Logical Foundations of Secure Resource Management in Protocol Implementations

Matteo Maffei[1]
joint work with Michele Bugliesi[2], Fabienne Eigner[1] and Stefano Calzavara[2]

[1]CISPA, Saarland University

[2]Università Ca' Foscari Venezia

26 November 2014, Joint EasyCrypt-F*-CryptoVerif School, Paris

POST'13, best EATCS paper award at ETAPS'13.

Outline

- *Why?* (Beyond FOL refinements)
- *What?* (Affine logic for security protocols)
- *How?* (Proof techniques)

Refinement types for secure implementations

Verified implementations

- narrow the gap between formal model and implementation
- combine type-checking with general-purpose theorem proving
- efficient and modular verification

Refinement types for secure implementations

Verified implementations

- narrow the gap between formal model and implementation
- combine type-checking with general-purpose theorem proving
- efficient and modular verification

Methodology

- annotate the code with logical formulas
  - *assumptions*: formulas which are assumed to hold
  - *assertions*: formulas which must be entailed by the assumptions

Refinement types for secure implementations

Verified implementations

- narrow the gap between formal model and implementation
- combine type-checking with general-purpose theorem proving
- efficient and modular verification

Methodology

- annotate the code with logical formulas
  - *assumptions*: formulas which are assumed to hold
  - *assertions*: formulas which must be entailed by the assumptions
- type-check the code against appropriate refinement types

Refinement types for secure implementations

Verified implementations

- narrow the gap between formal model and implementation
- combine type-checking with general-purpose theorem proving
- efficient and modular verification

Methodology

- annotate the code with logical formulas
  - *assumptions*: formulas which are assumed to hold
  - *assertions*: formulas which must be entailed by the assumptions
- type-check the code against appropriate refinement types
- well-typed programs are robustly safe: assertions are always entailed by the introduced assumptions, even in presence of an opponent

A glance at refinement typing

C $\qquad\qquad\qquad\qquad$ S

$\textit{assume}\ \mathit{Good}(\mathit{msg})$

$$\xrightarrow{\hspace{3em}\mathrm{sign}(\mathit{msg},\mathrm{sk}(C))\hspace{3em}}$$

$\textit{assert}\ \mathit{Good}(\mathit{msg})$

A glance at refinement typing

C $\qquad\qquad\qquad\qquad$ S

assume Good(msg)

$$\xrightarrow{\mathrm{sign}(msg{:}\{x{:}string \mid Good(x)\},\mathrm{sk}(C))}$$

assert Good(msg)

A glance at refinement typing

C $\qquad$ S

$\textit{assume } \mathit{Good}(msg)$

$$\xrightarrow{\mathrm{sign}(msg:\{x:string \mid Good(x)\},\mathrm{sk}(C))}$$

$\textit{assert } \mathit{Good}(msg)$

Refinement typing

$$\mathrm{sk}(C) : SigKey(\{x : string \mid Good(x)\})$$

- C must prove that $Good(msg)$ holds true upon signing
- S can rely on $Good(msg)$ being true upon verification

Replays

C $\qquad\qquad$ O $\qquad\qquad$ S

$\mathit{assume}\ \ \mathit{Good}(\mathit{msg})$

$C \xrightarrow{\text{sign}(\mathit{msg},\text{sk}(C))} O$

$O \xrightarrow{\text{sign}(\mathit{msg},\text{sk}(C))} S$

$\mathit{assert}\ \ \mathit{Good}(\mathit{msg})$

$O \xrightarrow{\text{sign}(\mathit{msg},\text{sk}(C))} S$

$\mathit{assert}\ \ \mathit{Good}(\mathit{msg})$

This run is safe...

In FOL: $\mathit{Good}(\mathit{msg}) \vdash \mathit{Good}(\mathit{msg}) \wedge \mathit{Good}(\mathit{msg})$

...but sometimes it should not!

- What if $\mathit{Good}(\mathit{msg})$ expresses a bank transaction?

