IS&C

UNIVERSITÄT DES SAARLANDES

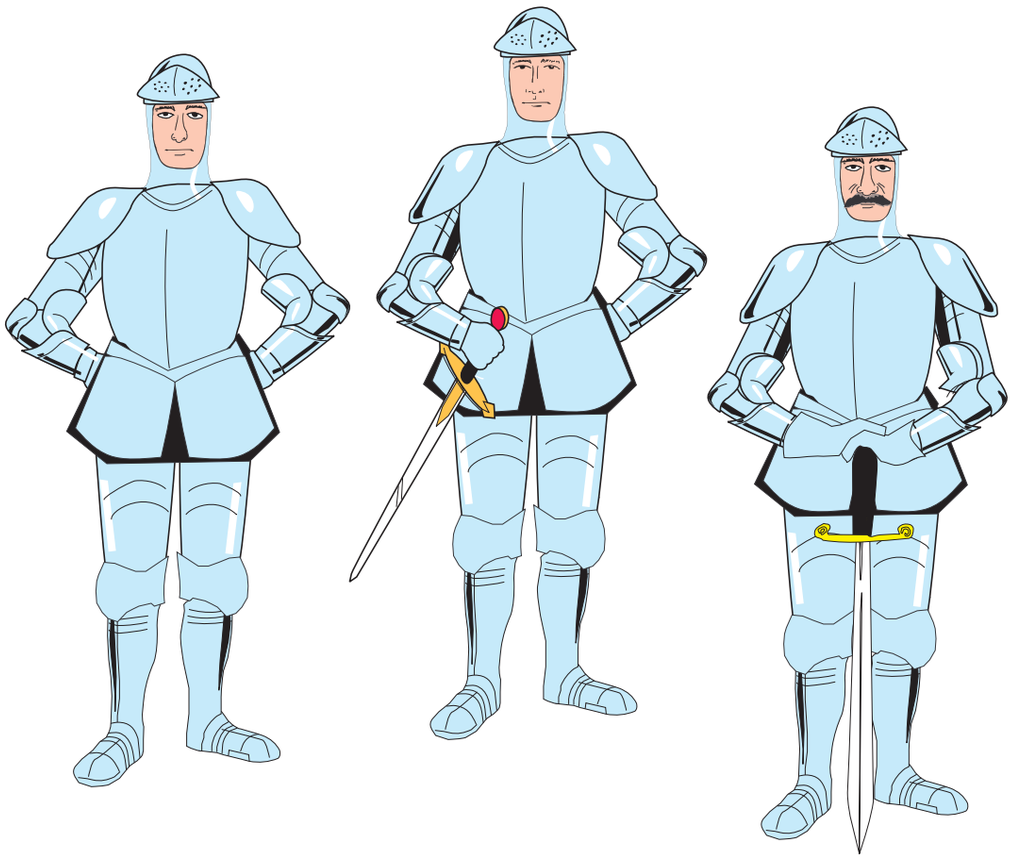
Achieving Security Despite Compromise Using Zero-knowledge

Cătălin Hriţcu

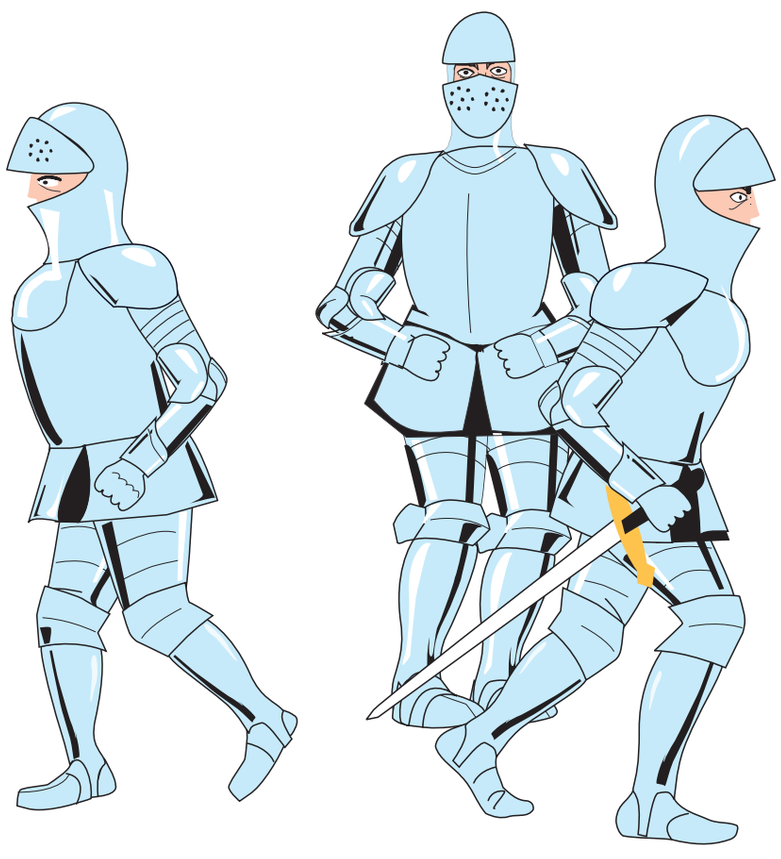
Saarland University, Saarbrücken, Germany

Joint work with: Michael Backes, Martin Grochulla and Matteo Maffei

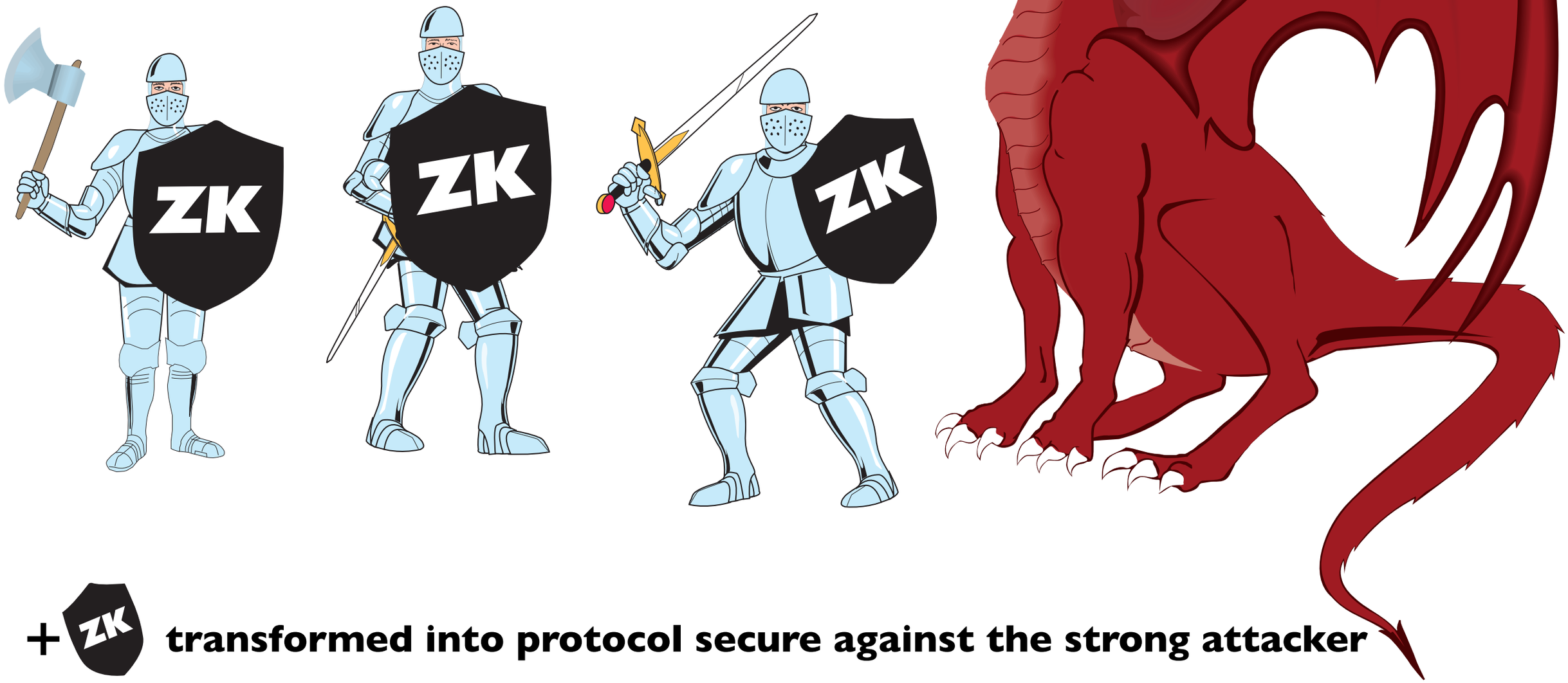
Computer Security Foundations Symposium, Port Jefferson, New York, 2009



protocol secure against a weak attacker



but insecure against strong attacker



+ ZK transformed into protocol secure against the strong attacker

Contributions

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

Contributions

- **Goal:** to aid secure protocol design
  - designer only needs to consider restricted security threats
- Automatic protocol transformation adding ZK proofs

  - Enforce authenticity even if participants compromised
  - Preserve secrecy if everybody is honest

Contributions

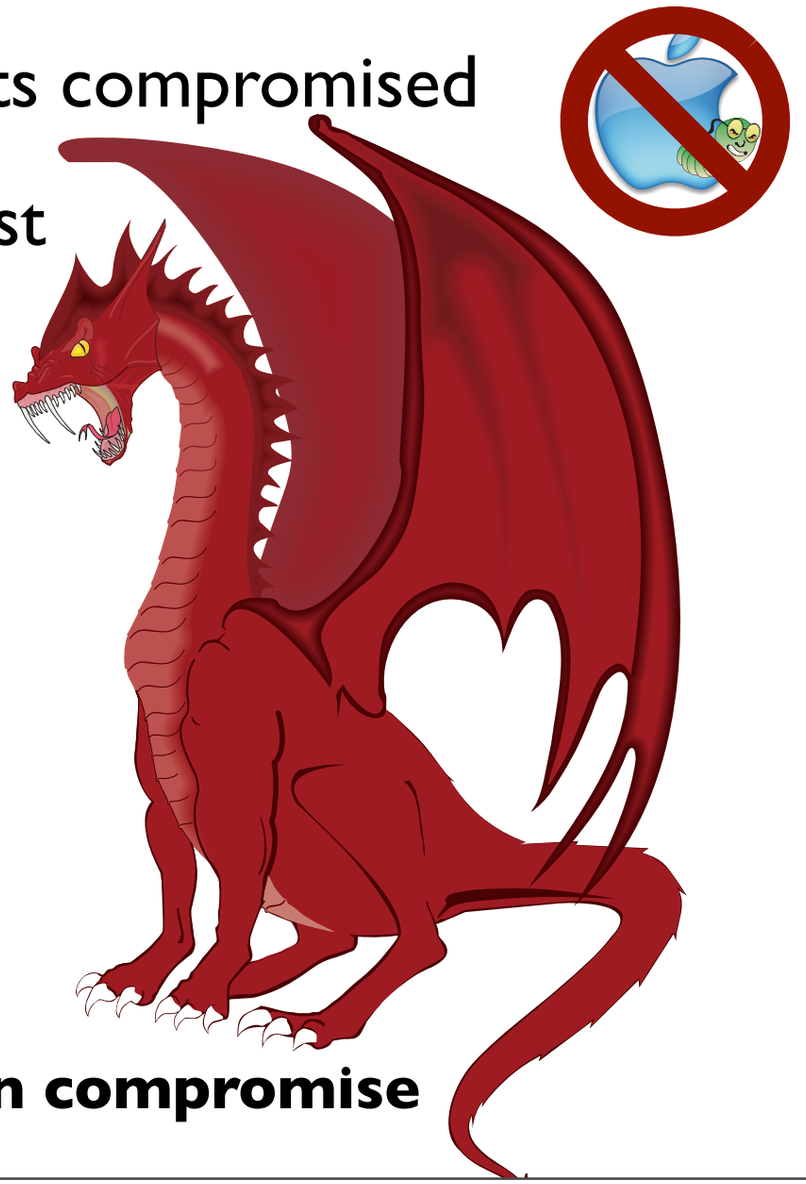
- **Goal:** to aid secure protocol design
  - designer only needs to consider restricted security threats
- Automatic protocol transformation adding ZK proofs

  - Enforce authenticity even if participants compromised
  - Preserve secrecy if everybody is honest

designer can assume attacker cannot compromise participants (easier)

Contributions

- **Goal:** to aid secure protocol design
  - designer only needs to consider restricted security threats
- Automatic protocol transformation adding ZK proofs
  - Enforce authenticity even if participants compromised
  - Preserve secrecy if everybody is honest



transformed protocol secure against attacker that can compromise

Contributions

- **Goal:** to aid secure protocol design
  - designer only needs to consider restricted security threats
- Automatic protocol transformation adding ZK proofs

  - Enforce authenticity even if participants compromised
  - Preserve secrecy if everybody is honest

- Automatic verification of the generated protocols

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

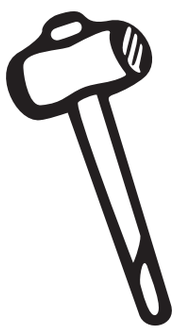
Contributions

- **Goal:** to aid secure protocol design and implementation
  - designer only needs to consider restricted security threats
- Automatic protocol transformation adding ZK proofs

  - Enforce authenticity even if participants compromised
  - Preserve secrecy if everybody is honest

- Automatic verification of the generated protocols

  - We use type system for ZK [Backes, Hritcu & Maffei, CCS '08]
    - now extended to handle security despite compromise
- Automatic code generation

