

Type-checking Zero-knowledge

Cătălin Hrițcu

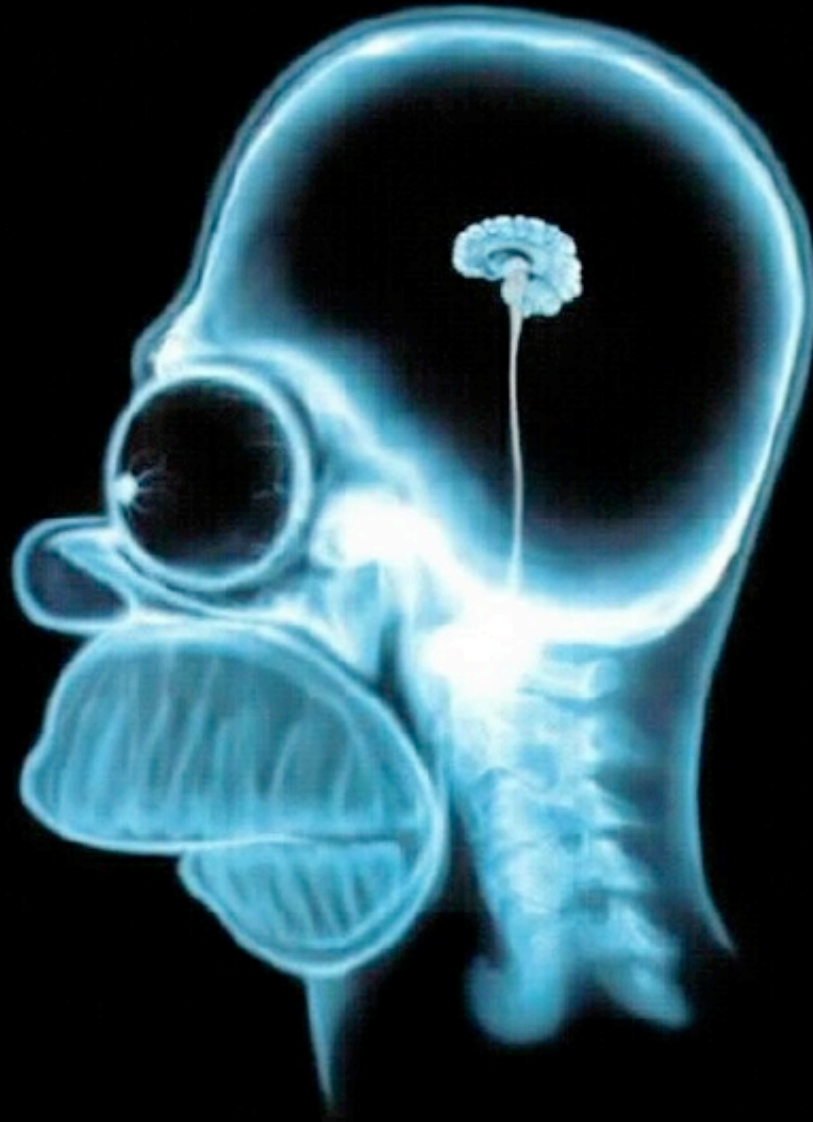
Saarland University, Saarbrücken, Germany

Joint work with: Michael Backes and Matteo Maffei

CCS 2008, Alexandria - Virginia, October 2008

The title of this talk is “Type-checking Zero-knowledge” ...

