **in**
    assert Authenticate(x_u,x_q).

let policy = **assume** ∀ u, q. (Request(u, q) ~~∧ Registered(u)~~ ⇒ Authenticate(u, q)) ...

assume Compromised(p).

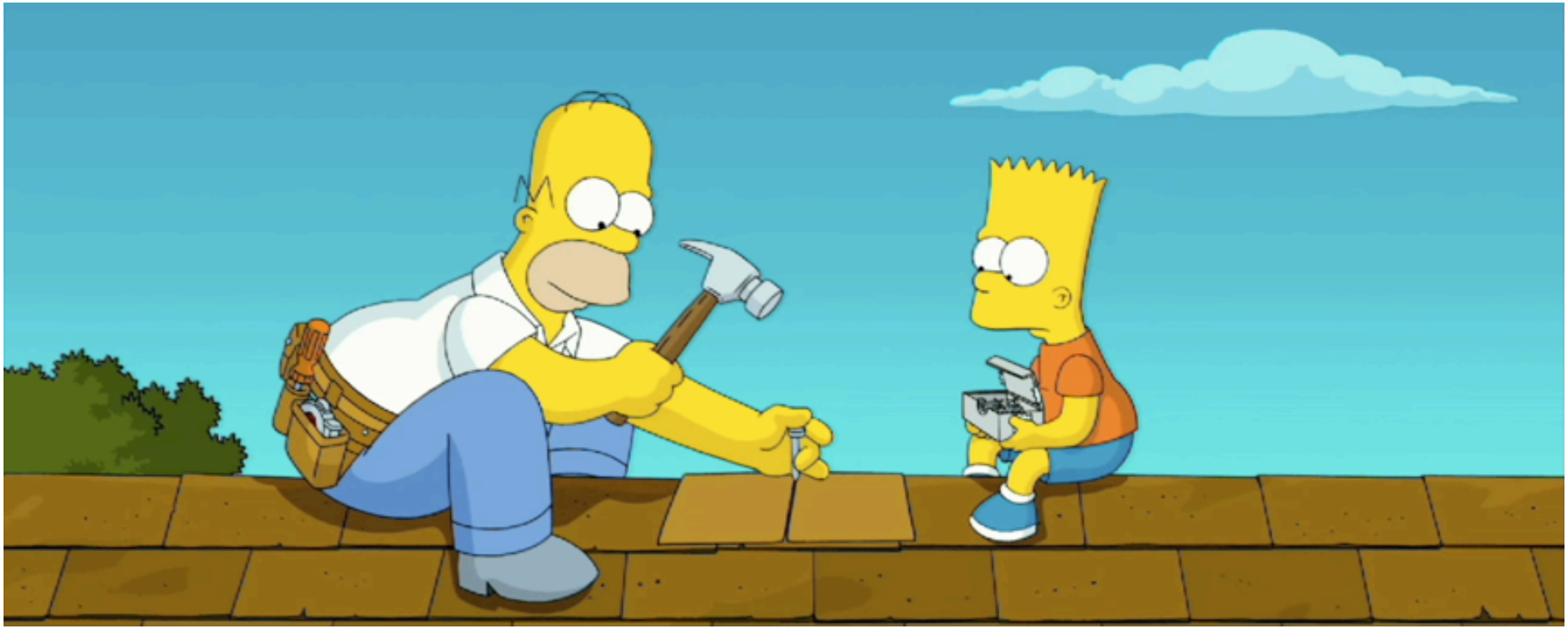
new k_U^{-} : SigKey(PubEnc(T_1));
new k_{PE}^{-} : DecKey(T_1);
new p_{wd} : PrivateUnlessP;

new k_{PS}^{-} : SigKey(PubEnc(T_2unlessP));
new k_S^{-} : DecKey(T_2unlessP);

(user | bad_proxy | store' | policy)

Implementation

- Transformation and type-checker written in O'Caml (~2000+6000 LOC)
- Both available under the Apache License: http://www.infsec.cs.uni-sb.de/projects/zk-compromise/ http://www.infsec.cs.uni-sb.de/projects/zk-typechecker/