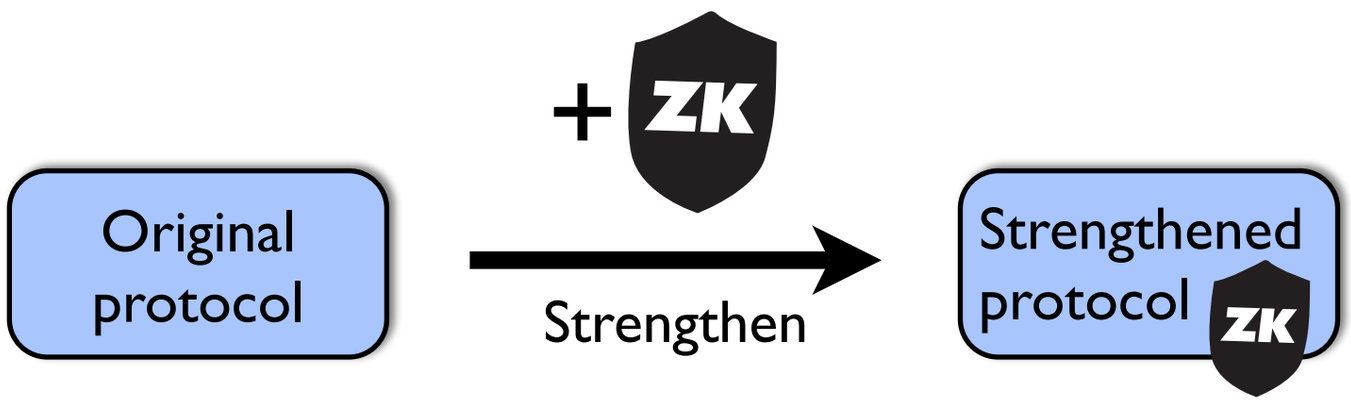


  - Spi2RCF: from Spi calculus to ML fragment

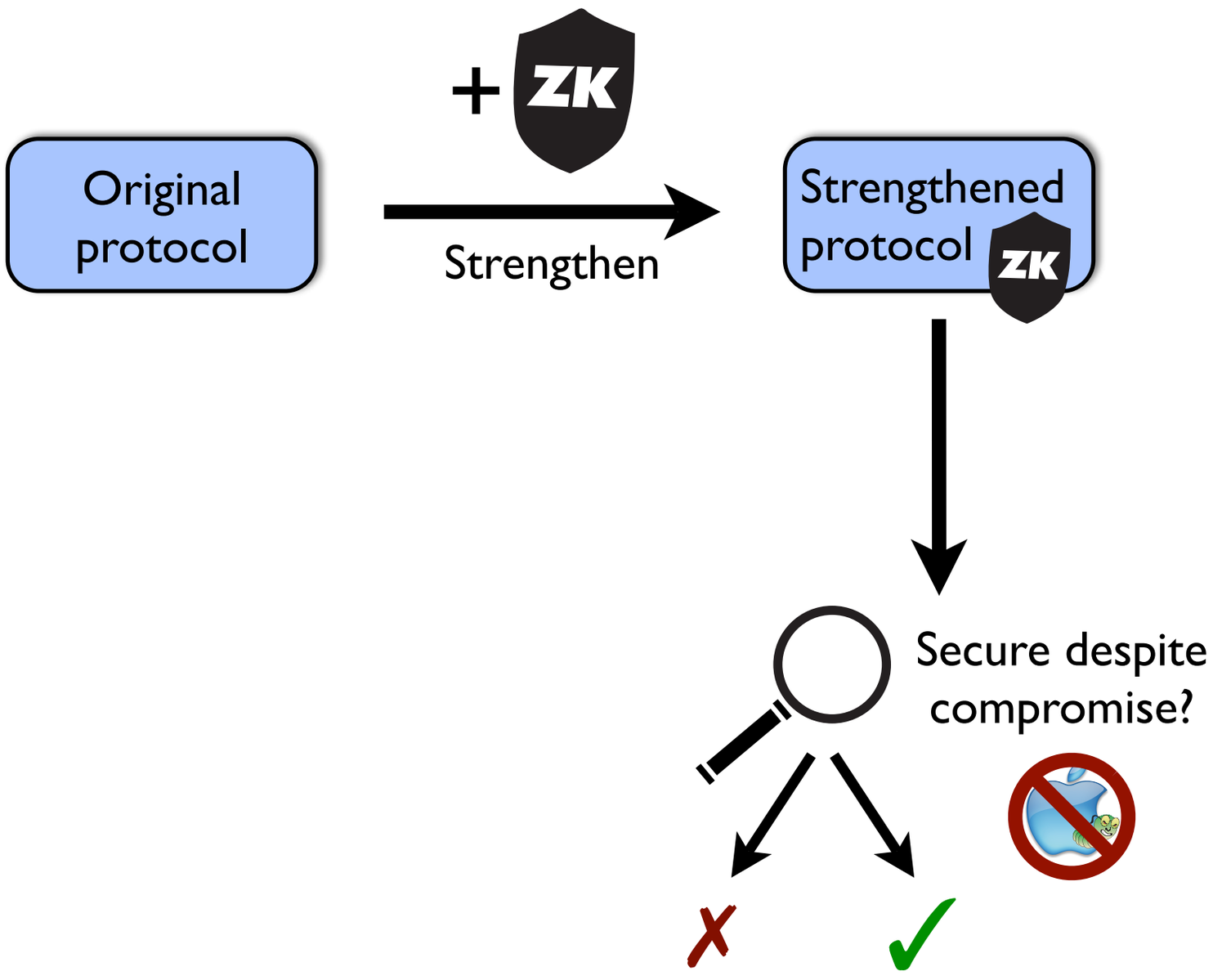
The big picture

Original protocol

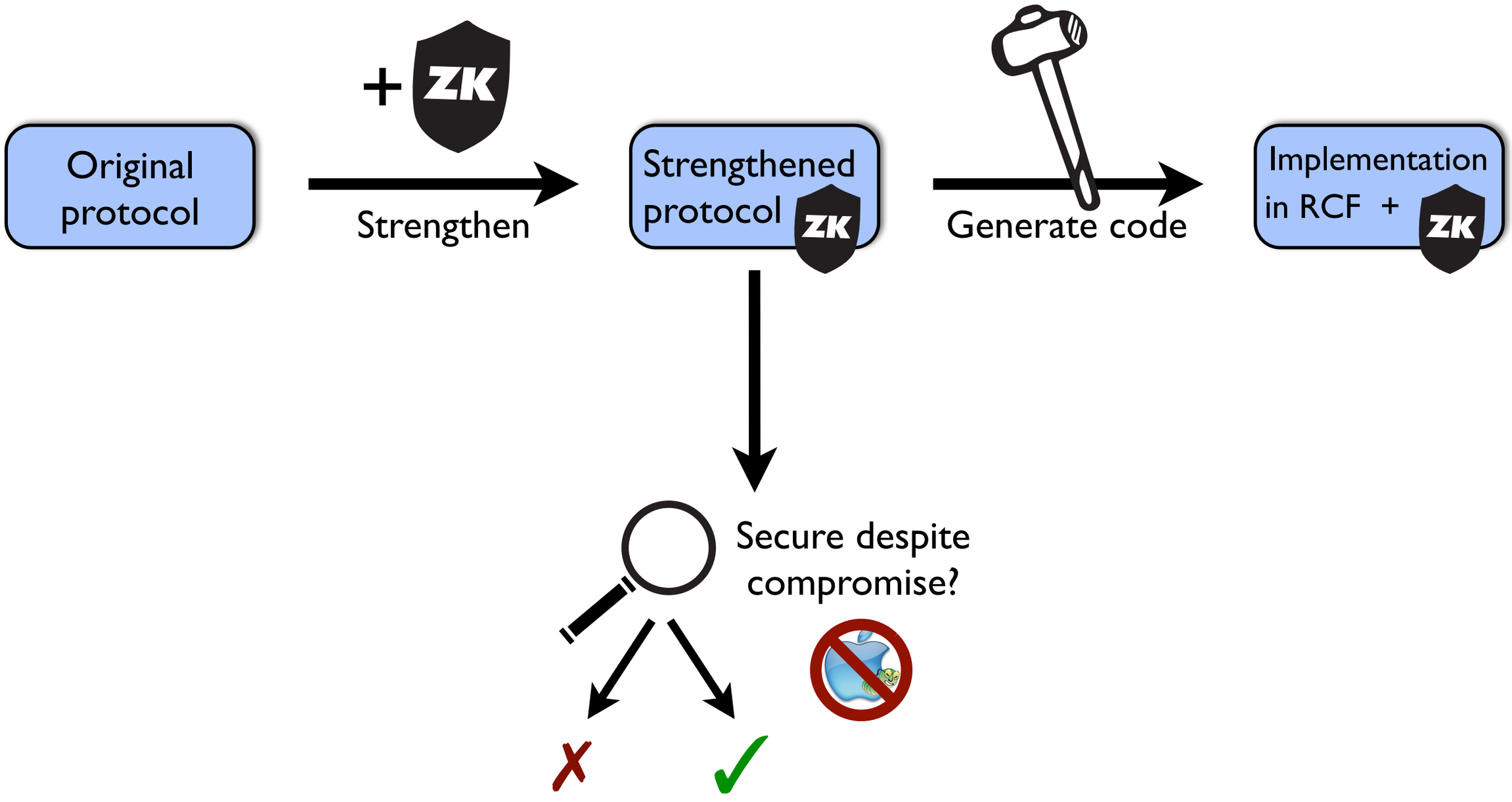
The big picture



The big picture

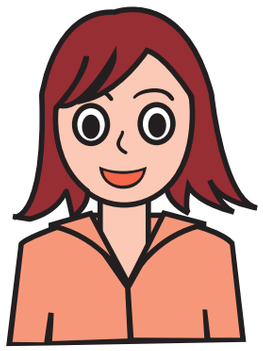


The big picture



A simple protocol

proxy



user

store

A simple protocol

proxy

user

store

(u,q,p_{wd})

A simple protocol

proxy

user

store

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

A simple protocol

proxy

user

store

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

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

A simple protocol

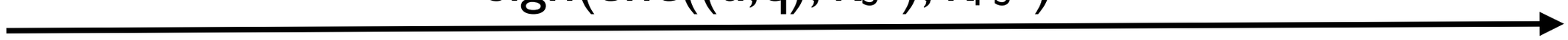
proxy

user

store

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

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



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

A simple protocol

proxy

user

store

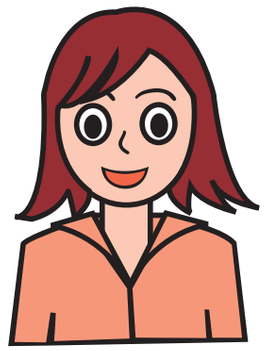
$sign(enc((u,q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

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

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

A simple protocol

proxy



user

store

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

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

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

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

A simple protocol

proxy

user

store

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

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

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

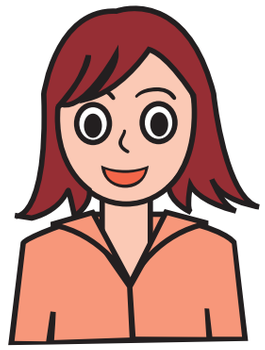
- This protocol is secure if all participants are honest (p_{wd} is secret and q is authentic)
- .... but insecure if the proxy is compromised
  - compromised proxy can leak p_{wd} (unavoidable)
  - **compromised proxy can fake request from the user (break authenticity)**

Transformation

+ ZK

Trying to strengthen the protocol

proxy



user

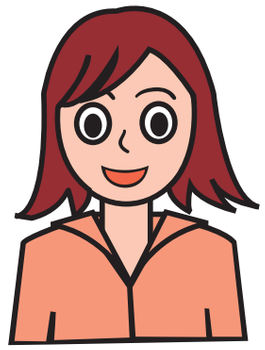
store

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

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

Trying to strengthen the protocol

proxy



user

store

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

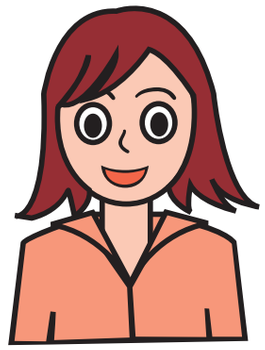
forward request

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

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

Trying to strengthen the protocol

proxy



user

store

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

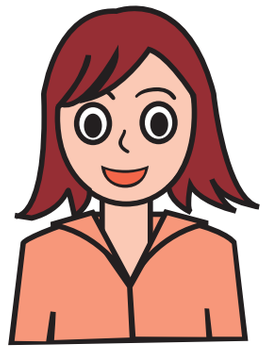
forward request

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

- 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

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

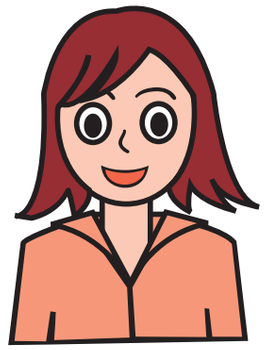
forward request

$\text{sign}(\text{enc}((u,\mathbf{q_{bad}}), k_S^{+}), k_{PS}^{-}), \text{sign}(\text{enc}((u,q,p_{wd}), k_{PE}^{+}), k_U^{-})$

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

Using non-interactive ZK

proxy



user

store

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

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

Using non-interactive ZK

proxy

user

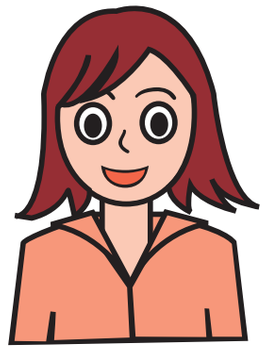
store

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

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

Using non-interactive ZK

proxy



user

store

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

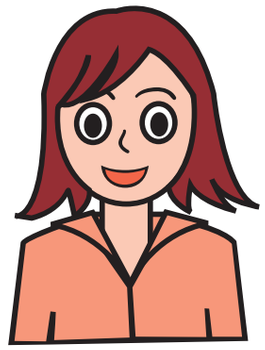
$\text{zk}_S(k_{PE}^{-}, q, p_{wd};\ \text{sign}(\text{enc}((u,q), k_S^{+}), k_{PS}^{-}), \text{sign}(\text{enc}((u,q, p_{wd}), k_{PE}^{+}), k_U^{-}), u)$

secret witnesses

[Backes, Maffei & Unruh, S&P '08]

Using non-interactive ZK

proxy



user

store

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

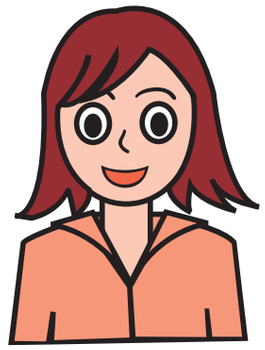
$\text{zk}_S(k_{PE}^{-}, q, p_{wd};\ \text{sign}(\text{enc}((u,q), k_S^{+}), k_{PS}^{-}), \text{sign}(\text{enc}((u,q, p_{wd}), k_{PE}^{+}), k_U^{-}), u)$

public terms

[Backes, Maffei & Unruh, S&P '08]

Using non-interactive ZK

proxy



user

store

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

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

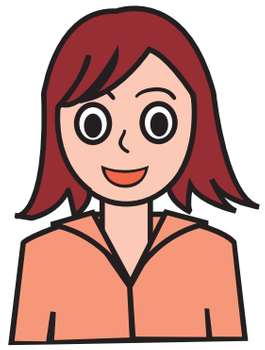
statement (= Boolean formula over equalities between terms with placeholders)