Outline

- *Why?* (Beyond FOL refinements)
- *What?* (Affine refinements for security protocols)
- *How?* (Proof techniques)

Resource-aware properties

Counting

Some properties require to count the number of times a certain resource is used (or an action is performed)

- Injective agreement or strong authentication: every end-event is preceded by a *distinct* begin-event
- No double-vote: cannot vote more than once

Resource-aware properties

Counting

Some properties require to count the number of times a certain resource is used (or an action is performed)

- Injective agreement or strong authentication: every end-event is preceded by a *distinct* begin-event
- No double-vote: cannot vote more than once

Affine (or resource-aware) logic

Such properties can be naturally expressed in *affine logic* (no contraction)

- affine hypotheses A can be used at most once
- exponential hypotheses !A can be used arbitrarily often

For instance, $\mathit{Good}(\mathit{msg}) \not\vdash \mathit{Good}(\mathit{msg}) \otimes \mathit{Good}(\mathit{msg})$
($\otimes$ denotes conjunction in affine logic)

Resource-aware properties

Counting

Some properties require to count the number of times a certain resource is used (or an action is performed)

- Injective agreement or strong authentication: every end-event is preceded by a *distinct* begin-event
- No double-vote: cannot vote more than once

Affine (or resource-aware) logic

Such properties can be naturally expressed in *affine logic* (no contraction)

- affine hypotheses A can be used at most once
- exponential hypotheses !A can be used arbitrarily often

For instance, $\mathit{Good}(\mathit{msg}) \nvdash \mathit{Good}(\mathit{msg}) \otimes \mathit{Good}(\mathit{msg})$
($\otimes$ denotes conjunction in affine logic)

How can we type-check cryptographic protocols that achieve resource-aware properties?

Resource-aware refinement typing

C $\qquad\qquad\qquad\qquad\qquad\qquad$ S

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $\mathit{assume}\ \mathit{Nonce}(n)$

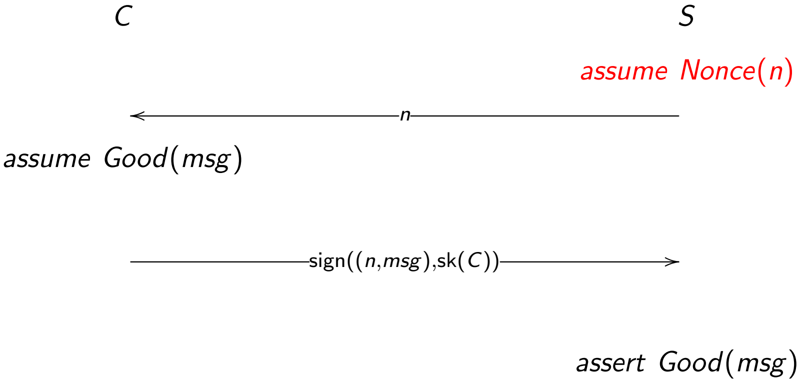
$$\xleftarrow{\quad\quad\quad\quad n \quad\quad\quad\quad}$$

$\mathit{assume}\ \mathit{Good}(\mathit{msg})$

$$\xrightarrow{\quad \mathrm{sign}((n,\mathit{msg}),\mathrm{sk}(C)) \quad}$$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $\mathit{assert}\ \mathit{Good}(\mathit{msg})$

Resource-aware refinement typing



Guarded refinement types

$\mathsf{sk}(C) : SigKey(x : int, \{y : string \mid\ !(Nonce(x) \multimap Good(y))\})$

Resource-aware refinement typing

C $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ S

$\mathit{assume}\ \mathit{Nonce}(n)$

$C \xleftarrow{\quad n \quad} S$

$\mathit{assume}\ \mathit{Good}(\mathit{msg})$

Prove: $!(\mathit{Nonce}(n) \multimap \mathit{Good}(\mathit{msg}))$

$C \xrightarrow{\text{sign}((n,\mathit{msg}),\text{sk}(C))} S$

Rely on: $!(\mathit{Nonce}(n) \multimap \mathit{Good}(\mathit{msg}))$

$\mathit{assert}\ \mathit{Good}(\mathit{msg})$

Guarded refinement types

$\text{sk}(C) : \mathit{SigKey}(x : \mathit{int}, \{y : \mathit{string} \mid !(\mathit{Nonce}(x) \multimap \mathit{Good}(y))\})$

Resource-aware refinement typing

C $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ S

$\mathit{assume}\ \ \mathit{Nonce}(n)$

$C \xleftarrow{\quad n \quad} S$

$\mathit{assume}\ \ \mathit{Good}(\mathit{msg})$

Prove: $!(\mathit{Nonce}(n) \multimap \mathit{Good}(\mathit{msg}))$

$C \xrightarrow{\ \mathrm{sign}((n,\mathit{msg}),\mathrm{sk}(C))\ } S$

Rely on: $!(\mathit{Nonce}(n) \multimap \mathit{Good}(\mathit{msg}))$

$\mathit{assert}\ \ \mathit{Good}(\mathit{msg})$

Type-checking the assertion

$\mathit{Nonce}(n), !(\mathit{Nonce}(n) \multimap \mathit{Good}(\mathit{msg})) \vdash \mathit{Good}(\mathit{msg})$