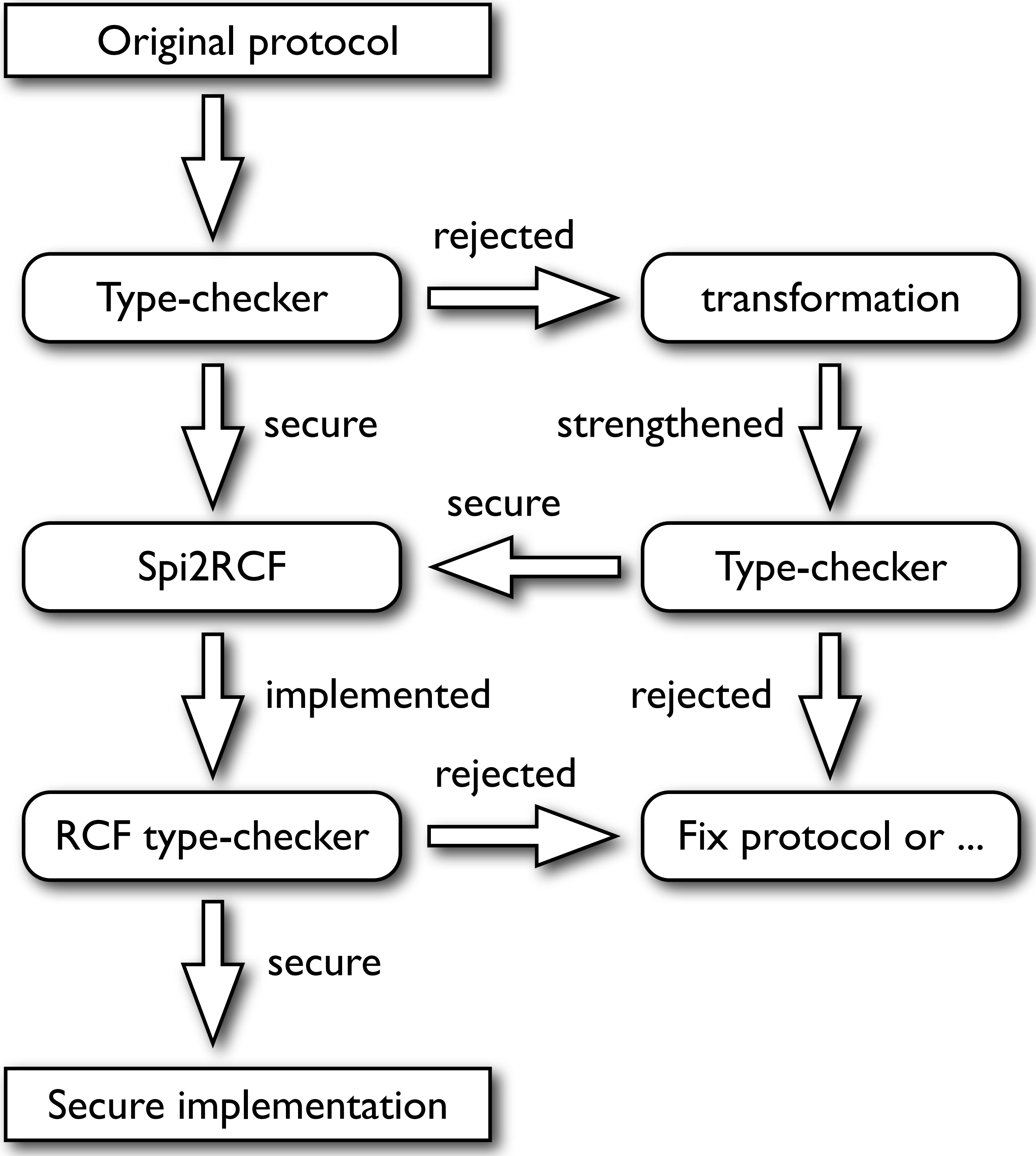


Automatic code generation

- Spi2RCF
  - **Input:**
    protocols in Spi calculus (using zero knowledge)
    + type annotations
  - **Output:**
    implementation in core fragment of ML (RCF: Refined Concurrent PCF) [Bengtson et. al., CSF '08]

    + symbolic implementation of zero-knowledge (using seals)

    + type annotations for RCF type-checker
- RCF type-checker validates the generated implementation



Original protocol

Type-checker

rejected

transformation

secure

strengthened

secure

Spi2RCF

Type-checker

implemented

rejected

rejected

RCF type-checker

Fix protocol or ...

secure

Secure implementation

Future Work

- Apply transformation to more protocols
- Optimize transformation
  - leverage authorization policy and types
  - could also use ideas from [Corin et. al, CSF ’07 & CSF ’09]
  - translation validation approach will help
    (no need to redo any proofs)
- Automatically generate concrete implementations of protocols using zero-knowledge
  - implementing ZK proof system is hard
  - efficiency is a big challenge

Thank you

Related Work

- Strengthening crypto protocols using transformations

  [Goldreich, Micali & Wigderson, STOC '87]