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

[Backes, Maffei & Unruh, S&P '08]

Using non-interactive ZK

proxy



user

store

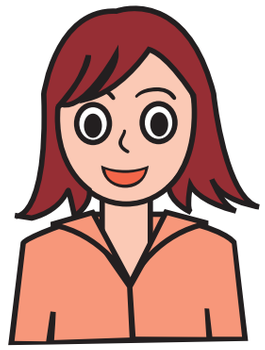
$\text{sign}(\text{enc}((q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

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

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

Using non-interactive ZK

proxy



user

store

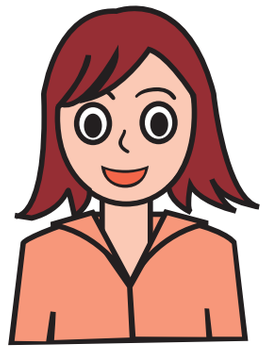
$\text{sign}(\text{enc}((q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

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

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

Using non-interactive ZK

proxy



user

store

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

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

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

Using non-interactive ZK

proxy

user

store

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

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

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

Using non-interactive ZK

proxy

user

store

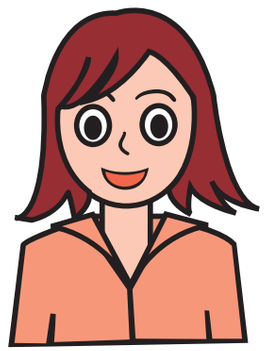
$\text{sign}(\text{enc}((q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

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

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

Using non-interactive ZK

proxy



user

store

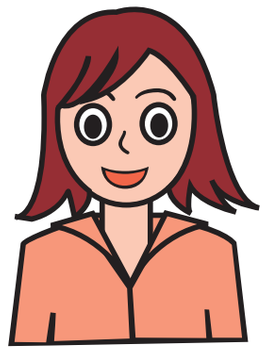
$\text{sign}(\text{enc}((q,p_{wd}), k_{PE}^{+}), k_{U}^{-})$

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

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

Using non-interactive ZK

proxy



user

store

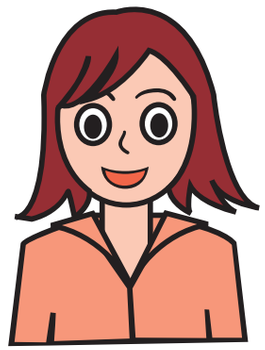
$\text{sign}(\text{enc}((q,p_{wd}), k_{PE}^{+}), k_U^{-})$

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

- 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

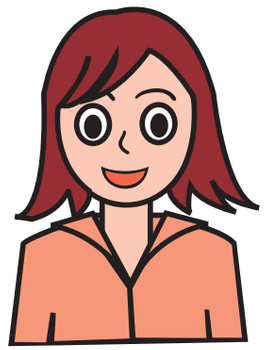
$\text{sign}(\text{enc}((q,p_{wd}), k_{PE}^{+}), k_U^{-})$

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

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

Using non-interactive ZK

proxy



user

store

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

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

- The proxy has to prove that its message is correctly generated from a request he received from the user
  - Compromised proxy can no longer cheat
- No secret data is revealed if everybody is honest

proxy

user

store

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

proxy

user

store

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

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

let proxy =
in(c_1, x);
let (=u, x_q, =p_{wd}) = dec(check(x, k_U^{+}), k_{PE}^{-}) **in**
…

proxy

user

store

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

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

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

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

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



proxy user store

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

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

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

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

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)

Transforming Processes

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)

Transforming Processes

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}^-), ,).

let store = ...

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

Transforming Processes

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(, , ; 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)

Transforming Processes

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(, , ; 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)

Transforming Processes

prove that message is correctly generated

stmt S = check(β_1,k_{PS}^+) = enc((u , x_q), 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(, , ; 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)

Transforming Processes

Secrecy analysis

public terms: c_1, c_2, u, x, k_{PS}^{+}, k_S^{+}, k_U^{+}

secret terms: x_q, p_{wd}, k_{PE}^{-}, k_{PS}^{-}

stmt *S* = check(β_1,k_{PS}^{+}) = enc((u , x_q), 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(, , ; 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)

Transforming Processes

public terms: c_1, c_2, u, x, k_{PS}^{+}, k_S^{+}, k_U^{+}

secret terms: x_q, p_{wd}, k_{PE}^{-}, k_{PS}^{-}

stmt *S* = check(β_1,k_{PS}^{+}) = enc((u , x_q), 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(, , ; sign(enc((u,x_q), k_S^{+}), k_{PS}^{-}), , u).

let store = ...

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

Transforming Processes

public terms: c_1, c_2, u, x, k_{PS}^{+}, k_S^{+}, k_U^{+}

secret terms: x_q, p_{wd}, k_{PE}^{-}, k_{PS}^{-}

stmt *S* = check(β_1,k_{PS}^{+}) = enc((β_3, x_q), 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(, , ; sign(enc((u,x_q), k_S^{+}), k_{PS}^{-}), , u).

let store = ...

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

Transforming Processes

public terms: c_1, c_2, u, x, k_{PS}^{+}, k_S^{+}, k_U^{+}

secret terms: x_q, p_{wd}, k_{PE}^{-}, k_{PS}^{-}

stmt *S* = check(β_1,k_{PS}^{+}) = enc((β_3, x_q), 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}^{-}), , u).

let store = …

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

Transforming Processes

public terms: c_1, c_2, u, x, k_{PS}^{+}, k_S^{+}, k_U^{+}

secret terms: x_q, p_{wd}, k_{PE}^{-}, k_{PS}^{-}

stmt *S* = check(β_1,k_{PS}^{+}) = enc((β_3,α_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}^{-}), , u).

let store = …

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

Transforming Processes

public terms: c_1, c_2, u, x, k_{PS}^{+}, k_S^{+}, k_U^{+}

secret terms: x_q, p_{wd}, k_{PE}^{-}, k_{PS}^{-}

stmt *S* = check(β_1,k_{PS}^{+}) = enc((β_3,α_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}^{-}), , u).

let store = …

term that depends on previous input

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

Transforming Processes

public terms: c_1, c_2, u, x, k_{PS}^{+}, k_S^{+}, k_U^{+}

secret terms: x_q, p_{wd}, k_{PE}^{-}, k_{PS}^{-}

stmt *S* = check(β_1,k_{PS}^{+}) = enc((β_3,α_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}^{-}), , u).

let store = ...

term that depends on previous input

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

Transforming Processes

We represent precise dependency symbolically

public terms: c_1, c_2, u, x, k_{PS}^{+}, k_S^{+}, k_U^{+}

secret terms: x_q, p_{wd}, k_{PE}^{-}, k_{PS}^{-}

stmt S = check(β_1,k_{PS}^{+}) = enc((β_3,α_2), k_S^{+}) $\wedge$ 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(, x_q, ; sign(enc((u,x_q), k_S^{+}), k_{PS}^{-}), , u).

let store = ...

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

Transforming Processes

public terms: c_1, c_2, u, x, k_{PS}^{+}, k_S^{+}, k_U^{+}

secret terms: x_q, p_{wd}, k_{PE}^{-}, k_{PS}^{-}

stmt S = check(β_1,k_{PS}^{+}) = enc((β_3,α_2), k_S^{+}) ∧ 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$_S$(, x_q, ; sign(enc((u,x_q), k_S^{+}), k_{PS}^{-}), x, u).

let store = ...

forward previous input

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

Transforming Processes

public terms: c_1, c_2, u, x, k_{PS}^{+}, k_S^{+}, k_U^{+}

secret terms: x_q, p_{wd}, k_{PE}^{-}, k_{PS}^{-}

stmt S = check(β_1,k_{PS}^{+}) = enc((β_3,α_2), k_S^{+}) $\wedge$ dec(check(β_2, 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(, x_q, ; 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)

Transforming Processes

public terms: c_1, c_2, u, x, k_{PS}^{+}, k_S^{+}, k_U^{+}

secret terms: x_q, p_{wd}, k_{PE}^{-}, k_{PS}^{-}

stmt S = check(β_1,k_{PS}^{+}) = enc((β_3,α_2), k_S^{+}) $\wedge$ dec(check(β_2, 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(k_{PE}^{-}, x_q, ; 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)

Transforming Processes

Asymmetry caused by k_S^- being unknown to the proxy

public terms: c_1, c_2, u, x, k_{PS}^+, k_S^+, k_U^+

secret terms: x_q, p_{wd}, k_{PE}^-, k_{PS}^-

stmt S = check(β_1,k_{PS}^+) = enc((β_3,α_2), k_S^+) $\wedge$ dec(check(β_2, 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(k_{PE}^-, x_q, ; 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)

Transforming Processes

public terms: c_1, c_2, u, x, k_{PS}^{+}, k_S^{+}, k_U^{+}

secret terms: x_q, p_{wd}, k_{PE}^{-}, k_{PS}^{-}

stmt S = check(β_1,k_{PS}^{+}) = enc((β_3,α_2), k_S^{+}) $\wedge$ dec(check(β_2, k_U^{+}), α_1) = (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(k_{PE}^{-}, x_q, ; 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)

Transforming Processes

public terms: c_1, c_2, u, x, k_{PS}^{+}, k_S^{+}, k_U^{+}

secret terms: x_q, p_{wd}, k_{PE}^{-}, k_{PS}^{-}

stmt S = check(β_1,k_{PS}^{+}) = enc((β_3,α_2), k_S^{+}) ∧ dec(check(β_2, k_U^{+}), α_1) = (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(k_{PE}^{-}, x_q, p_{wd}; 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)

Transforming Processes

Shows that outputs correctly constructed from inputs

stmt S = check(β_1,k_{PS}^+) = enc((β_3,α_2), k_S^+) $\wedge$ 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(k_{PE}^-, x_q, p_{wd}; 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)

Transforming Processes

stmt S = check(β_1,k_{PS}^+) = enc((β_3,α_2), k_S^+) $\wedge$ 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(k_{PE}^-, x_q, p_{wd}; sign(enc((u,x_q), k_S^+), k_{PS}^-), x, u).

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')

Transforming Processes

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

let user = ...

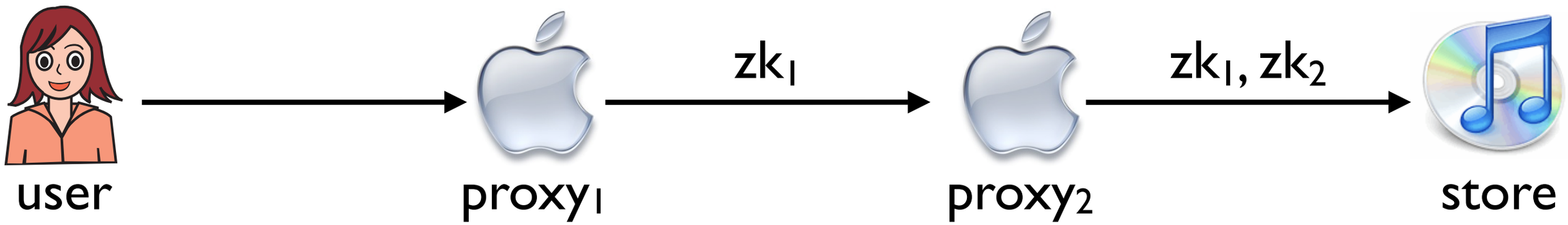
let proxy'=
in(c_1, x);
let (=u, x_q, =p_{wd}) = dec(check(x, k_U^+), k_{PE}^-) **in**
out(c_2, zk_S(k_{PE}^-, x_q, p_{wd}; sign(enc((u,x_q), k_S^+), k_{PS}^-), x, u).

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**
...

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

Further complications

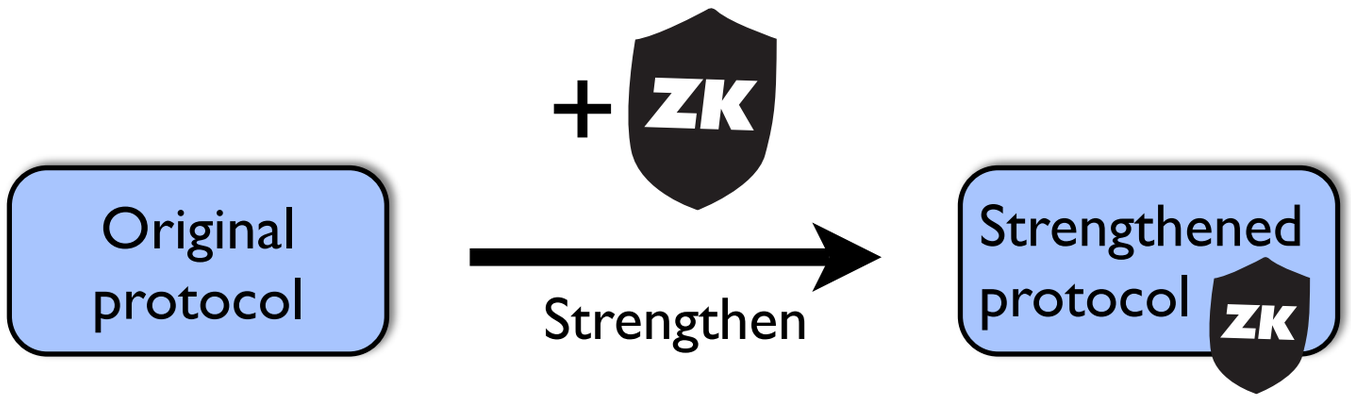
- Forwarding zero-knowledge proofs
  - Ensure correct behavior of all protocol participants