Resource-aware refinement typing

C $\qquad\qquad\qquad\qquad\qquad\qquad$ S

$assume\ \ Nonce(n)$

$\xleftarrow{\qquad\qquad n \qquad\qquad}$

$assume\ \ Good(msg)$

Prove: $!(Nonce(n) \multimap Good(msg))$

$\xrightarrow{\qquad \text{sign}((n, msg), \text{sk}(C)) \qquad}$

Rely on: $!(Nonce(n) \multimap Good(msg))$

$assert\ \ Good(msg)$

Preventing duplication

$Nonce(n), !(Nonce(n) \multimap Good(msg)) \nvdash Good(msg) \otimes Good(msg)$

Static enforcement of *resource-aware* policies

Contributions

- a theory of exponential serialization
- a type system for enforcing *affine logic* policies on application code

Static enforcement of *resource-aware* policies

Contributions

- a theory of exponential serialization
- a type system for enforcing *affine logic* policies on application code

The type system

- the theory is developed for RCF (a core calculus of F#)

Static enforcement of *resource-aware* policies

Contributions

- a theory of exponential serialization
- a type system for enforcing *affine logic* policies on application code

The type system

- the theory is developed for RCF (a core calculus of F#)
- the design leverages the theory of exponential serialization
  - no affine types: we can encode them
  - affine formulas: clearly separated from the typing information

Static enforcement of *resource-aware* policies

Contributions

- a theory of exponential serialization
- a type system for enforcing *affine logic* policies on application code

The type system

- the theory is developed for RCF (a core calculus of F#)
- the design leverages the theory of exponential serialization
  - no affine types: we can encode them
  - affine formulas: clearly separated from the typing information
  - this simplifies both the theory and the implementation

Static enforcement of *resource-aware* policies

Contributions

- a theory of exponential serialization
- a type system for enforcing *affine logic* policies on application code

The type system

- the theory is developed for RCF (a core calculus of F#)
- the design leverages the theory of exponential serialization
  - no affine types: we can encode them
  - affine formulas: clearly separated from the typing information
  - this simplifies both the theory and the implementation
- well-typed programs are *robustly safe*
  - the multiplicative conjunction of the assertions is entailed by the introduced assumptions

Static enforcement of *resource-aware* policies

Contributions

- a theory of exponential serialization
- a type system for enforcing *affine logic* policies on application code

The type system

- the theory is developed for RCF (a core calculus of F#)
- the design leverages the theory of exponential serialization
  - no affine types: we can encode them
  - affine formulas: clearly separated from the typing information
  - this simplifies both the theory and the implementation
- well-typed programs are *robustly safe*
  - the multiplicative conjunction of the assertions is entailed by the introduced assumptions
  - even in presence of an arbitrary opponent

Outline

- *Why?* (Beyond FOL refinements)
- *What?* (Affine refinements for security protocols)
- *How?* (Proof techniques)

Example: a streaming service

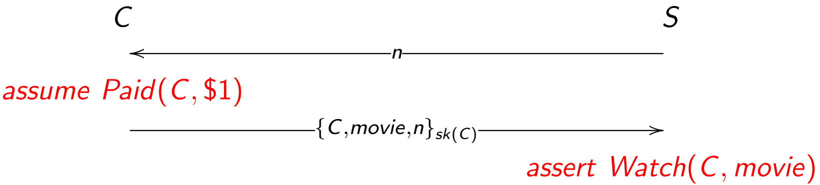
$$C \xleftarrow{\quad n \quad} S$$

$$C \xrightarrow{\quad \{C, movie, n\}_{sk(C)} \quad} S$$

Authorization policy

$$\mathcal{P} = \,!\forall x, y.(Paid(x, \$1) \multimap Watch(x, y))$$

Example: a streaming service



Authorization policy

$$\mathcal{P} = !\forall x, y.(Paid(x, \$1) \multimap Watch(x, y))$$

Implementing the streaming service

$$C \qquad\qquad\qquad\qquad\qquad\qquad S$$

$$C \xleftarrow{\quad n \quad} S$$

assume $Paid(C, \$1)$

$$C \xrightarrow{\quad \{C, movie, n\}_{sk(C)} \quad} S$$

assert $Watch(C, movie)$

Implementing the streaming service

C $\qquad\qquad\qquad\qquad\qquad\qquad$ S

$$C \xleftarrow{\quad n \quad} S$$

$assume\ Paid(C, \$1)$

$$C \xrightarrow{\quad \{C, movie, n\}_{sk(C)} \quad} S$$

$assert\ Watch(C, movie)$

```
let client C addC addS m sk =
    let xn = recv addC in assume Paid(C,$1);
    let msg = sign (C,m,xn) sk in send addS msg

val mkNonce: unit -> {x: bytes | Nonce(x)}

let serv S addC addS vk =
    let n = mkNonce () in send addC n;
    let msg = recv addS in
    let (xC, xm, xn) = verify msg vk in
    if (xn = n) then
        assert Watch(xC,xm)
```