  - Add ZK to multi-party protocol secure against honest-but-curious participants to protect against compromise
  - Computational cryptography, broadcast communication

  [Bellare, Canetti & Krawczyk, STOC '98]

  - Transformation removes authentication assumption

  [Katz & Yung, CRYPTO '03] [Cortier et al. ESORICS '07]

  - From passive (eavesdropping) to active attackers

  [Datta, Derek, Mitchell & Pavlovic, JCS '05]

  - Methodology for modular protocol design using generic protocol transformations

Related Work (continued)

- Generating protocols from high-level specifications

  [Corin, Dénielou, Fournet, Bhargavan & Leifer, CSF '07 & CSF '09]

  - Multi-party session specifications transformed to F# implementations that are secure despite compromise
  - Very efficient generated implementation
  - More recent transformation uses F7 type-checker for translation validation (original one was proven correct)
  - Main difference
    - Session specifications have no crypto
    - Our approach applies both to existing crypto protocols and to the ones generated from high-level specs (theirs not)

let user = **new** q; **assume** Request(u, q) |
out(c_1, sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})).

let proxy = **assume** Registered(u) | …

let store = …

let policy = **assume** $\forall$ u, q. (Request(u, q) $\wedge$ Registered(u) $\Rightarrow$ Authenticate(u, q))

let user = **new** q; **assume** Request(u, q) |
out(c_1, sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_U^{-})).

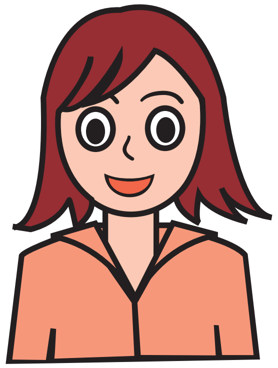
let proxy = **assume** Registered(u) | ...

let store = ...

let policy = **assume** ∀ u, q. (Request(u, q) ∧ Registered(u) ⇒ Authenticate(u, q)) |
assume Compromised(u) ⇒ ∀ q. Request(u, q) |
assume Compromised(p) ⇒ ∀ u. Registered(u)

security
despite compromise

Compromising the user



user

let user = **new** q; **assume** Request(u, q) |
out(c_1, sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_U^{-})).

let proxy = **assume** Registered(u) | ...

let store = ...

let policy = **assume** ∀ u, q. (Request(u, q) ∧ Registered(u) ⇒ Authenticate(u, q)) |
assume Compromised(u) ⇒ ∀ q. Request(u, q) |
assume Compromised(p) ⇒ ∀ u. Registered(u)

Compromising the user

user

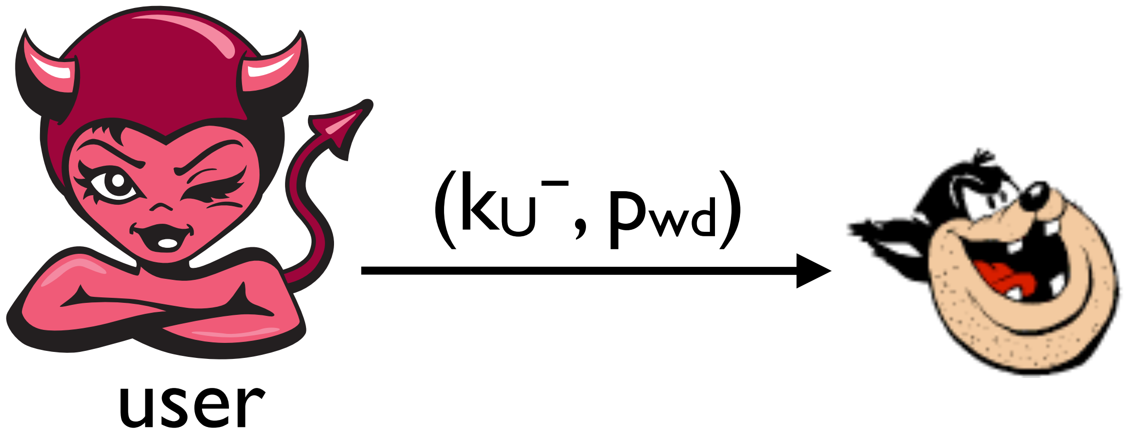
let user = **new** q; **assume** Request(u, q) |
          out(c_1, sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_U^{-})).

let proxy = **assume** Registered(u) | ...

let store = ...

let policy = **assume** ∀ u, q. (Request(u, q) ∧ Registered(u) ⇒ Authenticate(u, q)) |
          assume Compromised(u) ⇒ ∀ q. Request(u, q) |
          assume Compromised(p) ⇒ ∀ u. Registered(u)
          assume Compromised(u) ∧ ¬Compromised(p) ∧ ¬Compromised(s)