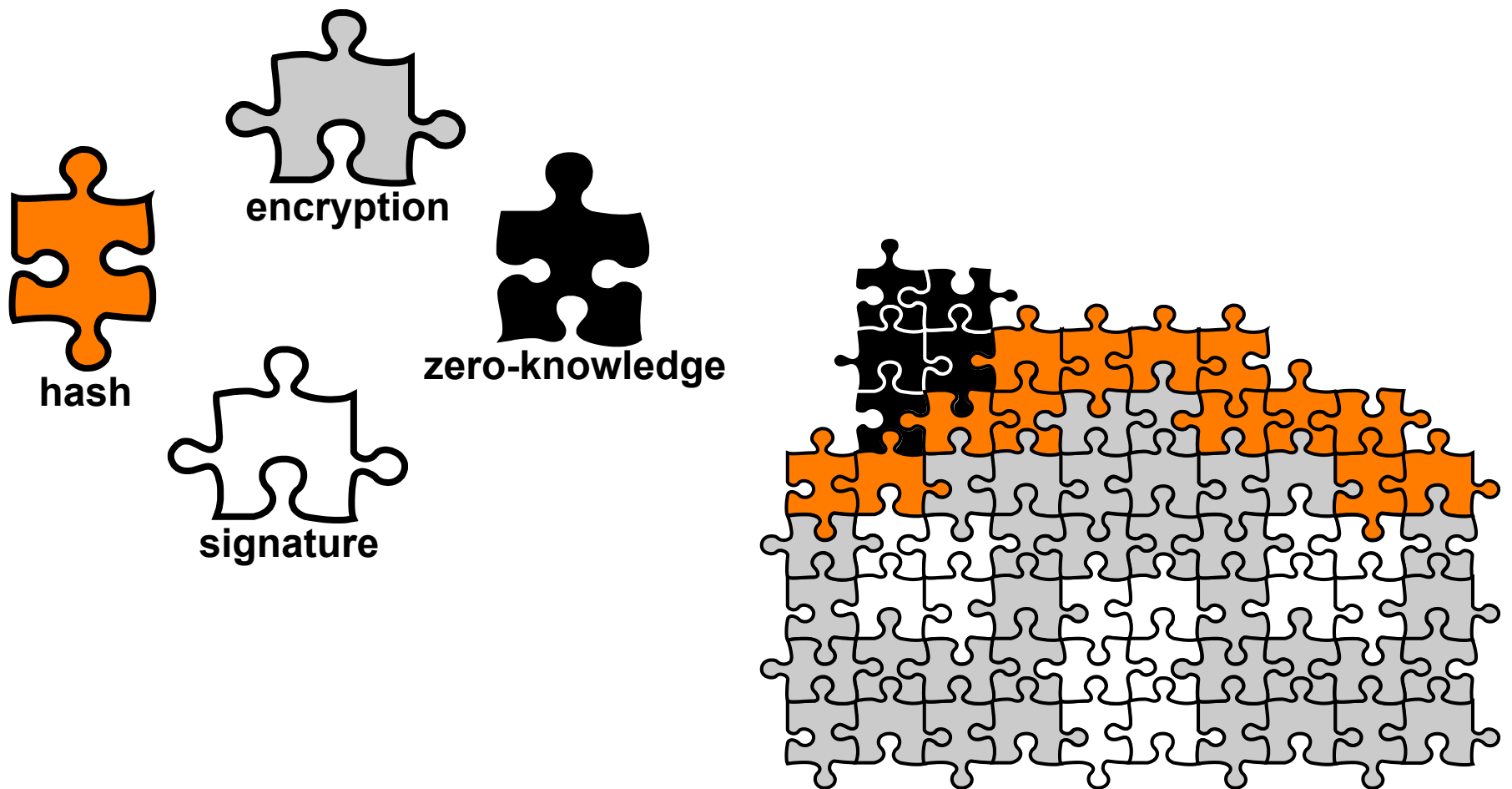
Zero-knowledge



However, I'm not going to talk about cryptography, or how zero-knowledge proofs are implemented. For me that's black magic anyway.

Security Protocols

designer's toolbox



Instead, I'm going to take a more abstract view **[click]**, and investigate how zero-knowledge proofs can be used as building blocks for designing large security protocols. When designing security protocols it is often convenient to look at the different cryptographic primitives as fully reliable black boxes, **[click]** that can be put together like the pieces of a puzzle in order to achieve the desired security goals.

Zero-knowledge Proofs



▶ Powerful primitives

- Prove the validity of a statement without revealing anything else
 - For instance, prove to know an object with certain properties without revealing this witness

▶ Early constructions general, but terribly inefficient

- Very limited practical impact

▶ More recent research provided

- Efficient constructions for special classes of statements
- Constructions for non-interactive zero-knowledge

– The zero-knowledge proofs are very powerful and exciting cryptographic primitives. However, for the purpose of this talk, all you need to know about them is that they allow a prover to show the validity of a statement without revealing any other information than that the statement is true. For instance, one can prove the knowledge of an object with certain properties without revealing this witness.

– **[click]** The early constructions for zero-knowledge were very general, but also very inefficient, so they had very limited practical impact.