Type-checking the server

Verification key type

$vk : VerKey(x : T_c, y : T_m, \{z : T_n \mid !(Nonce(z) \multimap Paid(x, \$1))\})$

Type-checking the server

Verification key type

$vk : VerKey(x : T_c, y : T_m, \{z : T_n \mid !(Nonce(z) \multimap Paid(x, \$1))\})$

```
val mkNonce: unit -> {x: bytes | Nonce(x)}

let serv S addC addS vk =
    let n = mkNonce () in send addC n;
    // Nonce(n) holds true
    let msg = recv addS in
    let (xC, xm, xn) = verify msg vk in
    // !(Nonce(xn) --o Paid(xC,$1)) holds true
    if (xn = n) then
        // !(xn = n) holds true
        assert Watch(xC,xm)
```

Type-checking the server

Verification key type

$$vk : VerKey(x : T_c, y : T_m, \{z : T_n \mid !(Nonce(z) \multimap Paid(x, \$1))\})$$

```
val mkNonce: unit -> {x: bytes | Nonce(x)}

let serv S addC addS vk =
    let n = mkNonce () in send addC n;
    // Nonce(n) holds true
    let msg = recv addS in
    let (xC, xm, xn) = verify msg vk in
    // !(Nonce(xn) --o Paid(xC,$1)) holds true
    if (xn = n) then
        // !(xn = n) holds true
        assert Watch(xC,xm)
```

Type-checking the assertion

Recall $\mathcal{P} = !\forall x, y.(Paid(x, \$1) \multimap Watch(x, y))$, we have:

$$\mathcal{P}, Nonce(n), !(xn = n), !(Nonce(xn) \multimap Paid(xC, \$1)) \vdash Watch(xC, xm)$$

Type-checking the client

Signing key type

$sk : SigKey(x : T_c, y : T_m, \{z : T_n \mid !(Nonce(z) \multimap Paid(x, \$1))\})$

Type-checking the client

Signing key type

$sk : SigKey(x : T_c, y : T_m, \{z : T_n \mid !(Nonce(z) \multimap Paid(x, \$1))\})$

```
let client C addC addS m sk =
    let xn = recv addC in assume Paid(C,$1);
    // Paid(C,$1) holds true
    let msg = sign (C,m,xn) sk in send addS msg
```

Type-checking the client

Signing key type

$sk : SigKey(x : T_c, y : T_m, \{z : T_n \mid !(Nonce(z) \multimap Paid(x, \$1))\})$

```
let client C addC addS m sk =
    let xn = recv addC in assume Paid(C,$1);
    // Paid(C,$1) holds true
    let msg = sign (C,m,xn) sk in send addS msg
```

Type-checking the signature

We must prove: $Paid(C, \$1) \vdash !(Nonce(xn) \multimap Paid(C, \$1))$

Type-checking the client

Signing key type

$sk : SigKey(x : T_c, y : T_m, \{z : T_n \mid !(Nonce(z) \multimap Paid(x, \$1))\})$

```
let client C addC addS m sk =
    let xn = recv addC in assume Paid(C,$1);
    // Paid(C,$1) holds true
    let msg = sign (C,m,xn) sk in send addS msg
```

Type-checking the signature

We must prove: $Paid(C, \$1) \vdash !(Nonce(xn) \multimap Paid(C, \$1))$

- ... but this is not true in affine logic!

Type-checking the client

Signing key type

$sk : SigKey(x : T_c, y : T_m, \{z : T_n \mid !(Nonce(z) \multimap Paid(x, \$1))\})$

```
let client C addC addS m sk =
    let xn = recv addC in assume Paid(C,$1);
    // Paid(C,$1) holds true
    let msg = sign (C,m,xn) sk in send addS msg
```

Type-checking the signature

We must prove: $Paid(C, \$1) \vdash !(Nonce(xn) \multimap Paid(C, \$1))$

- ... but this is not true in affine logic!
- This implies that the signing operation is not well-typed!