Compromising the user



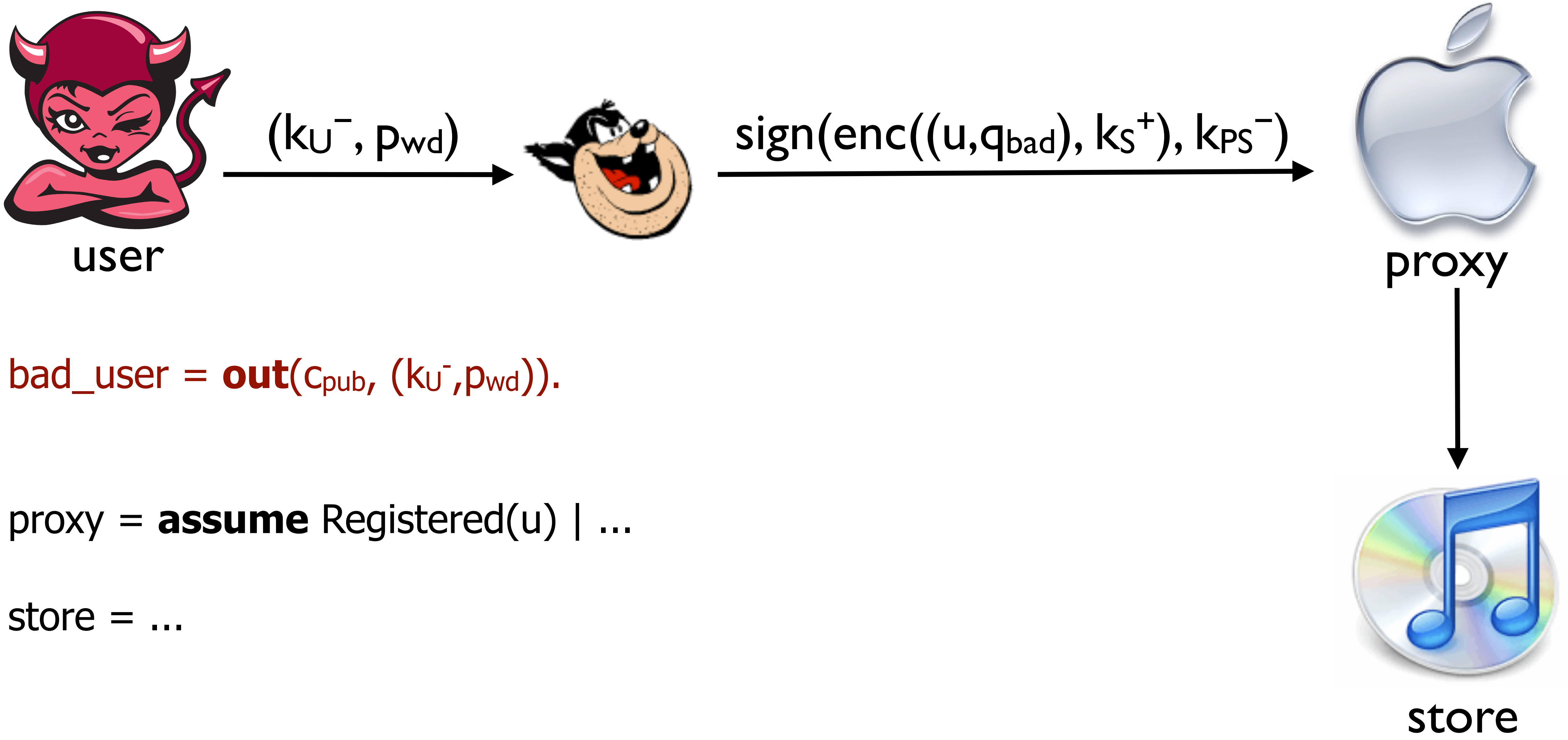
let bad_user = **out**(c_{pub}, (k_U^-,p_{wd})).

let proxy = **assume** Registered(u) | …

let store = …

let policy = **assume** ∀ u, q. (Request(u, q) ∧ Registered(u) ⇒ Authenticate(u, q)) |
assume Compromised(u) ⇒ ∀ q. Request(u, q) |
assume Compromised(p) ⇒ ∀ u. Registered(u)
assume Compromised(u) ∧ ¬Compromised(p) ∧ ¬Compromised(s)