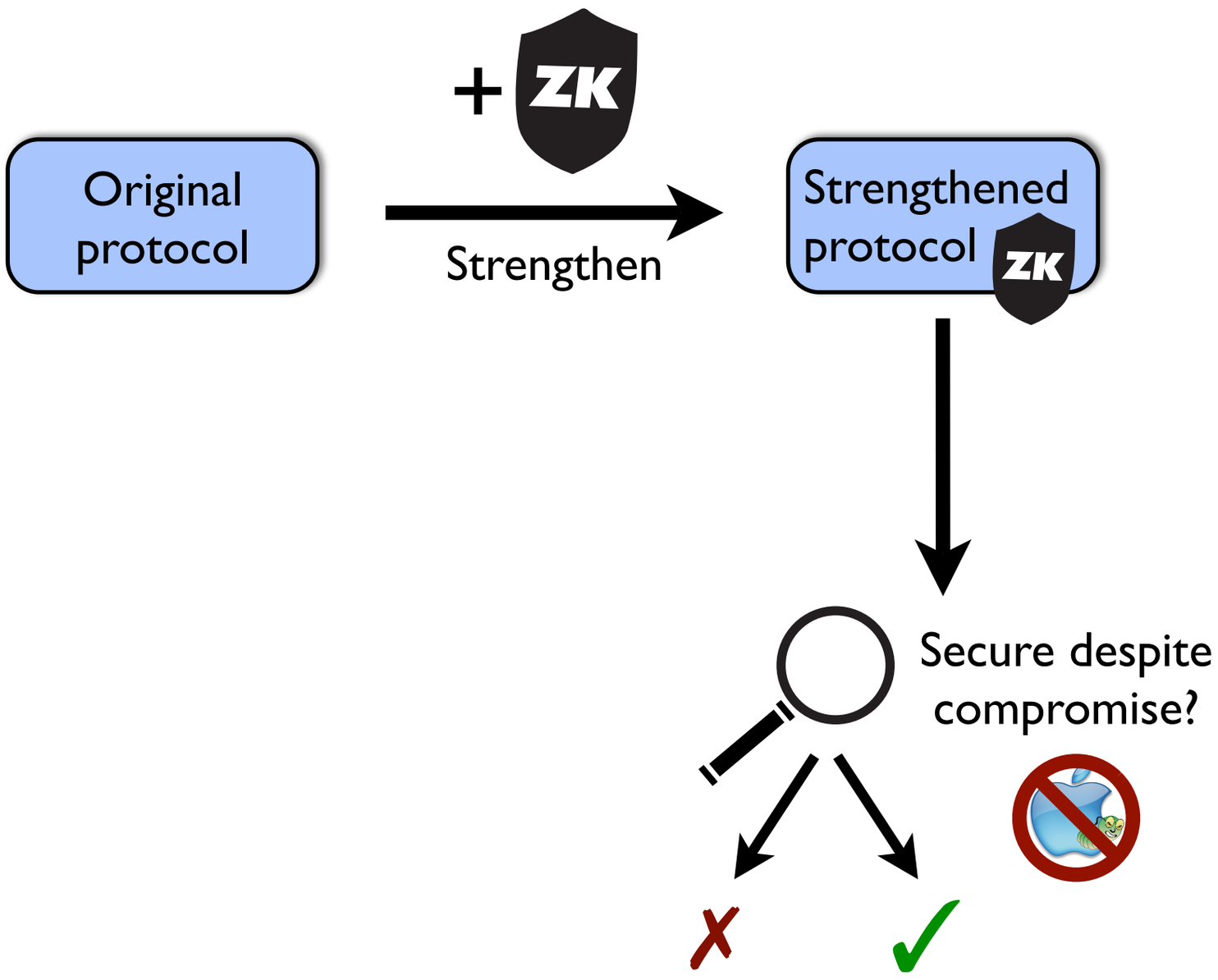
- Symmetric encryption: add proof of identity
  - digital signature or ZK proof (can preserve some anonymity)
- Transforming types

Type System

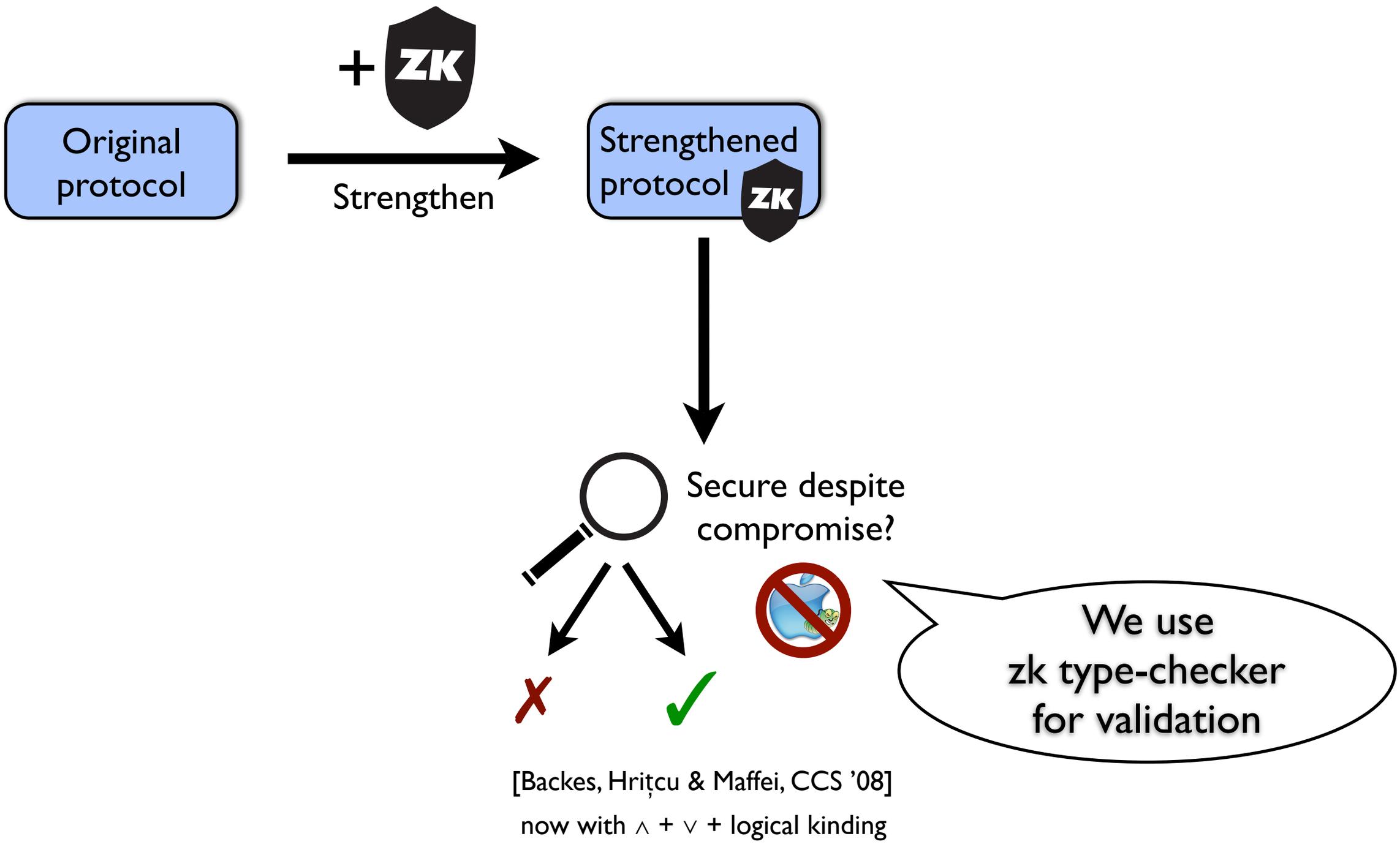
Translation validation



Translation validation



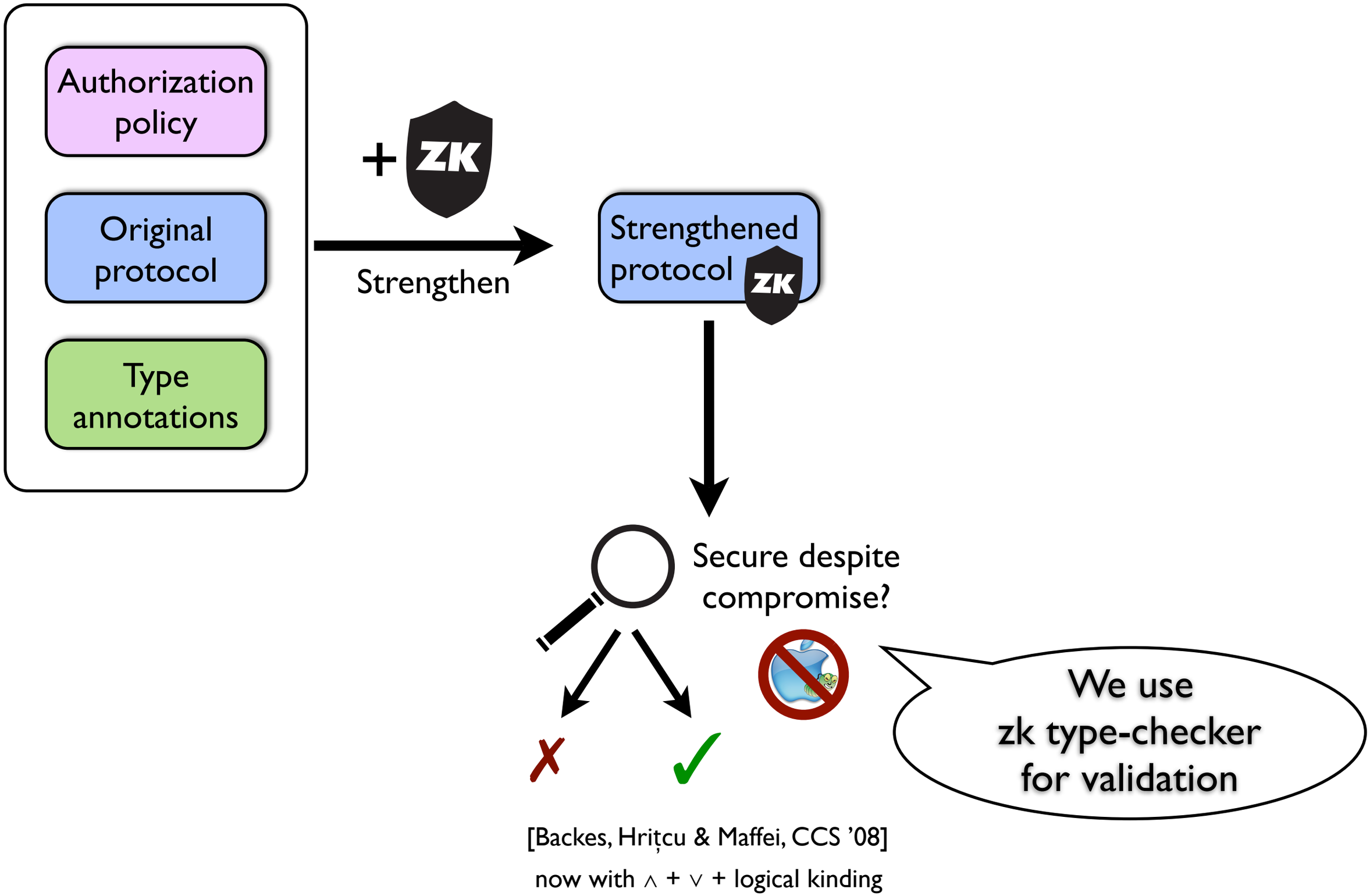
Translation validation



[Backes, Hriţcu & Maffei, CCS '08]