– **[click]** However, the more recent research on zero-knowledge provided efficient constructions for special classes of statements, as well as constructions for non-interactive zero-knowledge proofs



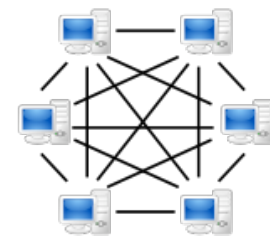
e-voting (Civitas)

Applications that use ZK

unique security features of ZK allows designing protocols fulfilling seemingly *conflicting requirements*



“trusted” computing (DAA)



p2p (PseudoTrust)

As a consequence, **many emerging application areas use zero-knowledge proofs**. I’m going to focus on 3 such areas: electronic voting, trusted computing and peer-to-peer protocols. In all these areas the unique security features of zero-knowledge proofs **allow the design of protocols that fulfill seemingly conflicting requirements**

– For instance when designing **electronic voting** protocols we need to preserve the **privacy** of the voters (so that nobody is able to find out how any of the voters voted -- in order to prevent vote buying or coercion), but at the same time we want the whole election process to be **verifiable** (we want to allow anyone to independently double-check the correctness of the elections).

– In certain **trusted computing** protocols we want users to **remotely attest** that their platform is certified and has not been tampered, while preserving the **anonymity** of the users.

– Similarly, in the case of certain **peer-to-peer protocols** we want users to be able to exchange information in a **trusted way**, without revealing the identity of the users.

So zero-knowledge proofs are the **critical building block** for many such useful protocols.



privacy

+



verifiability



e-voting (Civitas)



anonymity

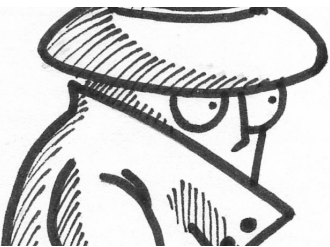
+



remote attestation



"trusted" computing (DAA)

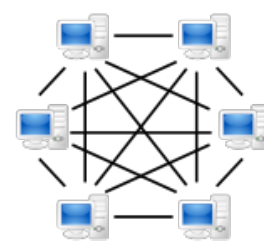


pseudonymity

+



trust



p2p (PseudoTrust)

As a consequence, many emerging application areas use **zero-knowledge proofs**. I'm going to focus on 3 such areas: electronic voting, trusted computing and peer-to-peer protocols. In all these areas the unique security features of zero-knowledge proofs **allow the design of protocols that fulfill seemingly conflicting requirements**

- For instance when designing **electronic voting** protocols we need to preserve the **privacy** of the voters (so that nobody is able to find out how any of the voters voted -- in order to prevent vote buying or coercion), but at the same time we want the whole election process to be **verifiable** (we want to allow anyone to independently double-check the correctness of the elections).

- In certain **trusted computing** protocols we want users to **remotely attest** that their platform is certified and has not been tampered, while preserving the **anonymity** of the users.

- Similarly, in the case of certain **peer-to-peer** protocols we want users to be able to exchange information in a **trusted** way, without revealing the identity of the users.

So zero-knowledge proofs are the **critical building block** for many such useful protocols.

Verifying Protocols Using ZK

- ▶ No verification tool for protocols using ZK
- ▶ Security protocols are hard to get right
- ▶ Automated verification can prevent errors
- ▶ Our goal
 - *To automatically analyze protocols using ZK in an efficient and scalable way*
- ▶ We built first *type system for zero-knowledge*

However, when we started this, there was **no automated verification tool** for protocols using ZK as primitive

- In general, security protocols are **hard to get right**, and **automated verification can really help** protocol designers prevent high-level errors
- So our goal was: **to automatically analyze protocols using zero-knowledge proofs in an efficient and scalable way**
- In order to achieve this, we built the **first type system for zero-knowledge**

Type System

- ▶ Type-checking fully automated and efficient
- ▶ Compositional therefore scalable