Compromising the user



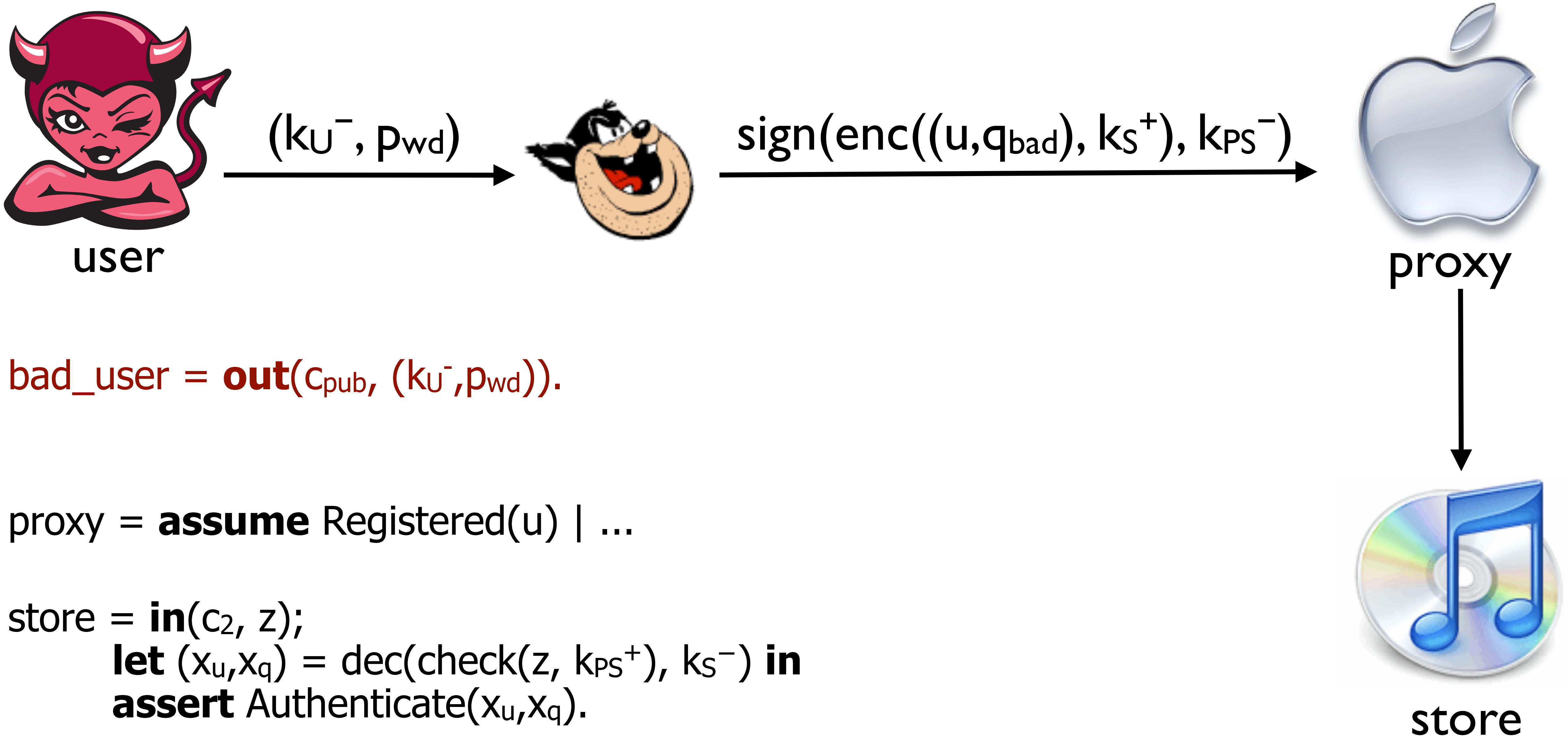
let bad_user = **out**(c_{pub}, (k_U^-,p_{wd})).

let proxy = **assume** Registered(u) | …

let store = …

let policy = **assume** ∀ u, q. (Request(u, q) ∧ Registered(u) ⇒ Authenticate(u, q)) |
assume Compromised(u) ⇒ ∀ q. Request(u, q) |
assume Compromised(p) ⇒ ∀ u. Registered(u)
assume Compromised(u) ∧ ¬Compromised(p) ∧ ¬Compromised(s)