now with ∧ + ∨ + logical kinding

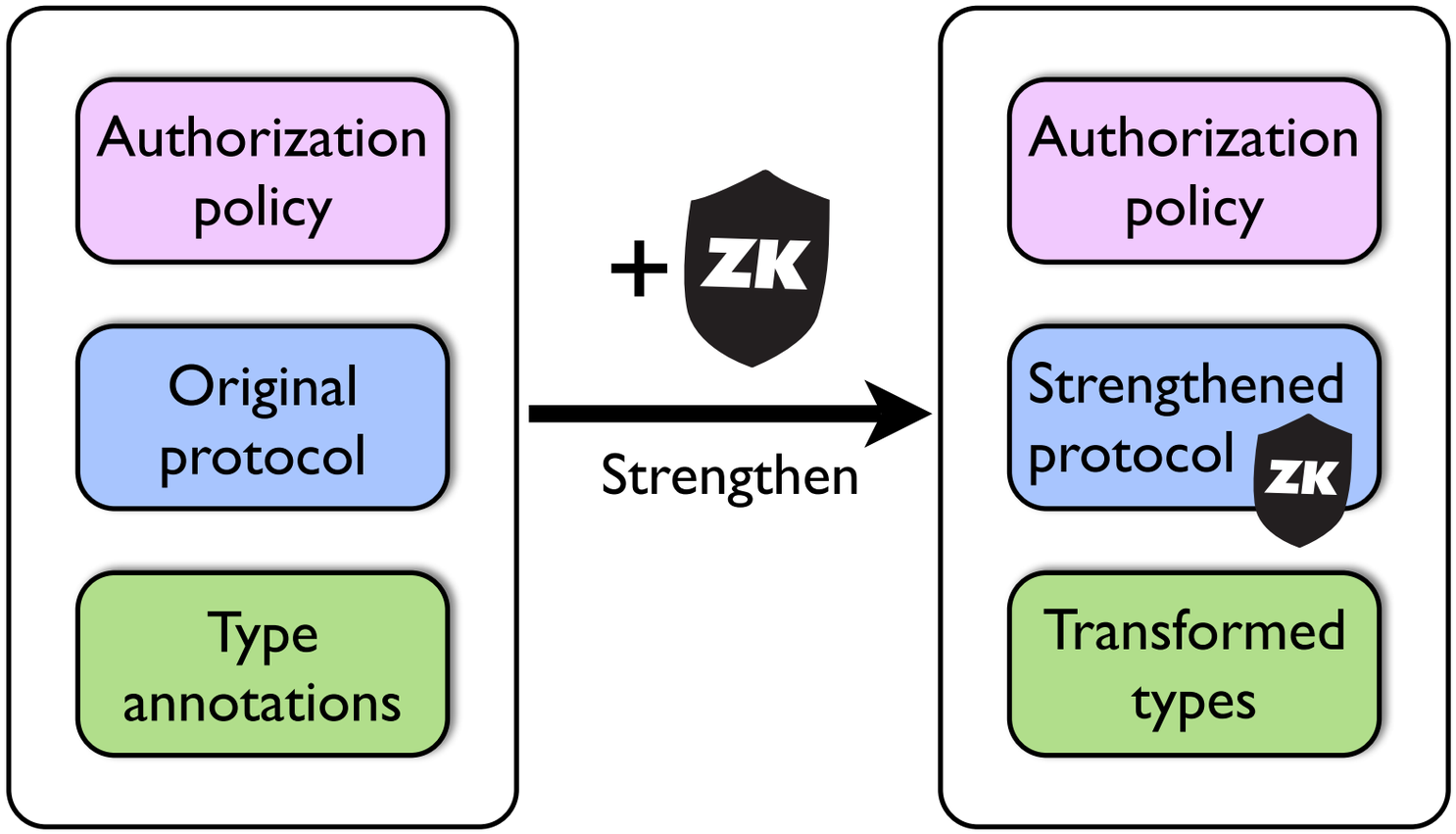
Translation validation

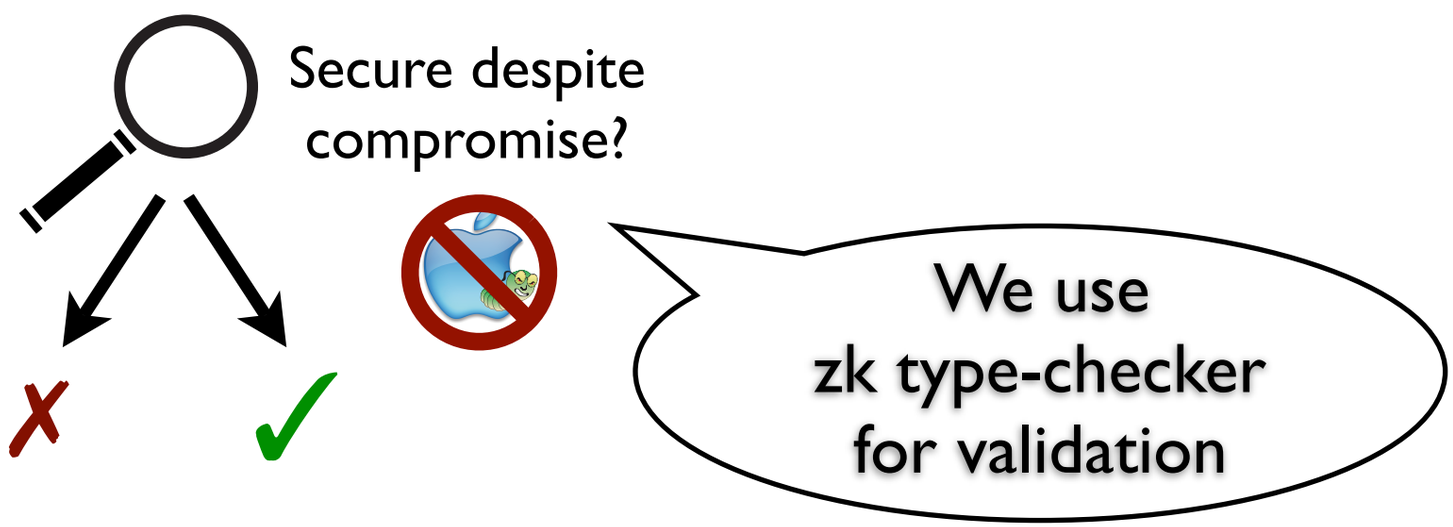


[Backes, Hriţcu & Maffei, CCS '08]

now with ∧ + ∨ + logical kinding

Translation validation

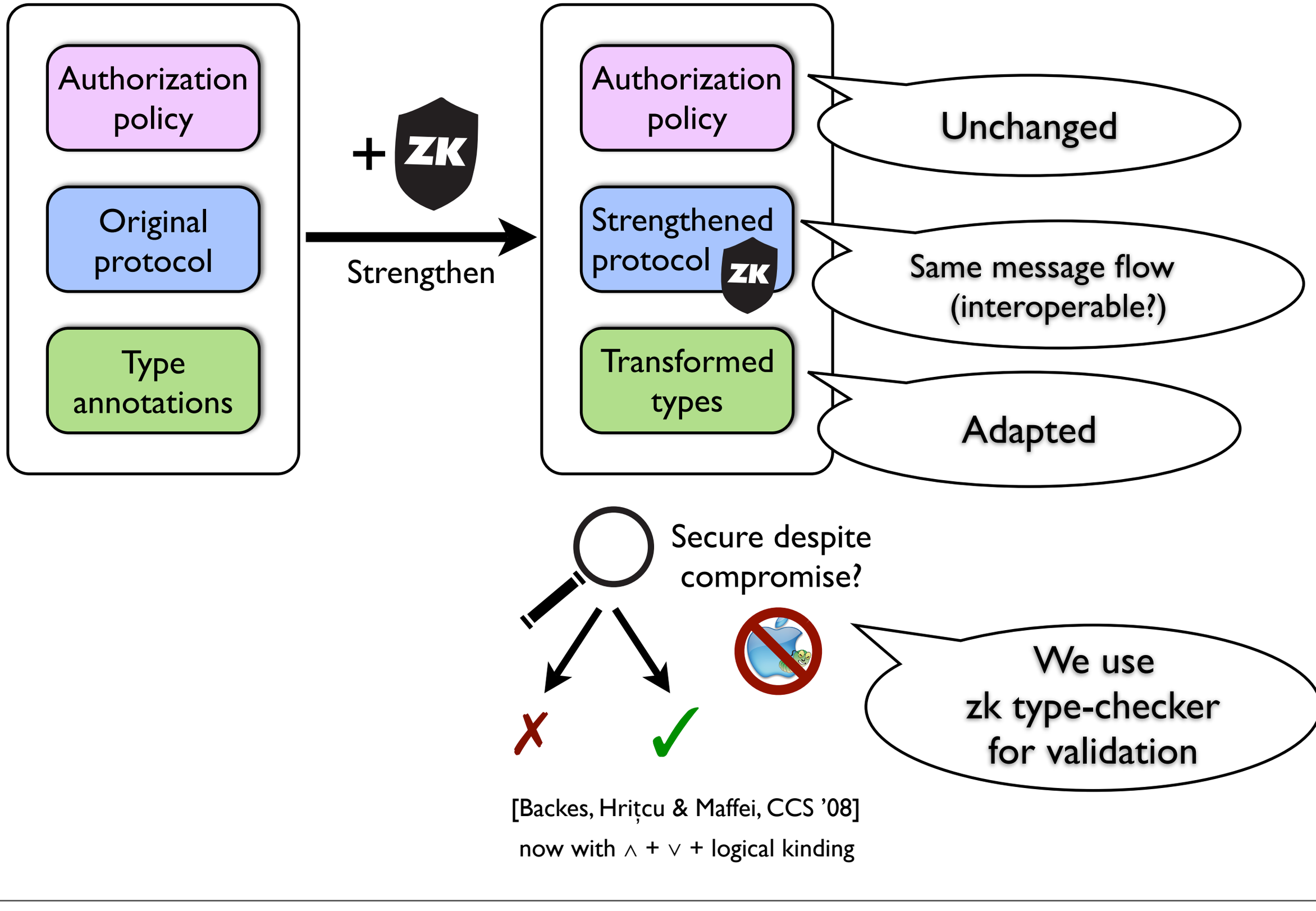




[Backes, Hriţcu & Maffei, CCS '08]

now with ∧ + ∨ + logical kinding

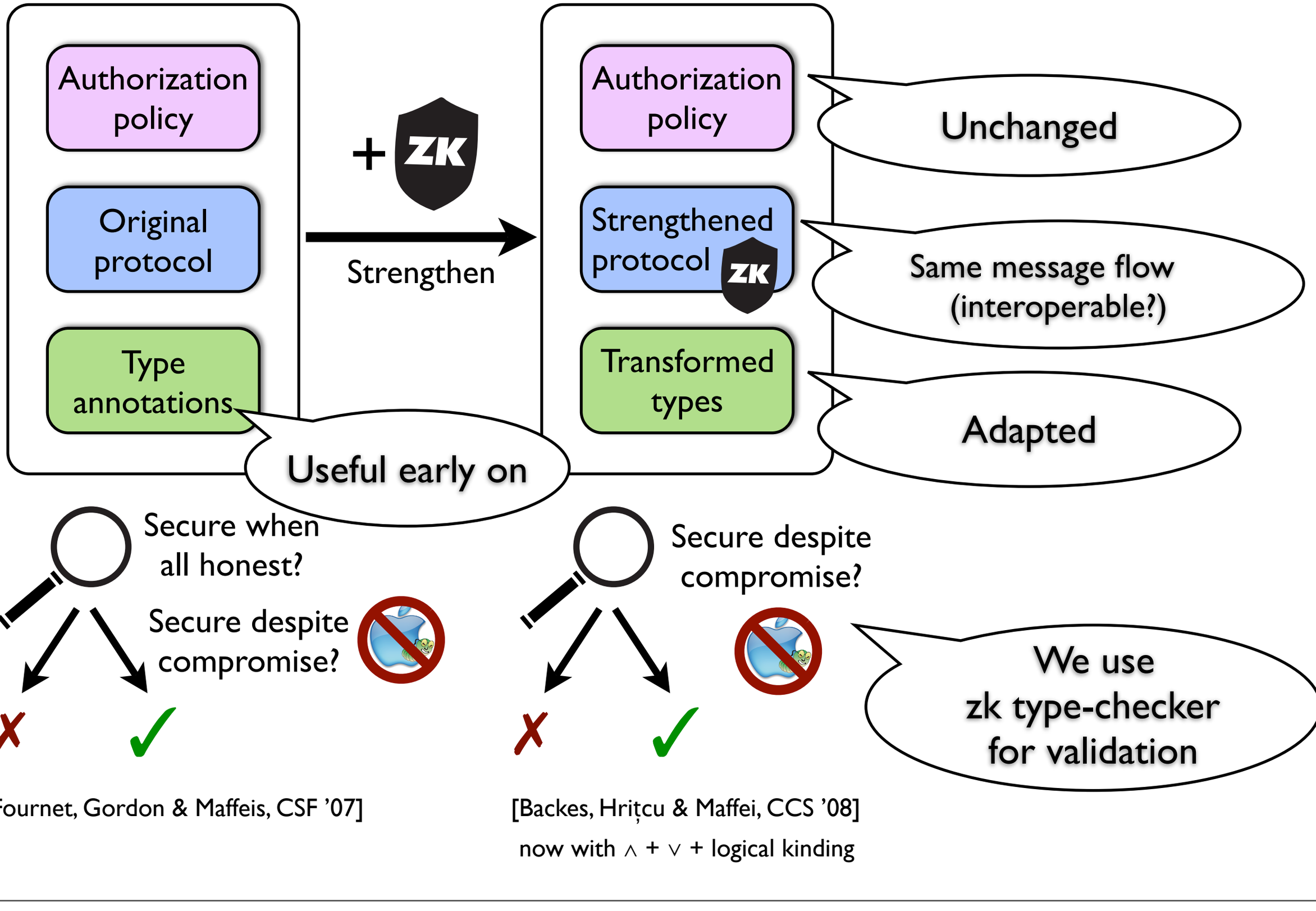
Translation validation



[Backes, Hriţcu & Maffei, CCS '08]

now with ∧ + ∨ + logical kinding

Translation validation



[Fournet, Gordon & Maffeis, CSF '07]

[Backes, Hriţcu & Maffei, CCS '08]

now with ∧ + ∨ + logical kinding

Type system

- Enhancement of [Backes, Hriţcu & Maffei, CCS ’08] which extends [Fournet, Gordon & Maffeis, CSF ’07]

Type system

- Enhancement of [Backes, Hriţcu & Maffei, CCS ’08] which extends [Fournet, Gordon & Maffeis, CSF ’07]
- Example types

Type system

- Enhancement of [Backes, Hriţcu & Maffei, CCS '08] which extends [Fournet, Gordon & Maffeis, CSF '07]
- Example types
  - u: Un

Public message - can be sent to the attacker

Type system

- Enhancement of [Backes, Hriţcu & Maffei, CCS ’08] which extends [Fournet, Gordon & Maffeis, CSF ’07]
- Example types
  - u: Un
  - p_{wd}: Private

Type system

- Enhancement of [Backes, Hriţcu & Maffei, CCS ’08] which extends [Fournet, Gordon & Maffeis, CSF ’07]
- Example types
  - u: Un
  - p_{wd}: Private
  - $T_2 = (x_u : \mathrm{Un}, \{x_q : \mathrm{Un} \mid \mathrm{Request}(x_u, x_q) \wedge \mathrm{Registered}(x_u)\})$
    (u,q): T_2

Refinement type - conveys logical formula

Type system

- Enhancement of [Backes, Hriţcu & Maffei, CCS ’08] which extends [Fournet, Gordon & Maffeis, CSF ’07]
- Example types
  - u: Un
  - p_{wd}: Private
  - $T_2 = (x_u : \mathrm{Un}, \{x_q : \mathrm{Un} \mid \mathrm{Request}(x_u, x_q) \wedge \mathrm{Registered}(x_u)\})$
    (u,q): T_2
  - k_{PS}^{-}: SigKey(PubEnc(T_2))

Key only used to sign encryptions of terms of type T_2 (if proxy is honest)

Type system

- Enhancement of [Backes, Hriţcu & Maffei, CCS ’08] which extends [Fournet, Gordon & Maffeis, CSF ’07]
- Example types
  - u: Un
  - p_{wd}: Private
  - $T_2 = (x_u : Un, \{x_q : Un \mid Request(x_u, x_q) \wedge Registered(x_u)\})$
    (u,q): T_2
  - k_{PS}^{-}: SigKey(PubEnc(T_2))
  - z: $ZKProof_S(y{:}T, \exists x.C)$

Zero-knowledge proof of statement *S*

Enhancements

- Added union and intersection types

Enhancements

- Added union and intersection types
  - Conditionally secure types (depending on compromise scenario)

    PrivateUnlessP = {Private | ¬Compromised(p)} ∨ {Un | Compromised(p)}

    p_{wd}: PrivateUnlessP

Enhancements

- Added union and intersection types
  - Conditionally secure types (depending on compromise scenario)

    PrivateUnlessP = {Private | ¬Compromised(p)} ∨ {Un | Compromised(p)}

    p_{wd}: PrivateUnlessP
- Logical characterization of type compromise
  - pub(PrivateUnlessP) = tnt(PrivateUnlessP) = Compromised(p)

Enhancements

- Added union and intersection types
  - Conditionally secure types (depending on compromise scenario)

    PrivateUnlessP = {Private | ¬Compromised(p)} ∨ {Un | Compromised(p)}

    p_{wd}: PrivateUnlessP
- Logical characterization of type compromise
  - pub(PrivateUnlessP) = tnt(PrivateUnlessP) = Compromised(p)
  - m: U ∨ { ⊤ | tnt(U)}

Enhancements

- Added union and intersection types
  - Conditionally secure types (depending on compromise scenario)

    PrivateUnlessP = {Private | ¬Compromised(p)} ∨ {Un | Compromised(p)}

    p_{wd}: PrivateUnlessP
- Logical characterization of type compromise
  - pub(PrivateUnlessP) = tnt(PrivateUnlessP) = Compromised(p)
  - m: U ∨ { ⊤ | tnt(U)}
- More precise type inference for witnesses of ZK proofs
  - stmt S = check(β_1,k_{PS}^+)=β_2 β_2: T k_{PS}^+:VerKey(U)

Enhancements

- Added union and intersection types
  - Conditionally secure types (depending on compromise scenario)