Exponential serialization (1/2)

Signing key type

$sk : SigKey(x : T_c, y : T_m, \{z : T_n \mid !(Nonce(z) \multimap Paid(x, \$1))\})$

```
let client C addC addS m sk =
    let xn = recv addC in assume Paid(C,$1);
    // Paid(C,$1) holds true
    let msg = sign (C,m,xn) sk in send addS msg
```

Exponential serialization (1/2)

Signing key type

$sk : SigKey(x : T_c, y : T_m, \{z : T_n \mid !(Nonce(z) \multimap Paid(x, \$1))\})$

```
let client C addC addS m sk =
    let xn = recv addC in assume Paid(C,$1);
    // Paid(C,$1) holds true
    let msg = sign (C,m,xn) sk in send addS msg
```

Solution

Add an explicit serializer among the assumptions

$$\mathcal{S} = !\forall x, y.(Paid(x, \$1) \multimap !(Nonce(y) \multimap Paid(x, \$1)))$$

Exponential serialization (1/2)

Signing key type

$sk : SigKey(x : T_c, y : T_m, \{z : T_n \mid !(Nonce(z) \multimap Paid(x, \$1))\})$

```
let client C addC addS m sk =
    let xn = recv addC in assume Paid(C,$1);
    // Paid(C,$1) holds true
    let msg = sign (C,m,xn) sk in send addS msg
```

Solution

Add an explicit serializer among the assumptions

$$\mathcal{S} = !\forall x, y.(Paid(x, \$1) \multimap !(Nonce(y) \multimap Paid(x, \$1)))$$

Notice that we have:

$$\mathcal{S}, Paid(C, \$1) \vdash !(Nonce(xn) \multimap Paid(C, \$1))$$

Exponential serialization (2/2)

Benefits

- modularity: let affine logic handle resource management!

Exponential serialization (2/2)

Benefits

- modularity: let affine logic handle resource management!
- expressiveness: nonces, TIDs, timestamps, session keys, affine types

Exponential serialization (2/2)

Benefits

- modularity: let affine logic handle resource management!
- expressiveness: nonces, TIDs, timestamps, session keys, affine types
- insight: a general logical encoding for distributed authorization

Exponential serialization (2/2)

Benefits

- modularity: let affine logic handle resource management!
- expressiveness: nonces, TIDs, timestamps, session keys, affine types
- insight: a general logical encoding for distributed authorization

Soundness

Adding serializers may weaken the authorization policy

Exponential serialization (2/2)

Benefits

- modularity: let affine logic handle resource management!
- expressiveness: nonces, TIDs, timestamps, session keys, affine types
- insight: a general logical encoding for distributed authorization

Soundness

Adding serializers may weaken the authorization policy

$$\mathcal{S} = !\forall x, y.(Paid(x, \$1) \multimap !(Nonce(y) \multimap Paid(x, \$1)))$$

$$Paid(C, \$1), Nonce(n), Nonce(n) \nvdash Paid(C, \$1) \otimes Paid(C, \$1)$$
$$Paid(C, \$1), Nonce(n), Nonce(n), \mathcal{S} \vdash Paid(C, \$1) \otimes Paid(C, \$1)$$

Exponential serialization (2/2)

Benefits

- modularity: let affine logic handle resource management!
- expressiveness: nonces, TIDs, timestamps, session keys, affine types
- insight: a general logical encoding for distributed authorization

Soundness

Adding serializers may weaken the authorization policy

$$\mathcal{S} = !\forall x, y.(\mathit{Paid}(x, \$1) \multimap !(\mathit{Nonce}(y) \multimap \mathit{Paid}(x, \$1)))$$

$$\mathit{Paid}(C, \$1), \mathit{Nonce}(n), \mathit{Nonce}(n) \nvdash \mathit{Paid}(C, \$1) \otimes \mathit{Paid}(C, \$1)$$
$$\mathit{Paid}(C, \$1), \mathit{Nonce}(n), \mathit{Nonce}(n), \mathcal{S} \vdash \mathit{Paid}(C, \$1) \otimes \mathit{Paid}(C, \$1)$$

In the paper we identify sufficient syntactic conditions for soundness

Overview of the type system

Typing environments

Type judgements of the form $\Gamma; \Delta \vdash \mathcal{J}$

- Γ is a list of type bindings
- Δ is a multiset of affine formulas

Overview of the type system

Typing environments

Type judgements of the form $\Gamma; \Delta \vdash \mathcal{J}$

- Γ is a list of type bindings
- Δ is a multiset of affine formulas

Typing rules

General structure of a typing rule

$$\frac{\Gamma; \Delta_1 \vdash \mathcal{J}_1 \qquad \ldots \qquad \Gamma; \Delta_n \vdash \mathcal{J}_n \qquad \Delta \hookrightarrow \Delta_1, \ldots, \Delta_n}{\Gamma; \Delta \vdash \mathcal{J}}$$