Compromising the user



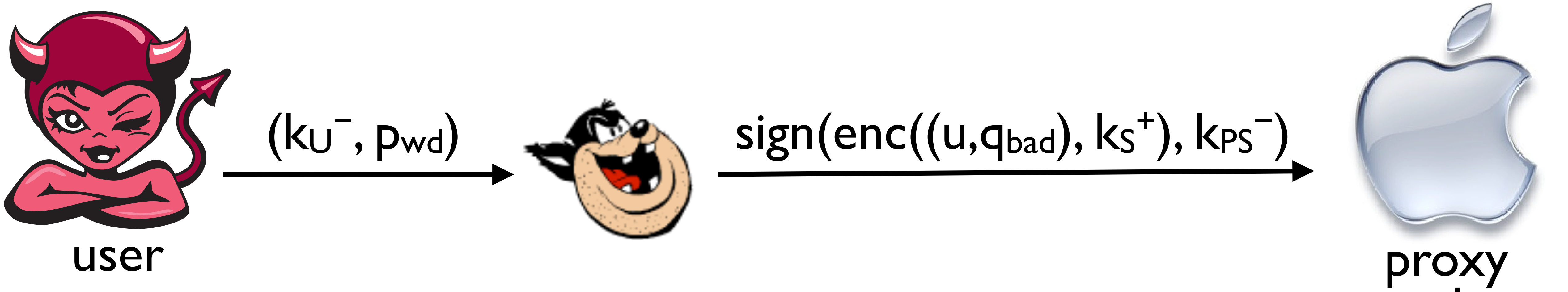
let bad_user = **out**(c_{pub}, (k_U^-,p_{wd})).

let proxy = **assume** Registered(u) | …

let store = **in**(c_2, z);
    let (x_u,x_q) = dec(check(z, k_{PS}^+), k_S^-) **in**
    assert Authenticate(x_u,x_q).

let policy = **assume** $\forall$ u, q. (Request(u, q) $\wedge$ Registered(u) $\Rightarrow$ Authenticate(u, q)) |
    assume Compromised(u) $\Rightarrow$ $\forall$ q. Request(u, q) |
    assume Compromised(p) $\Rightarrow$ $\forall$ u. Registered(u)
    assume Compromised(u) $\wedge$ $\neg$Compromised(p) $\wedge$ $\neg$Compromised(s)

Compromising the user



let bad_user = **out**(c_{pub}, (k_U^-,p_{wd})).

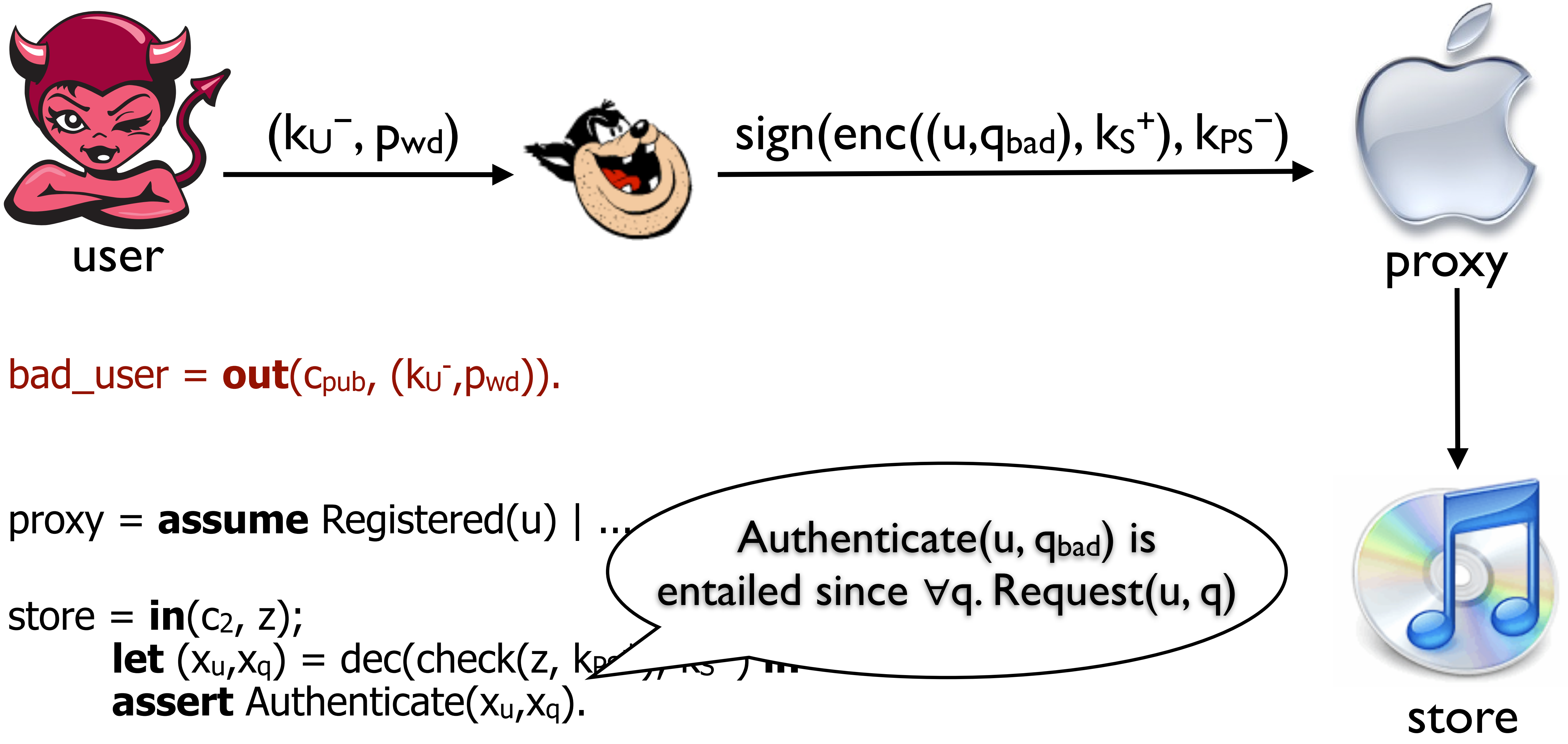
let proxy = **assume** Registered(u) | ...