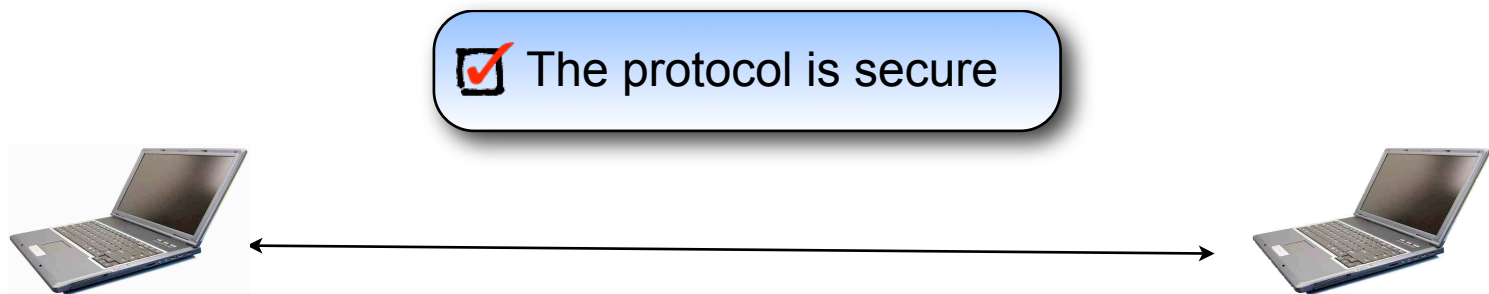
This code is safe

This code is safe



Type System

- ▶ Type-checking fully automated and efficient
- ▶ Compositional therefore scalable



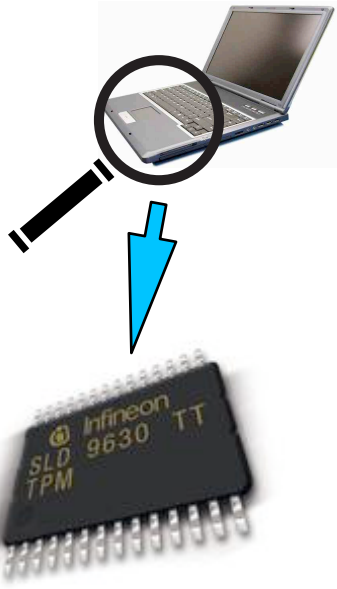
- ▶ Predictable termination behavior
- ▶ User needs to provide annotations (no free lunch)
 - ▶ in certain cases these can be automatically inferred

Type-checking DAA

(Direct Anonymous Attestation)

In the remainder of this talk, I'm going to show how our type system works on an example; and the example I chose is called the Direct Anonymous Attestation protocol, or DAA.

DAA (Direct Anonymous Attestation)



The user wants to authenticate a message m by proving that her platform has a valid TPM inside (*attestation*) ...

... but no other party should learn *which* TPM is used to authenticate m (*anonymity*)

[Brickell, Camenisch & Chen, CCS 2004]

Direct Anonymous Attestation (DAA)



f_{tpm}
(secret TPM identifier)

Issuer



!!! Mention: secret TPM identifier is also called “f value”

Direct Anonymous Attestation (DAA)