The rewriting $\Delta \hookrightarrow \Delta'$ allows for manipulating the logical context according to the entailment relation (e.g., split conjunctions or duplicate exponential resources)

Example

Standard Refinement Typing

$$\textsc{Val Refine}\quad \frac{\Gamma; \Delta \vdash M : T \qquad \Gamma; \Delta \vdash F\{M/x\}}{\Gamma; \Delta \vdash M : \{x : T \mid F\}}$$

Example

Standard Refinement Typing

$$\dfrac{\Gamma; \Delta \vdash M : T \qquad \Gamma; \Delta \vdash F\{M/x\}}{\Gamma; \Delta \vdash M : \{x : T \mid F\}}\ \textsc{Val Refine}$$

Affine Refinement Typing

We have to choose where to use affine resources:

$$\dfrac{\Gamma; \Delta_1 \vdash M : T \qquad \Gamma; \Delta_2 \vdash F\{M/x\} \qquad \Delta \hookrightarrow \Delta_1, \Delta_2}{\Gamma; \Delta \vdash M : \{x : T \mid F\}}\ \textsc{Val Refine}$$

E.g., $M : T; A \otimes B \vdash M : \{y : \{x : T \mid A\} \mid B\}$ since $A \otimes B \hookrightarrow A, B$

Algorithmic type-checking

Environment rewriting is highly non-deterministic and this hinders the implementation of a type-checker:

- the type-checker should incorporate logical entailment...
- ... and distribute formulas among subderivations

Algorithmic type-checking

Environment rewriting is highly non-deterministic and this hinders the implementation of a type-checker:

- the type-checker should incorporate logical entailment...
- ... and distribute formulas among subderivations

Algorithmic formulation

$\Gamma; \Delta \vdash \mathcal{J}$ replaced by $\Gamma \vdash_{\mathsf{alg}} \mathcal{J}; F$

- we remove the non-determinism associated to Δ
- deterministic, syntax-directed construction of a proof obligation F

Algorithmic type-checking

Environment rewriting is highly non-deterministic and this hinders the implementation of a type-checker:

- the type-checker should incorporate logical entailment...
- ... and distribute formulas among subderivations

Algorithmic formulation

$\Gamma; \Delta \vdash \mathcal{J}$ replaced by $\Gamma \vdash_{\mathsf{alg}} \mathcal{J}; F$

- we remove the non-determinism associated to Δ
- deterministic, syntax-directed construction of a proof obligation F
- F is later discharged by an external theorem prover
- if both steps succeed, the code is well-typed

Algorithmic type-checking

Environment rewriting is highly non-deterministic and this hinders the implementation of a type-checker:

- the type-checker should incorporate logical entailment...
- ... and distribute formulas among subderivations

Algorithmic formulation

$\Gamma; \Delta \vdash \mathcal{J}$ replaced by $\Gamma \vdash_{\text{alg}} \mathcal{J}; F$

- we remove the non-determinism associated to Δ
- deterministic, syntax-directed construction of a proof obligation F
- F is later discharged by an external theorem prover
- if both steps succeed, the code is well-typed
- algorithmic type-checking is sound and complete

Example

Affine Refinement Typing

We have to choose where to use affine resources:

$$\textsc{Val Refine}\;\frac{\Gamma; \Delta_1 \vdash M : T \qquad \Gamma; \Delta_2 \vdash F\{M/x\} \qquad \Delta \hookrightarrow \Delta_1, \Delta_2}{\Gamma; \Delta \vdash M : \{x : T \mid F\}}$$

Example

Affine Refinement Typing

We have to choose where to use affine resources:

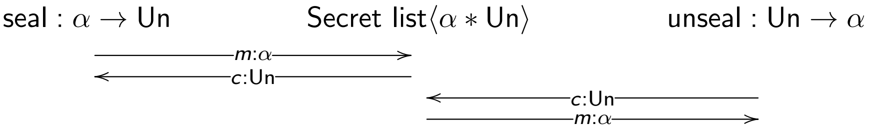
$$\textsc{Val Refine}\quad \frac{\Gamma; \Delta_1 \vdash M : T \qquad \Gamma; \Delta_2 \vdash F\{M/x\} \qquad \Delta \hookrightarrow \Delta_1, \Delta_2}{\Gamma; \Delta \vdash M : \{x : T \mid F\}}$$