let store = **in**(c_2, z);
let (x_u,x_q) = dec(check(z, k_{PS}^+), k_S^-) **in**
assert Authenticate(x_u,x_q).

Authenticate(u, q_{bad}) is entailed since ∀q. Request(u, q)

let policy = **assume** ∀ u, q. (Request(u, q) ∧ Registered(u) ⇒ Authenticate(u, q)) |
assume Compromised(u) ⇒ ∀ q. Request(u, q) |
assume Compromised(p) ⇒ ∀ u. Registered(u)
assume Compromised(u) ∧ ¬Compromised(p) ∧ ¬Compromised(s)

Compromising the user



let bad_user = **out**(c_{pub}, (k_U^-,p_{wd})).

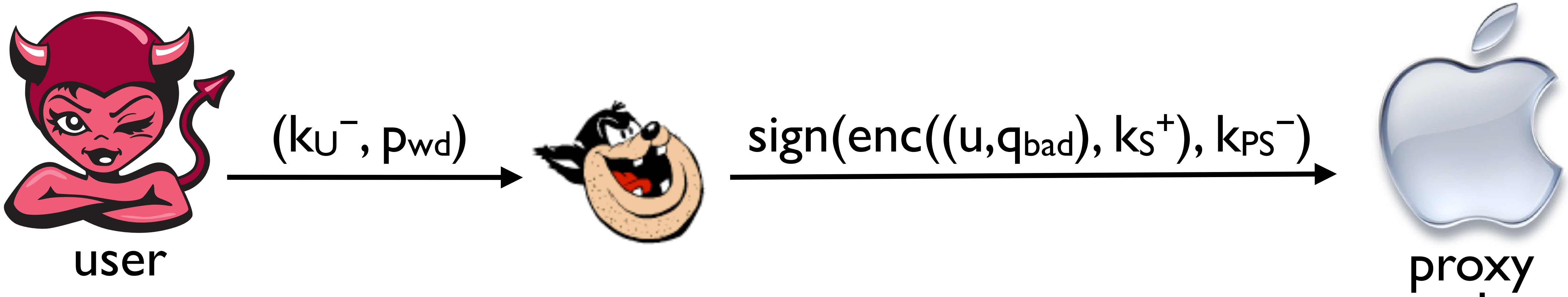
let proxy = **assume** Registered(u) | ...

let store = **in**(c_2, z);
    let (x_u,x_q) = dec(check(z, k_{PS}^+), k_S^-) **in**
    assert Authenticate(x_u,x_q).