    PrivateUnlessP = {Private | ¬Compromised(p)} ∨ {Un | Compromised(p)}

    p_{wd}: PrivateUnlessP
- Logical characterization of type compromise
  - pub(PrivateUnlessP) = tnt(PrivateUnlessP) = Compromised(p)
  - m: U ∨ { ⊤ | tnt(U)}
- More precise type inference for witnesses of ZK proofs
  - stmt S = check(β_1,k_{PS}^+)=β_2 β_2: T k_{PS}^+:VerKey(U)
  - if check succeeds then β_2: T ∧ (U ∨ { ⊤ | tnt(U)})

Enhancements

- Added union and intersection types
  - Conditionally secure types (depending on compromise scenario)

    PrivateUnlessP = {Private | ¬Compromised(p)} ∨ {Un | Compromised(p)}

    p_{wd}: PrivateUnlessP
- Logical characterization of type compromise
  - pub(PrivateUnlessP) = tnt(PrivateUnlessP) = Compromised(p)
  - m: U ∨ { ⊤ | tnt(U)}
- More precise type inference for witnesses of ZK proofs
  - stmt S = check(β_1,k_{PS}^+)=β_2 β_2: T k_{PS}^+:VerKey(U)
  - if check succeeds then β_2: T ∧ (U ∨ { ⊤ | tnt(U)})
- These increase precision and flexibility of type system

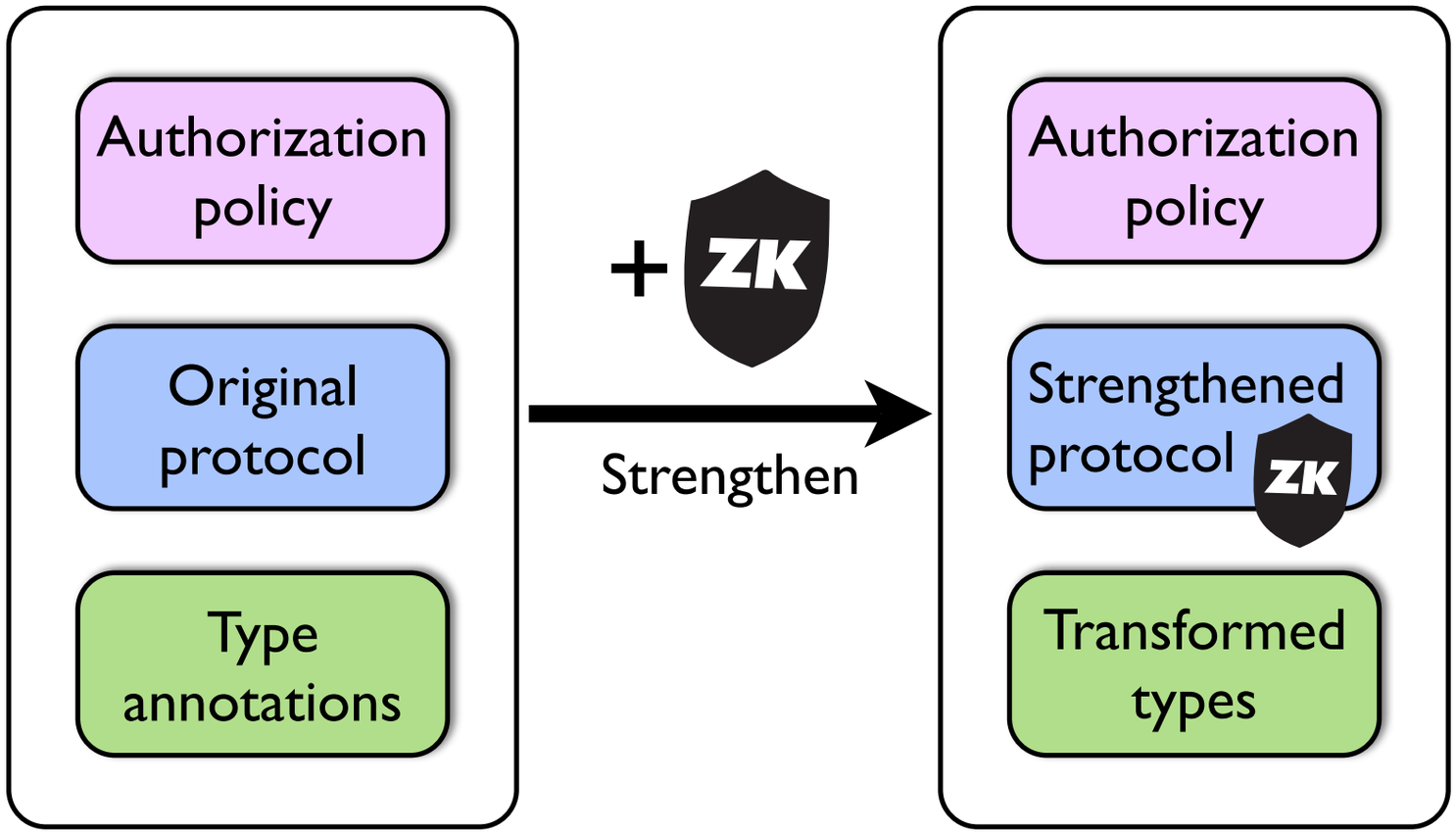
Enhancements

- Added union and intersection types
  - Conditionally secure types (depending on compromise scenario)

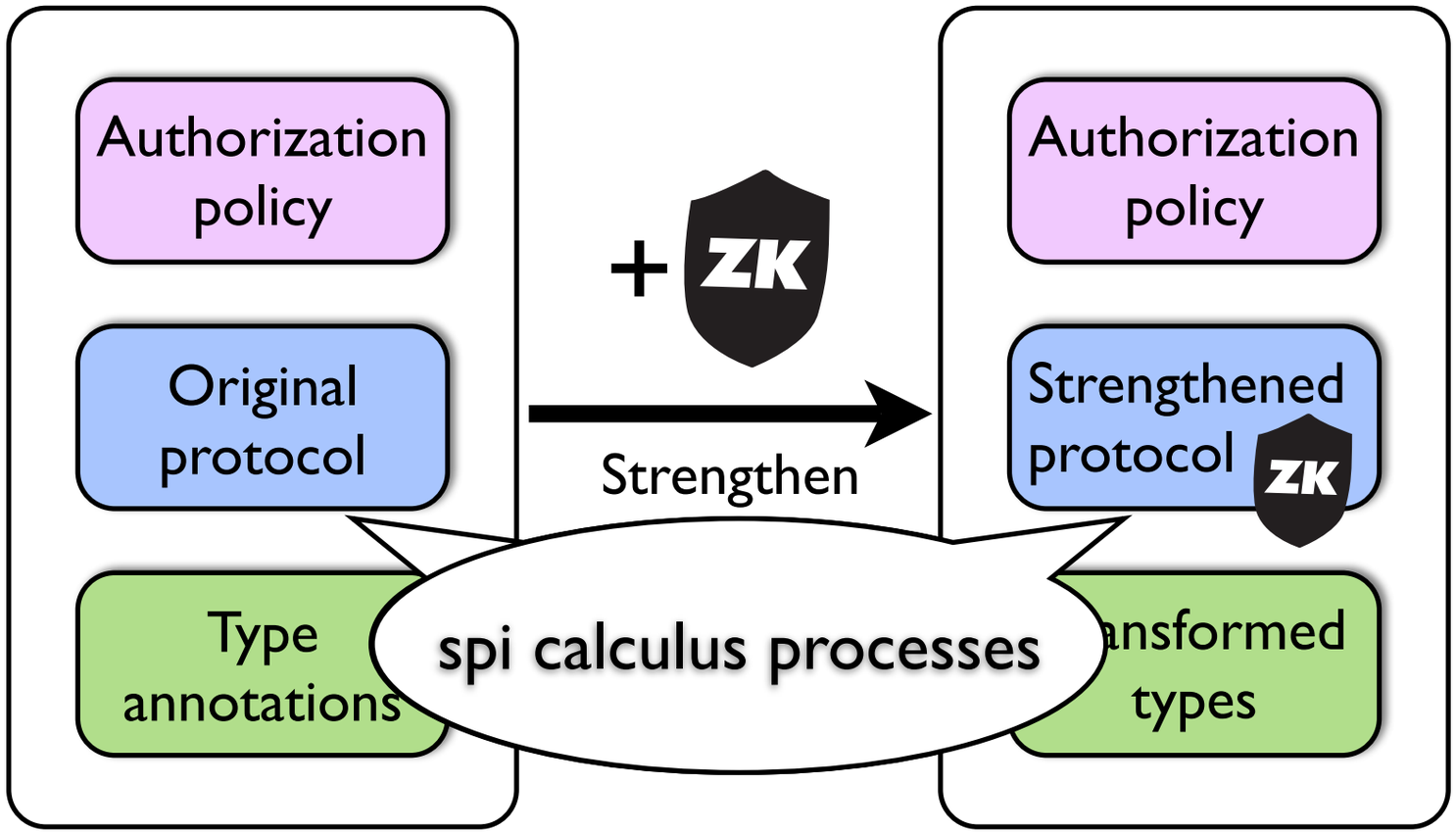
    PrivateUnlessP = {Private | ¬Compromised(p)} ∨ {Un | Compromised(p)}

    p_{wd}: PrivateUnlessP
- Logical characterization of type compromise
  - pub(PrivateUnlessP) = tnt(PrivateUnlessP) = Compromised(p)
  - m: U ∨ { ⊤ | tnt(U)}
- More precise type inference for witnesses of ZK proofs
  - stmt S = check(β_1,k_{PS}^+)=β_2 β_2: T k_{PS}^+:VerKey(U)
  - if check succeeds then β_2: T ∧ (U ∨ { ⊤ | tnt(U)})
- These increase precision and flexibility of type system