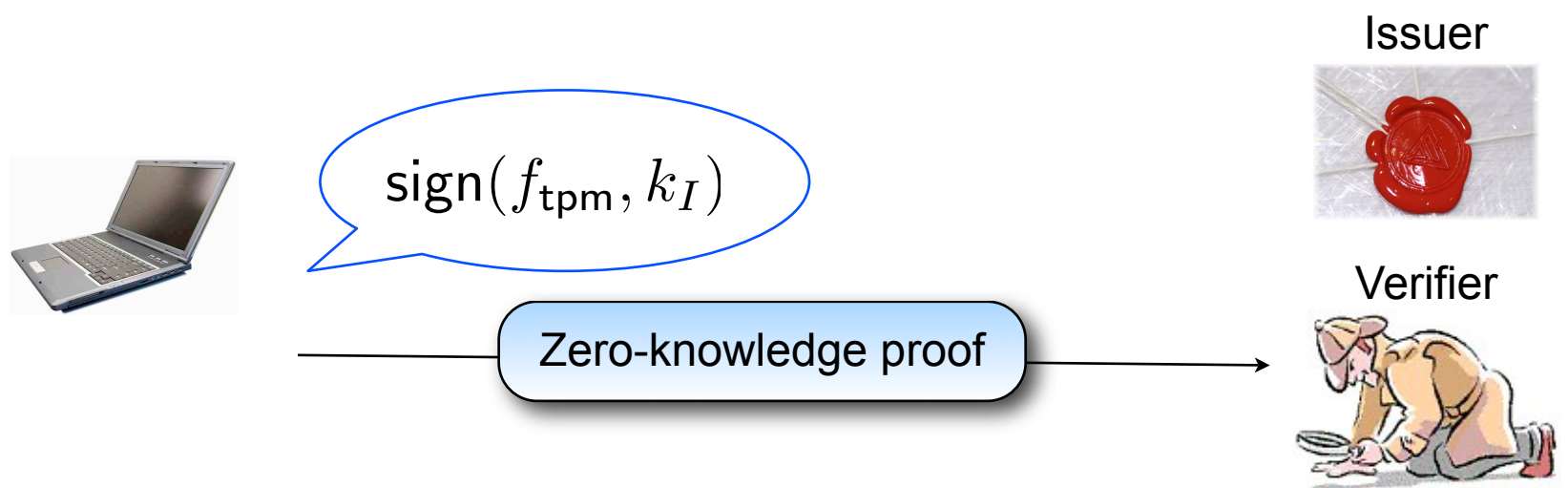


Joining Protocol

The user receives a certificate of f_{tpm} from the issuer

!!! Mention: secret TPM identifier is also called “f value”

Direct Anonymous Attestation (DAA)