Affine Refinement Typing

- Values are partially annotated to make type-checking syntax-directed
- The algorithmic typing rules collect the formulas required for typing, which are then automatically discharged:

$$\textsc{Val Refine}\quad \frac{\Gamma \vdash_{\mathsf{alg}} M : T; F'}{\Gamma \vdash_{\mathsf{alg}} M_{\{x:\_|F\}} : \{x : T \mid F\}; F' \otimes F\{M/x\}}$$

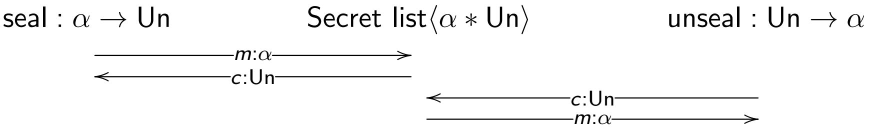
Typing cryptographic libraries



Symbolic cryptography

We prove properties in the symbolic setting, using standard *sealing-based* cryptographic libraries developed for F7/F*

Typing cryptographic libraries



Symbolic cryptography

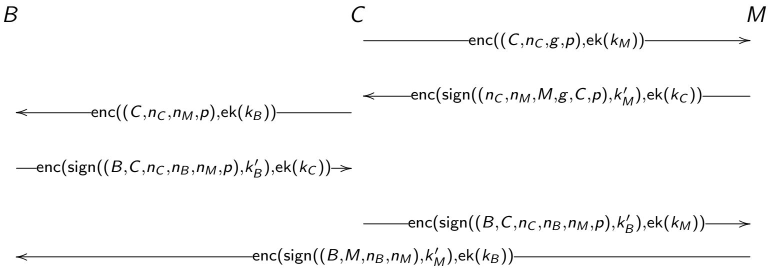
We prove properties in the symbolic setting, using standard *sealing-based* cryptographic libraries developed for F7/F*

Key aspects

- Communication and cryptographic libraries build on *exponential types*, which do not carry any affine refinements (they are all serialized)
- Consequently, we can just reuse standard typed cryptographic libraries

Case study: EPMO (1/2)

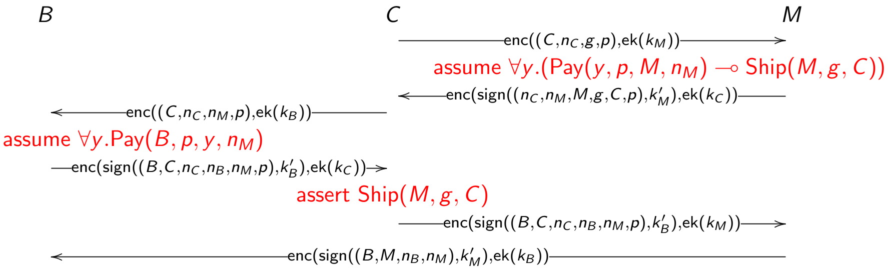
An e-commerce protocol proposed by Guttman et al.[1]



[1]We consider here a simpler (safe) variant of the protocol

Case study: EPMO (1/2)

An e-commerce protocol proposed by Guttman et al.[1]



[1] We consider here a simpler (safe) variant of the protocol

Case study: EPMO (1/2)

An e-commerce protocol proposed by Guttman et al.[1]

B $\qquad\qquad\qquad$ C $\qquad\qquad\qquad$ M

$C \longrightarrow M$: $\mathrm{enc}((C, n_C, g, p), \mathrm{ek}(k_M))$

assume $\forall y.(\mathrm{Pay}(y, p, M, n_M) \multimap \mathrm{Ship}(M, g, C))$

$M \longrightarrow C$: $\mathrm{enc}(\mathrm{sign}((n_C, n_M, M, g, C, p), k'_M), \mathrm{ek}(k_C))$

$C \longrightarrow B$: $\mathrm{enc}((C, n_C, n_M, p), \mathrm{ek}(k_B))$

assume $\forall y.\mathrm{Pay}(B, p, y, n_M)$

$B \longrightarrow C$: $\mathrm{enc}(\mathrm{sign}((B, C, n_C, n_B, n_M, p), k'_B), \mathrm{ek}(k_C))$

assert $\mathrm{Ship}(M, g, C)$

$C \longrightarrow M$: $\mathrm{enc}(\mathrm{sign}((B, C, n_C, n_B, n_M, p), k'_B), \mathrm{ek}(k_M))$