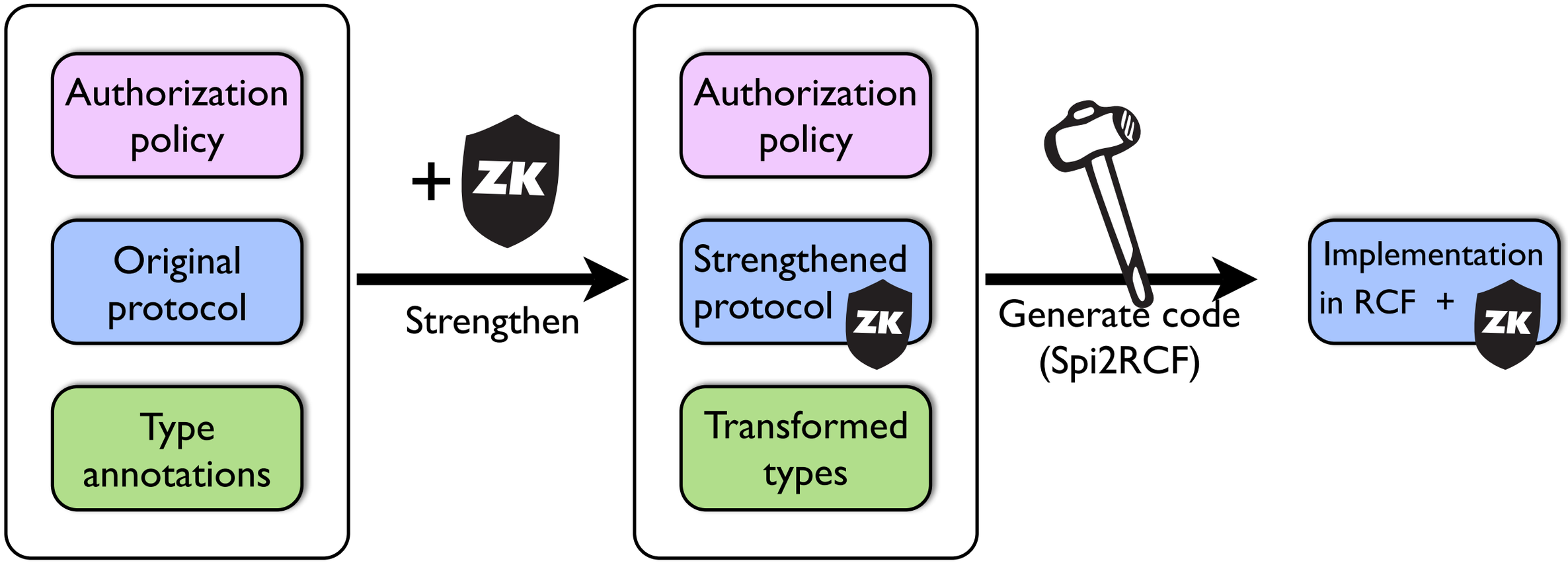
Automatic code generation



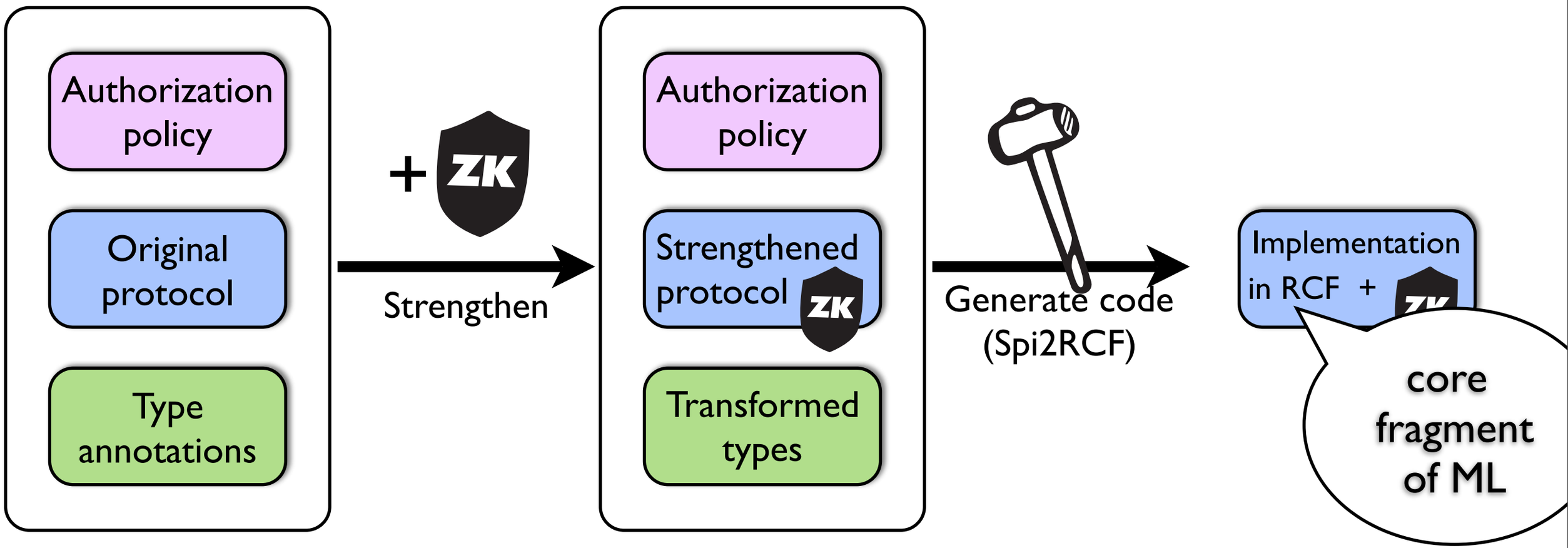
Automatic code generation



Automatic code generation

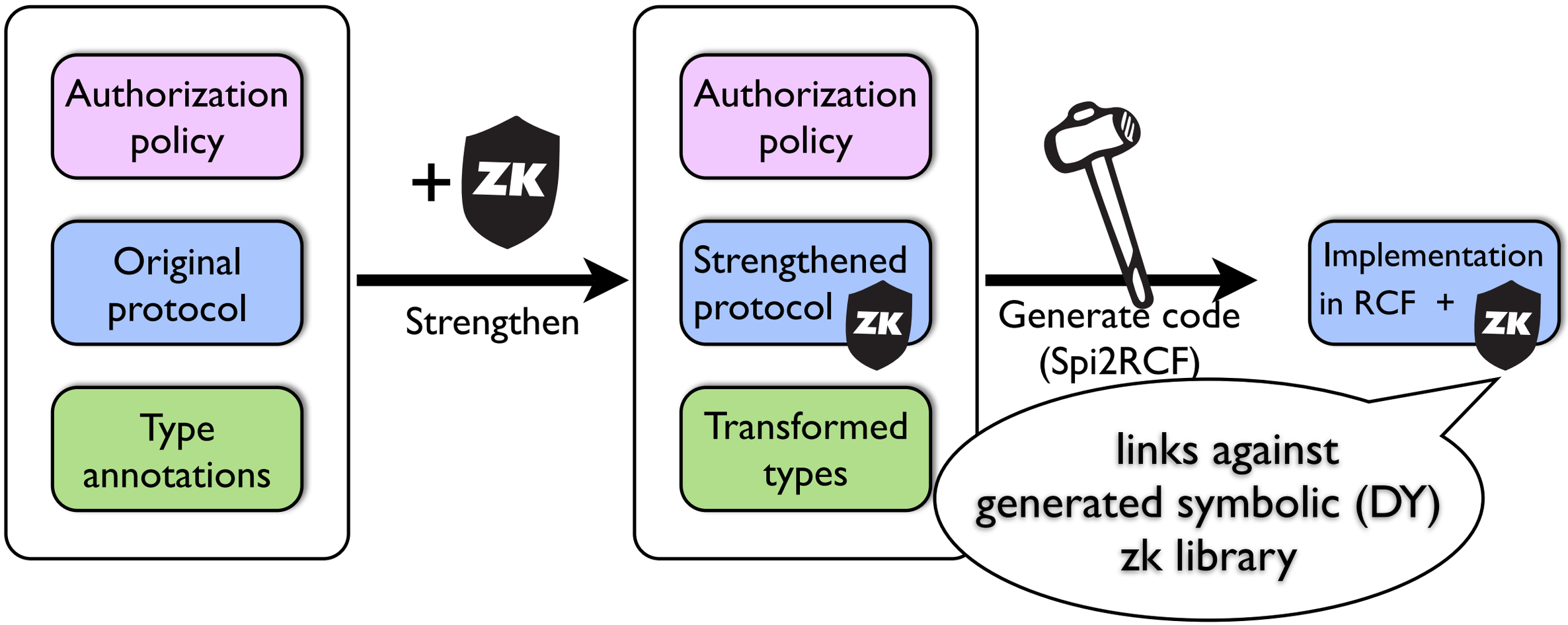


Automatic code generation



[Bengtson et. al., CSF '08]

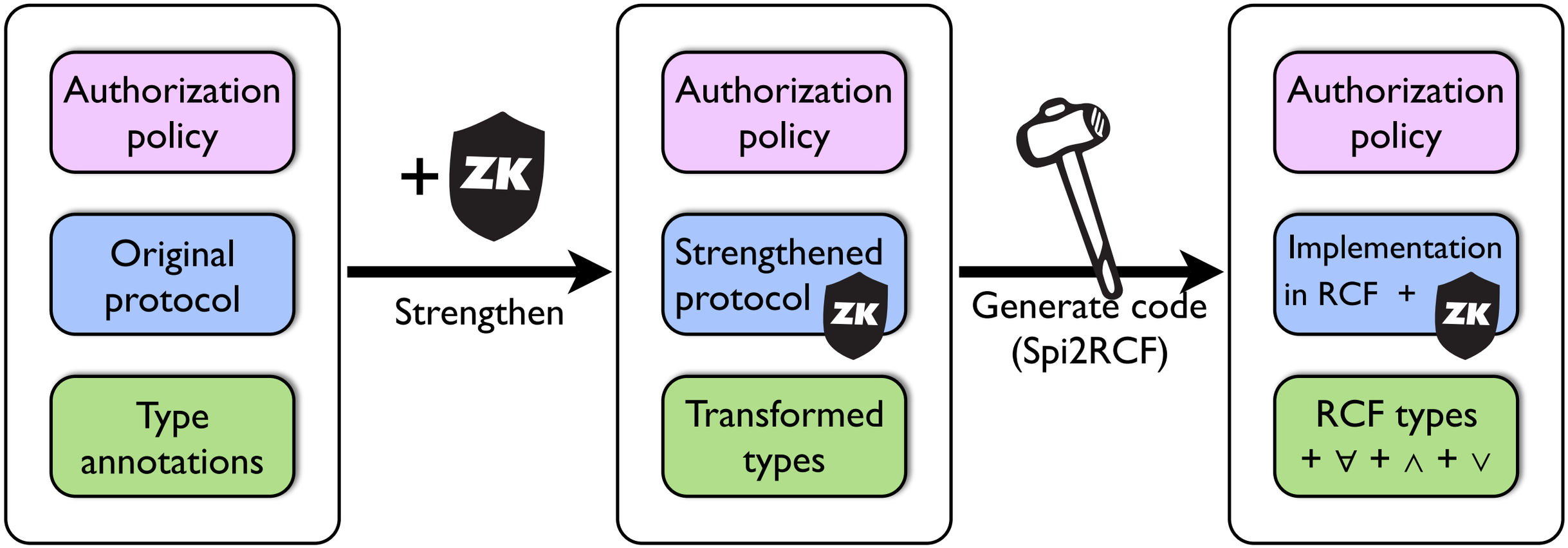
Automatic code generation



[Bengtson et. al., CSF '08]

[Backes, Hriţcu, Maffei & Tarrach, FCS '09 in August]

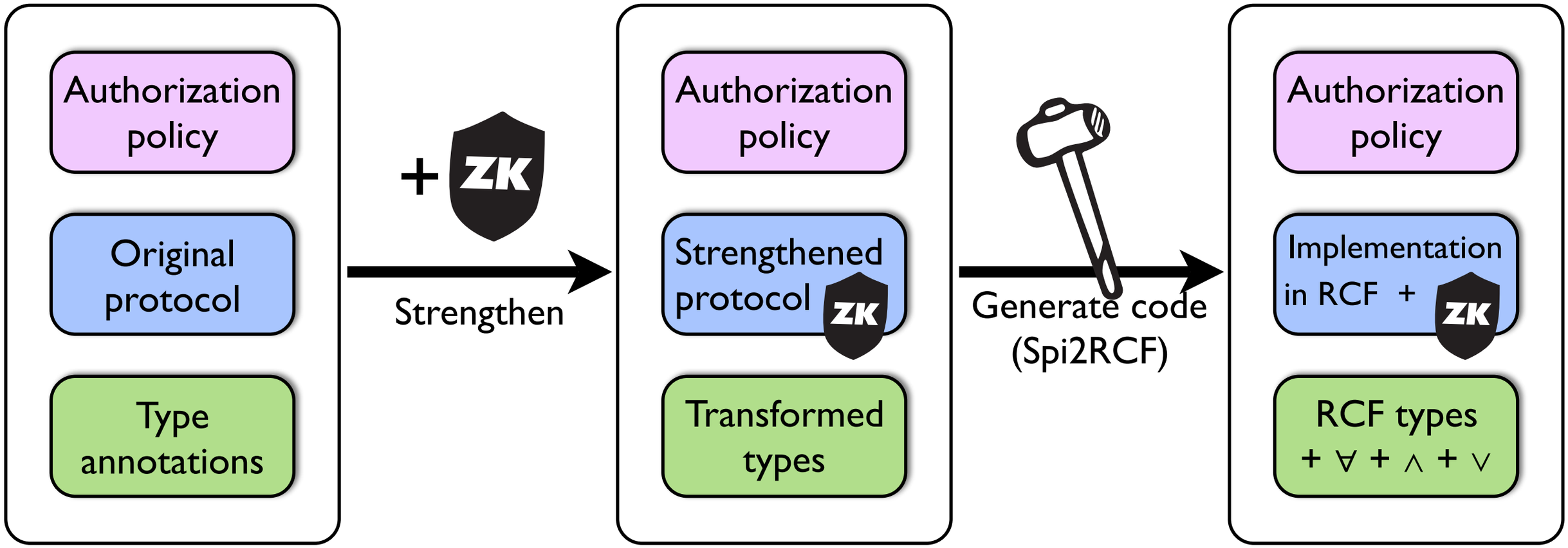
Automatic code generation

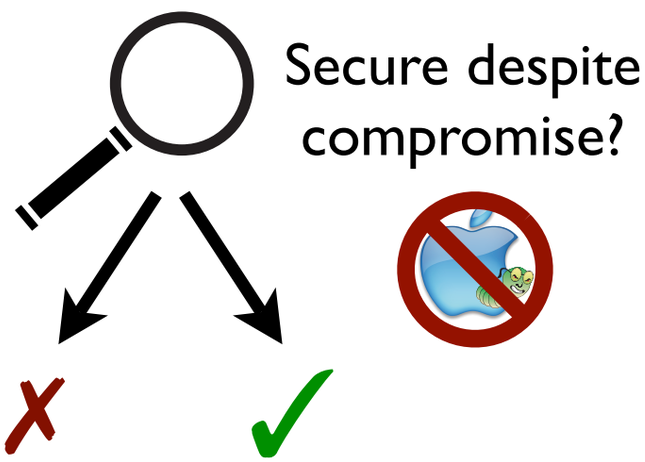


[Bengtson et. al., CSF '08]

[Backes, Hriţcu, Maffei & Tarrach, FCS '09 in August]

Automatic code generation

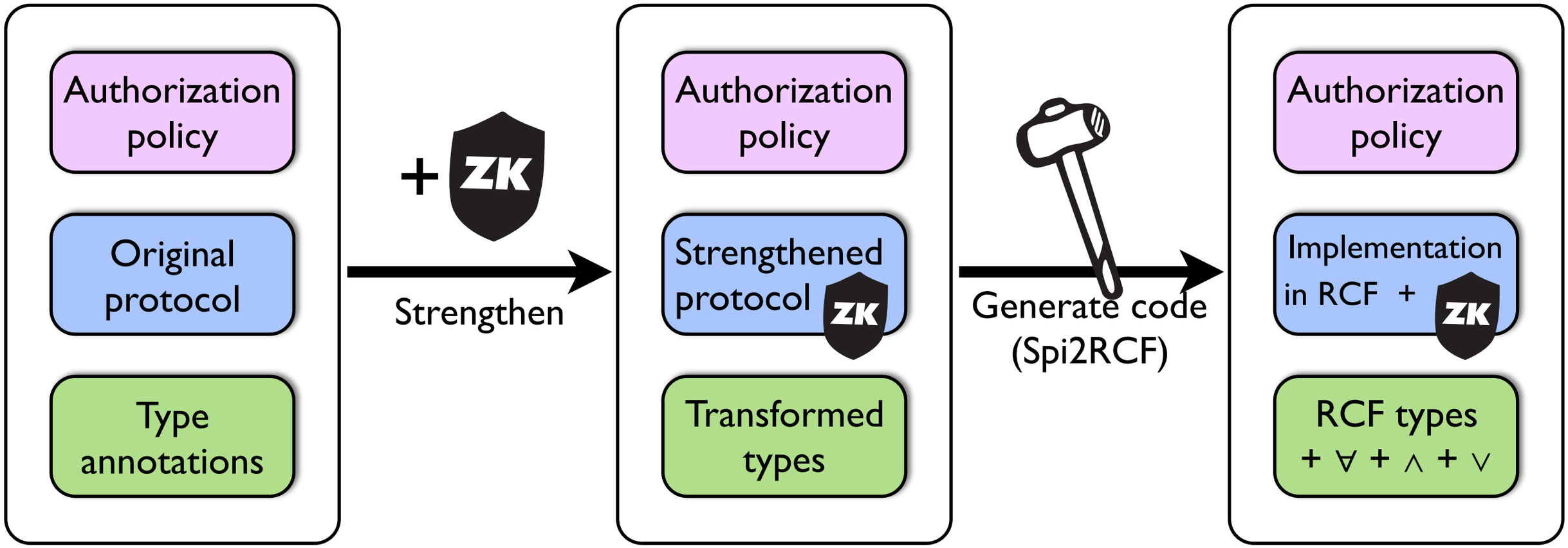


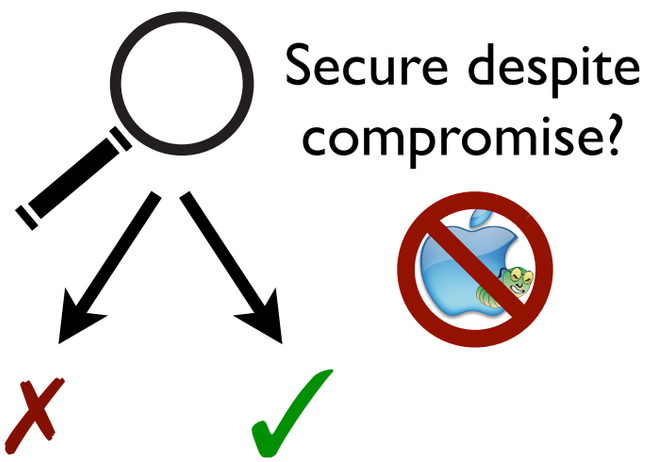


[Bengtson et. al., CSF '08]

[Backes, Hriţcu, Maffei & Tarrach, FCS '09 in August]