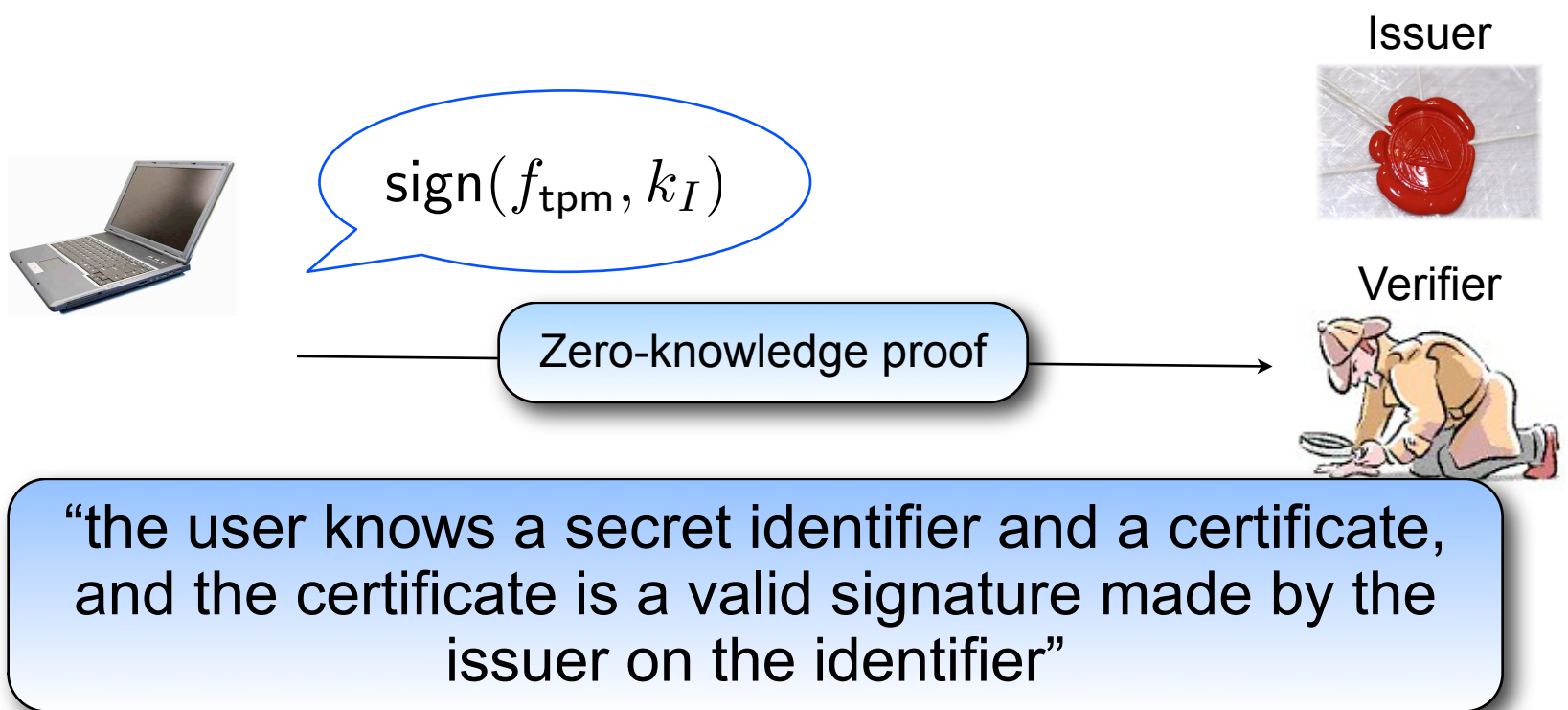


Signing Protocol

The user proves the knowledge of a certificate for his secret TPM identifier f_{tpm} ... without revealing it!

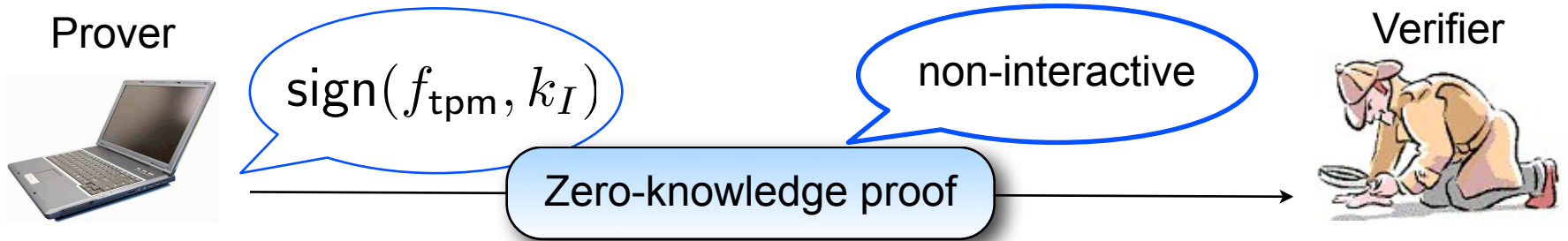
!!! Mention: secret TPM identifier is also called "f value"

Direct Anonymous Attestation (DAA)



!!! Mention: secret TPM identifier is also called "f value"

Idealization of Zero-knowledge

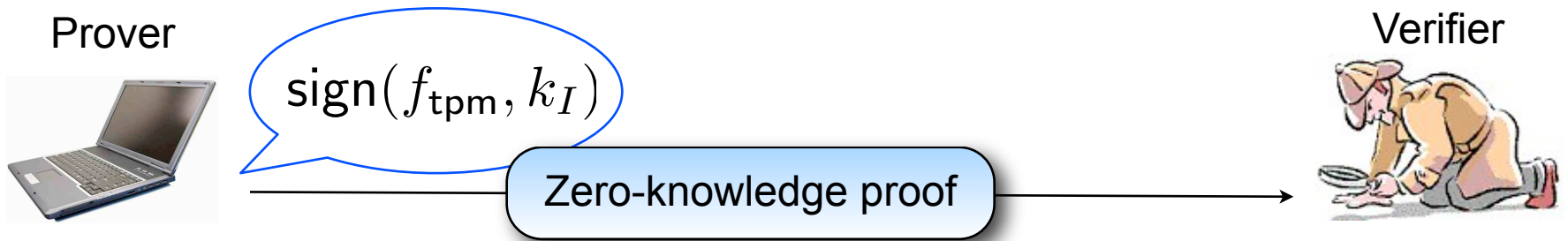


“the user knows a secret identifier and a certificate, and the certificate is a valid signature made by the issuer on the identifier”

[Backes, Maffei & Unruh, S&P 2008] [Backes & Unruh, CSF 2008]

In order to construct a formal specification of DAA in a process calculus (with the standard DY assumptions), we need a way to express zero knowledge proofs.