Authenticate(u, q_{bad}) is entailed since ∀q. Request(u, q)

let policy = **assume** ∀ u, q. (~~Request(u, q) ∧~~ Registered(u) ⇒ Authenticate(u, q)) |
    assume Compromised(u) ⇒ ∀ q. Request(u, q) |
    assume Compromised(p) ⇒ ∀ u. Registered(u)
    assume Compromised(u) ∧ ¬Compromised(p) ∧ ¬Compromised(s)

Compromising the user



let bad_user = **out**(c_{pub}, (k_U^-,p_{wd})).

let proxy = **assume** Registered(u) | …

let store = **in**(c_2, z);
  let (x_u,x_q) = dec(check(z, k_{PS}^+))
  assert Authenticate(x_u,x_q).

assert succeeds, so protocol is secure despite the user's compromise

let policy = **assume** ∀ u, q. (~~Request(u, q) ∧~~ Registered(u) ⇒ Authenticate(u, q)) |
  assume Compromised(u) ⇒ ∀ q. Request(u, q) |
  assume Compromised(p) ⇒ ∀ u. Registered(u)
  assume Compromised(u) ∧ ¬Compromised(p) ∧ ¬Compromised(s)

```
typedef PrivateUnlessP = {Private | ¬Compromised(p)} ∨ {Un | Compromised(p)}
typedef $T_1$ = Triple($x_u$ : Un, {$x_q$ : Un | Request($x_u$, $x_q$)}, $x_p$:PrivateUnlessP)
typedef $T_2$ = Pair($x_u$ : Un, {$x_q$ : Un | Request($x_u$, $x_q$) ∧ Registered($x_u$)})
typedef $T_2$unlessP = {$T_2$ | ¬Compromised(p)} ∨ {Un | Compromised(p)}
new $k_U^-$ : SigKey(PubEnc($T_1$));
new $k_{PE}^-$ : DecKey($T_1$);
new $k_{PS}^-$ : SigKey(PubEnc($T_2$unlessP));
new $k_S^-$ : DecKey($T_2$unlessP);
new $p_{wd}$ : Private; (user | proxy | store | policy)


let user = new q : Un; assume Request(u, q) |
           out($c_1$, sign(enc((u,q,$p_{wd}$), $k_{PE}^+$), $k_U^-$)).

let proxy = assume Registered(u) |
           in($c_1$, x);
           let (=u, $x_q$, =$p_{wd}$) = dec(check(x, $k_U^+$), $k_{PE}^-$) in
           out($c_2$, sign(enc((u,$x_q$), $k_S^+$), $k_{PS}^-$)).

let store = in($c_2$, z);
           let ($x_u$,$x_q$) = dec(check(z, $k_{PS}^+$), $k_S^-$) in
           assert Authenticate($x_u$,$x_q$).

let policy = assume ∀ u, q. (Request(u, q) ∧ Registered(u) ⇒ Authenticate(u, q)) |
              assume Compromised(p) ⇒ ∀ u. Registered(u) |
              assume Compromised(u) ⇒ ∀ q. Request(u, q).
```

```
typedef PrivateUnlessP = {Private | ¬Compromised(p)} ∨ {Un | Compromised(p)}
typedef $T_1$ = Triple($x_u$ : Un, {$x_q$ : Un | Request($x_u$, $x_q$)}, $x_p$:PrivateUnlessP)
typedef $T_2$ = Pair($x_u$ : Un, {$x_q$ : Un | Request($x_u$, $x_q$) ∧ Registered($x_u$)})
typedef $T_2$unlessP = {$T_2$ | ¬Compromised(p)} ∨ {Un | Compromised(p)}
new $k_U^-$ : SigKey(PubEnc($T_1$));
new $k_{PE}^-$ : DecKey($T_1$);
new $k_{PS}^-$ : SigKey(PubEnc($T_2$unlessP));
new $k_S^-$ : DecKey($T_2$unlessP);
new $p_{wd}$ : Private; (user | proxy | store | policy)

stmt $S$ = check($\beta_1$,$k_{PS}^+$) = enc(($\beta_3$,$\alpha_2$), $k_S^+$) ∧ dec(check($\beta_2$, $k_U^+$), $\alpha_1$) = ($\beta_3$,$\alpha_2$,$\alpha_3$)

let user = new q : Un; assume Request(u, q) |
           out($c_1$, sign(enc((u,q,$p_{wd}$), $k_{PE}^+$), $k_U^-$)).

let proxy = assume Registered(u) |
           in($c_1$, x);
           let (=u, $x_q$, =$p_{wd}$) = dec(check(x, $k_U^+$), $k_{PE}^-$) in
           out($c_2$, sign(enc((u,$x_q$), $k_S^+$), $k_{PS}^-$)).

let store = in($c_2$, z);
           let ($x_u$,$x_q$) = dec(check(z, $k_{PS}^+$), $k_S^-$) in
           assert Authenticate($x_u$,$x_q$).

let policy = assume ∀ u, q. (Request(u, q) ∧ Registered(u) ⇒ Authenticate(u, q)) |
              assume Compromised(p) ⇒ ∀ u. Registered(u) |
              assume Compromised(u) ⇒ ∀ q. Request(u, q).
```

Thank you