$M \longrightarrow B$: $\mathrm{enc}(\mathrm{sign}((B, M, n_B, n_M), k'_M), \mathrm{ek}(k_B))$

Challenges

1. the nonce n_C is checked twice by C (steps 2 and 4)
2. the verification key types must convey structured formulas

[1] We consider here a simpler (safe) variant of the protocol

Case study: EPMO (2/2)

Challenges

1. the nonce n_C is checked twice by C (steps 2 and 4)

Case study: EPMO (2/2)

Challenges

1. the nonce n_C is checked twice by C (steps 2 and 4)
   - we cannot construe n_C as an *affine* value

Case study: EPMO (2/2)

Challenges

1. the nonce n_C is checked twice by C (steps 2 and 4)
   - we cannot construe n_C as an *affine* value
   - solution: assume two predicates $Nonce_1(n_C)$ and $Nonce_2(n_C)$

Case study: EPMO (2/2)

Challenges

1. the nonce n_C is checked twice by C (steps 2 and 4)
   - we cannot construe n_C as an *affine* value
   - solution: assume two predicates $Nonce_1(n_C)$ and $Nonce_2(n_C)$
2. the verification key types must convey structured formulas

Case study: EPMO (2/2)

Challenges

1. the nonce n_C is checked twice by C (steps 2 and 4)
   - we cannot construe n_C as an *affine* value
   - solution: assume two predicates $Nonce_1(n_C)$ and $Nonce_2(n_C)$
2. the verification key types must convey structured formulas
   - if we had only affine values, we should look for an *encoding* (cf. F^*)

Case study: EPMO (2/2)

Challenges

1. the nonce n_C is checked twice by C (steps 2 and 4)
   - we cannot construe n_C as an *affine* value
   - solution: assume two predicates $Nonce_1(n_C)$ and $Nonce_2(n_C)$
2. the verification key types must convey structured formulas
   - if we had only affine values, we should look for an *encoding* (cf. F^*)
   - but this is not a problem for our framework

Case study: EPMO (2/2)

Challenges

1. the nonce n_C is checked twice by C (steps 2 and 4)
   - we cannot construe n_C as an *affine* value
   - solution: assume two predicates $Nonce_1(n_C)$ and $Nonce_2(n_C)$
2. the verification key types must convey structured formulas
   - if we had only affine values, we should look for an *encoding* (cf. F^*)
   - but this is not a problem for our framework

Verification

1. we enrich the code with suitable (refinement) types
2. we introduce the necessary serializers
3. we type-check the protocol

The proof obligation is dispatched by `llprover` in less than 20 ms (nevertheless, automated theorem provers for affine logic are far from being optimal)

Take-home message

Affine logic for security

- Affine logic elegantly captures security properties where the number of actions matters (e.g., injective agreement)
- Affine security properties for distributed systems can be statically enforced, modularly and efficiently

Take-home message

Affine logic for security

- Affine logic elegantly captures security properties where the number of actions matters (e.g., injective agreement)
- Affine security properties for distributed systems can be statically enforced, modularly and efficiently

Under the hood...

- We investigated a link between affine logic connectives and standard cryptographic patterns for authorization

Take-home message

Affine logic for security

- Affine logic elegantly captures security properties where the number of actions matters (e.g., injective agreement)
- Affine security properties for distributed systems can be statically enforced, modularly and efficiently

Under the hood...

- We investigated a link between affine logic connectives and standard cryptographic patterns for authorization
- We designed a modular refinement type system for enforcing *resource-aware* authorization policies in protocol implementations

Take-home message

Affine logic for security

- Affine logic elegantly captures security properties where the number of actions matters (e.g., injective agreement)
- Affine security properties for distributed systems can be statically enforced, modularly and efficiently

Under the hood...

- We investigated a link between affine logic connectives and standard cryptographic patterns for authorization
- We designed a modular refinement type system for enforcing *resource-aware* authorization policies in protocol implementations
- We devised a sound and complete algorithmic variant

Take-home message

Affine logic for security

- Affine logic elegantly captures security properties where the number of actions matters (e.g., injective agreement)
- Affine security properties for distributed systems can be statically enforced, modularly and efficiently

Under the hood...

- We investigated a link between affine logic connectives and standard cryptographic patterns for authorization
- We designed a modular refinement type system for enforcing *resource-aware* authorization policies in protocol implementations
- We devised a sound and complete algorithmic variant
- We showed the expressiveness of our framework by type-checking the implementation of
  - a variant of the EPMO protocol
  - Kerberos

Thank you for your attention!