Idealization of Zero-knowledge

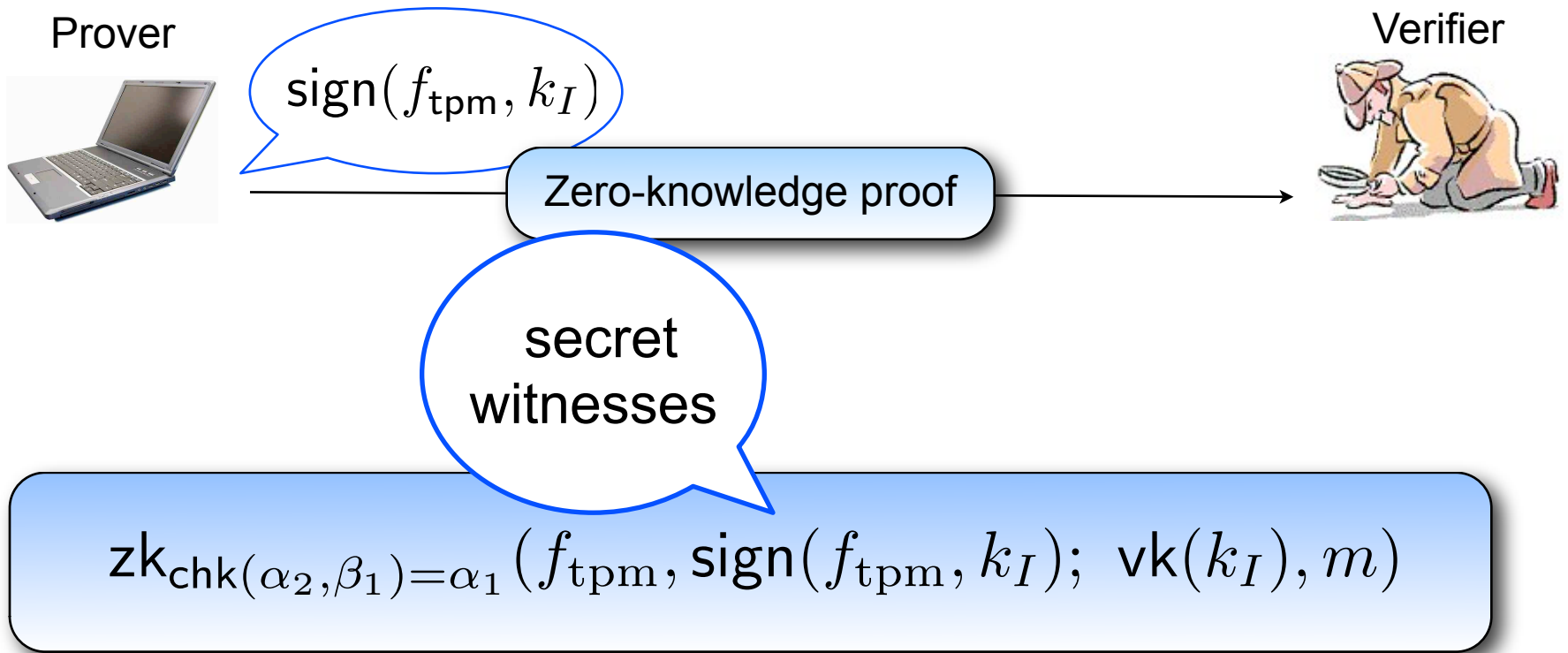


$$\text{zk}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1}(f_{\text{tpm}}, \text{sign}(f_{\text{tpm}}, k_I); \text{vk}(k_I), m)$$

[Backes, Maffei & Unruh, S&P 2008] [Backes & Unruh, CSF 2008]

In order to construct a formal specification of DAA in a process calculus (with the standard DY assumptions), we need a way to express zero knowledge proofs.