Automatic code generation





Recent work Matteo presented in the short talks session

[Bengtson et. al., CSF '08]

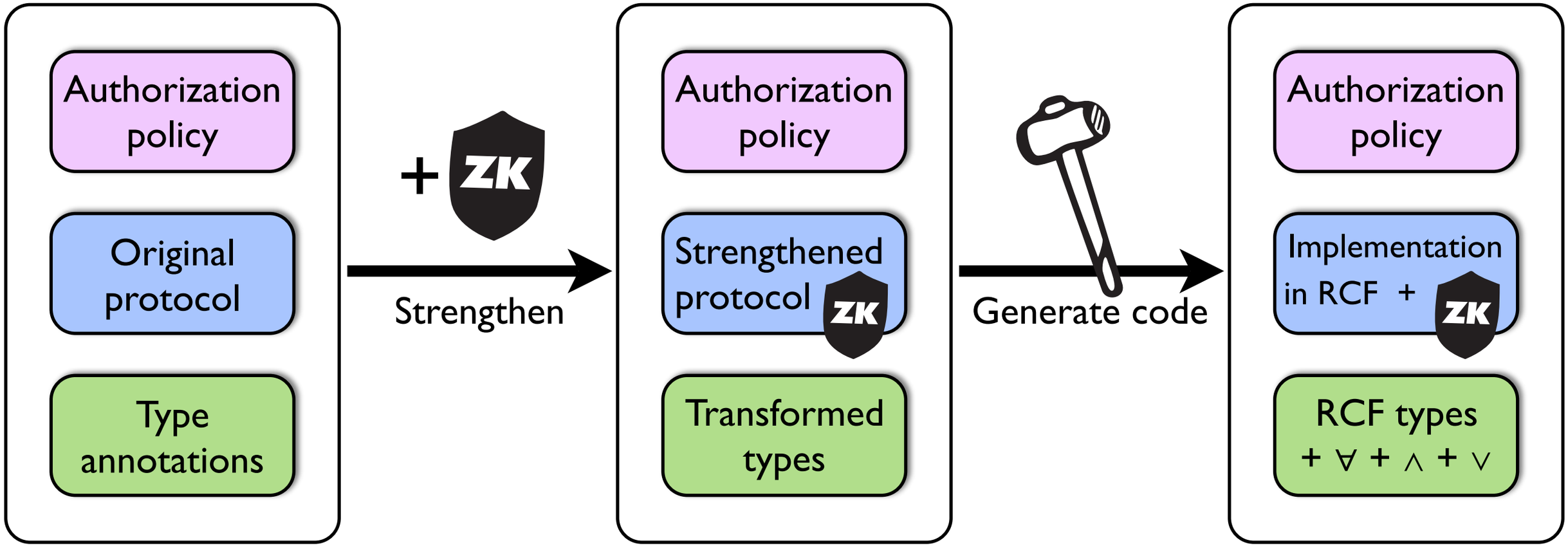
[Backes, Hriţcu, Maffei & Tarrach, FCS '09 in August]

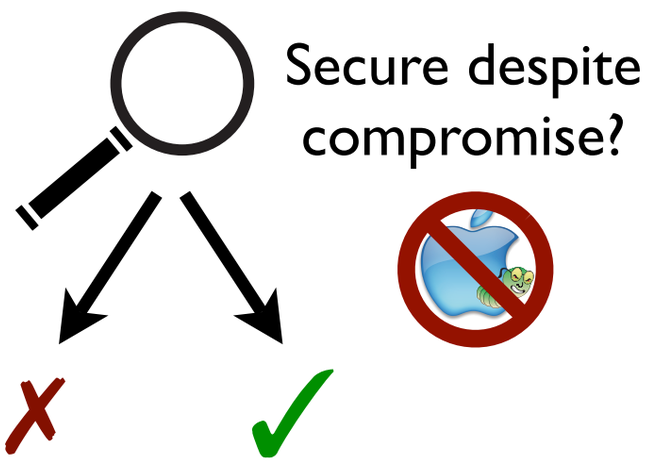
Future Work

- Apply transformation to more protocols (e.g. web services)
- Optimize transformation
  - leverage authorization policy and types
  - maybe also use ideas from work on multiparty sessions [Corin et. al, CSF '07 & CSF '09]
  - translation validation approach well-suited for this
- Automatically generate zero-knowledge proof system corresponding to abstract statement
  - concrete cryptographic implementation hard to do by hand
  - efficiency is a big challenge

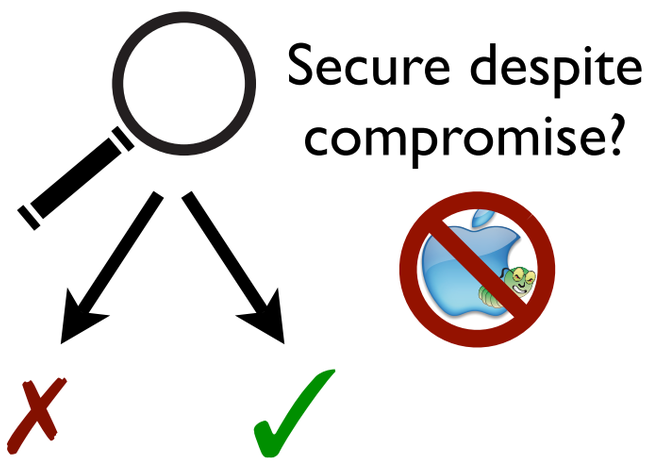
Thank you

The big picture



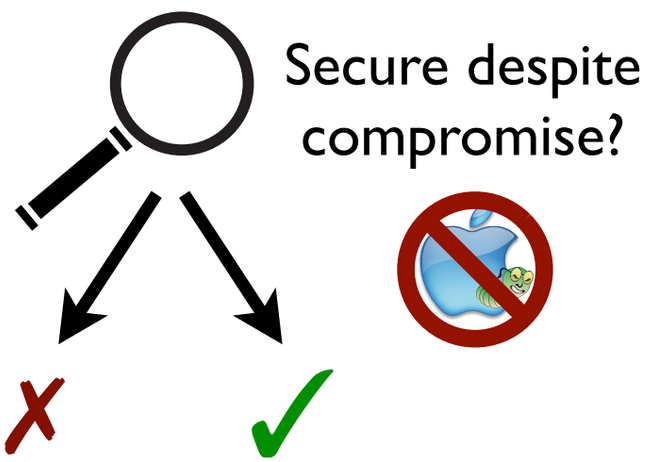


[Fournet, Gordon & Maffeis, CSF '07]



[Backes, Hriţcu & Maffei, CCS '08]

now with ∧ + ∨ + logical kinding



[Bengtson et. al., CSF '08]

[Backes, Hriţcu, Maffei & Tarrach, FCS '09]

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)

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
  - \+ guarantees about actual implementation of transformation
  - \+ changing transformation is very easy (e.g. optimizing)
  - − guarantees only for a specific policy (e.g. authorization policy)
  - − no guarantees if validation fails

Security properties (informal)

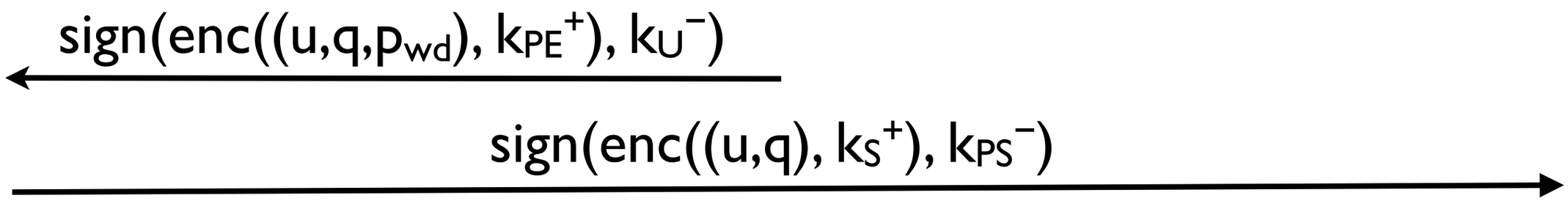
- **Robust safety:** in <u>all</u> executions all asserts succeed
  (i.e. asserts are logically entailed by the active assumes)
  - in the presence of <u>arbitrary DY attacker</u>
  - but where all participants are assumed honest

- **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]

Authorization policy



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 despite compromise

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

Compromising the proxy

proxy

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

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

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).

The transformed protocol is

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)

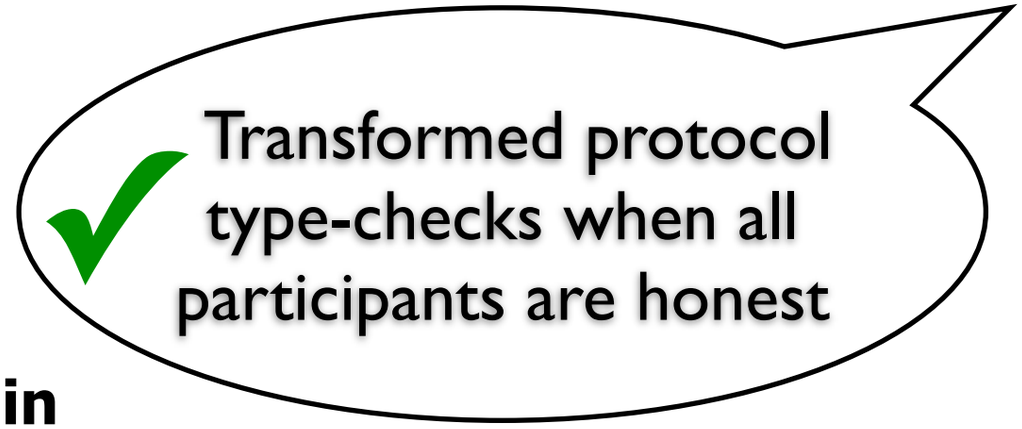
typedef T_1 = Triple(x_u : Un, {x_q : Un | Request(x_u, x_q)}, x_p : Private)
typedef T_2 = Pair(x_u : Un, {x_q : Un | Request(x_u, x_q) ∧ Registered(x_u)})

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** ∀ u, q. (Request(u, q) ∧ Registered(u) ⇒ 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) ∧ 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** ∀ u, q. (Request(u, q) ∧ Registered(u) ⇒ Authenticate(u, q)) ...

But these annotations are not appropriate when proxy is compromised

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

new k_{PS}^- : SigKey(PubEnc(T_2)) ;
new k_S^- : DecKey(T_2) ;

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})).

Type of keys does not help if proxy 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** ∀ 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)
...

$\exists \alpha_1,\alpha_2,\alpha_3.\text{check}(\beta_1,k_{PS}^{+}) = \text{enc}((\beta_3,\alpha_2), k_S^{+}) \wedge \text{dec}(\text{check}(\beta_2, k_U^{+}), \alpha_1) = (\beta_3,\alpha_2,\alpha_3)$
$\wedge$ 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;
(user | bad_proxy | store' | policy)

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

Compromising the user

user

(k_U^-, p_{wd})

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

proxy

store

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_2unlessP = {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_2unlessP));
new k_S^- : DecKey(T_2unlessP);
new p_{wd} : Private; (user | proxy | store | policy)

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

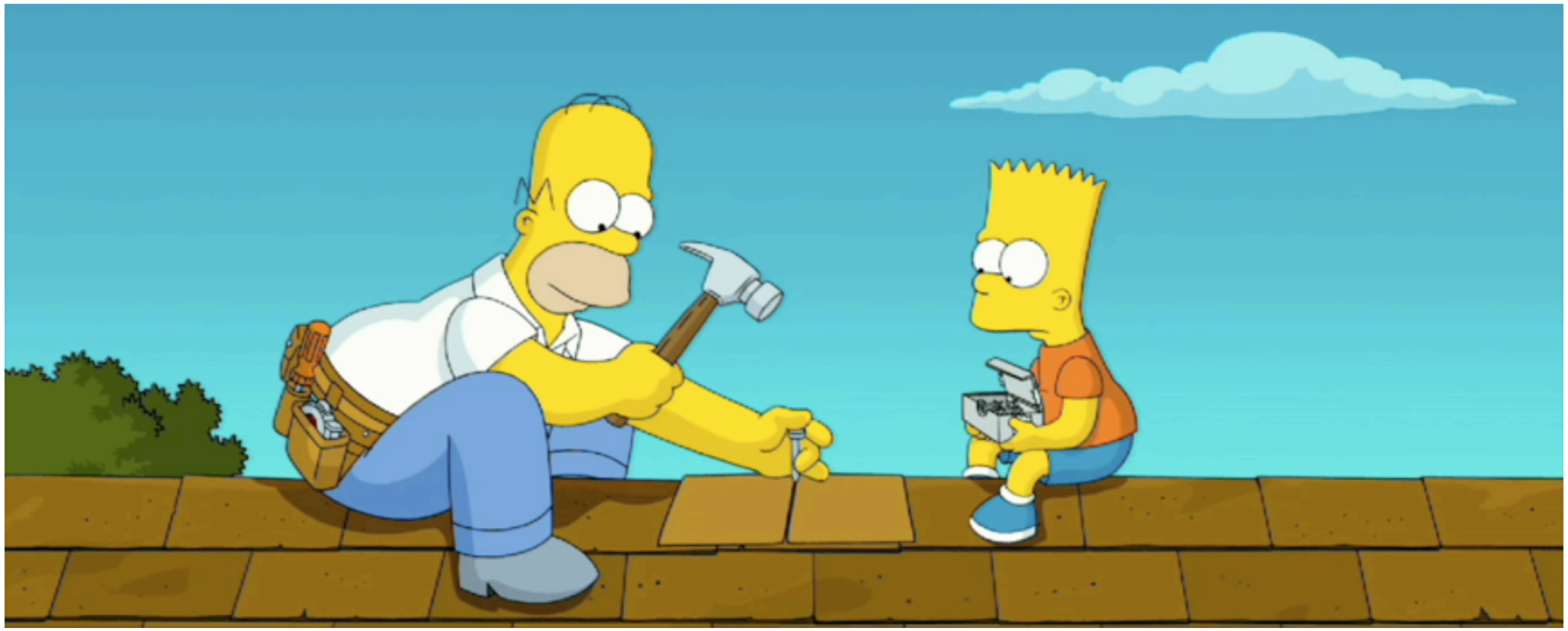
let proxy = **assume** Registered(u) |
in(c_1, x);
let (=u, x_q, =p_{wd}) = dec(check(x, k_U^+), k_{PE}^-) **in**
out(c_2, sign(enc((u,x_q), k_S^+), k_{PS}^-)).

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).

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/
- Spi2RCF release coming soon



Thank you