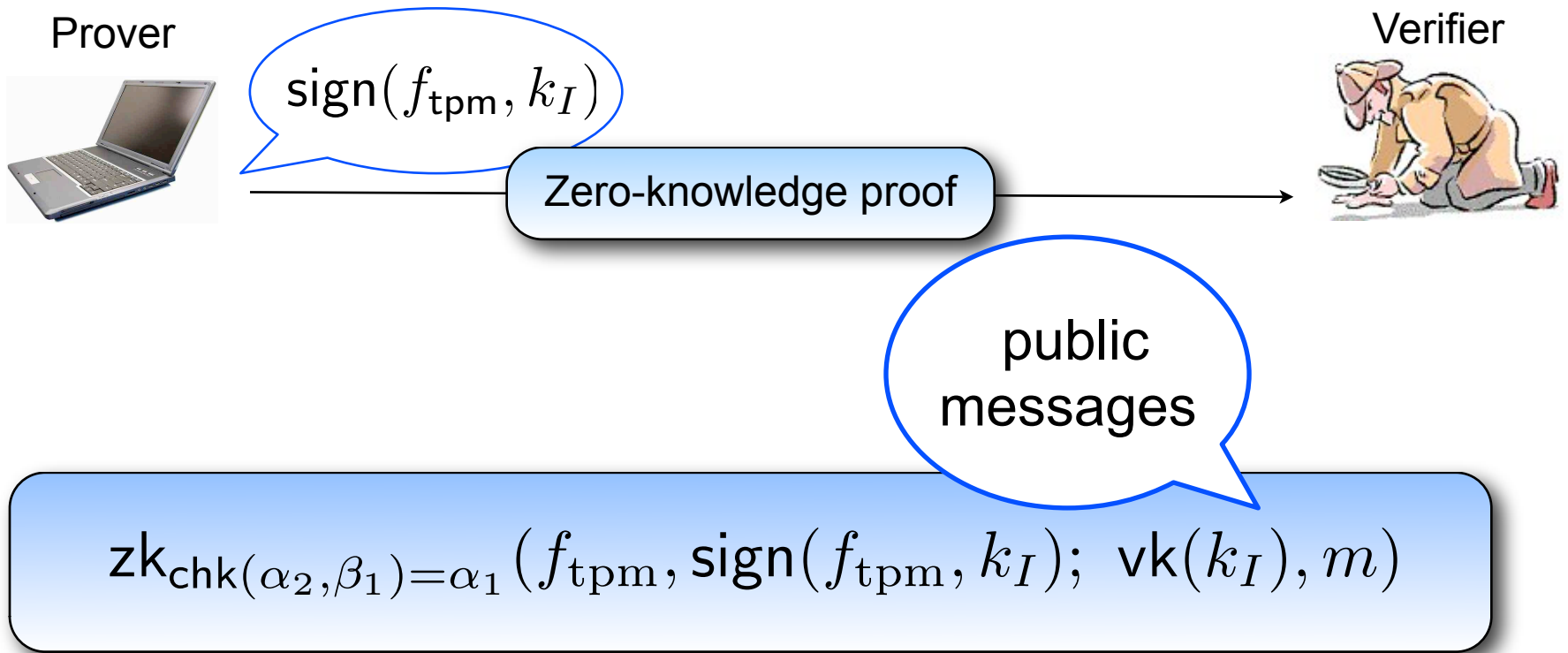
Idealization of Zero-knowledge



[Backes, Maffei & Unruh, S&P 2008] [Backes & Unruh, CSF 2008]

In order to construct a formal specification of DAA in a process calculus (with the standard DY assumptions), we need a way to express zero knowledge proofs.

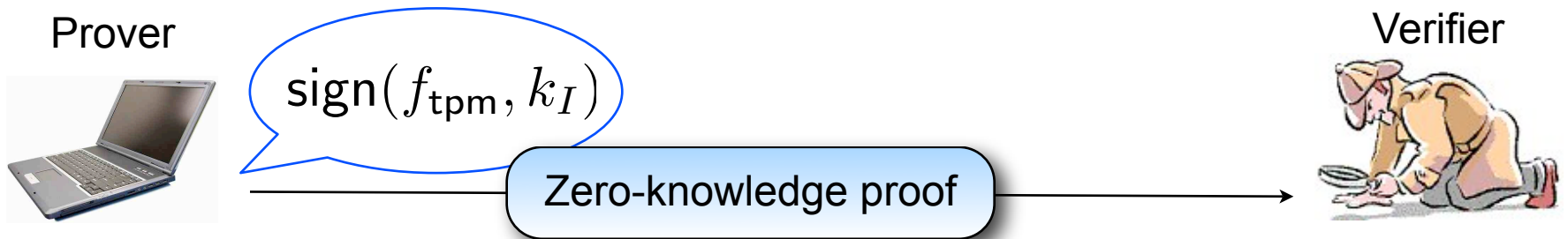
Idealization of Zero-knowledge



[Backes, Maffei & Unruh, S&P 2008] [Backes & Unruh, CSF 2008]

In order to construct a formal specification of DAA in a process calculus (with the standard DY assumptions), we need a way to express zero knowledge proofs.

Idealization of Zero-knowledge



statement

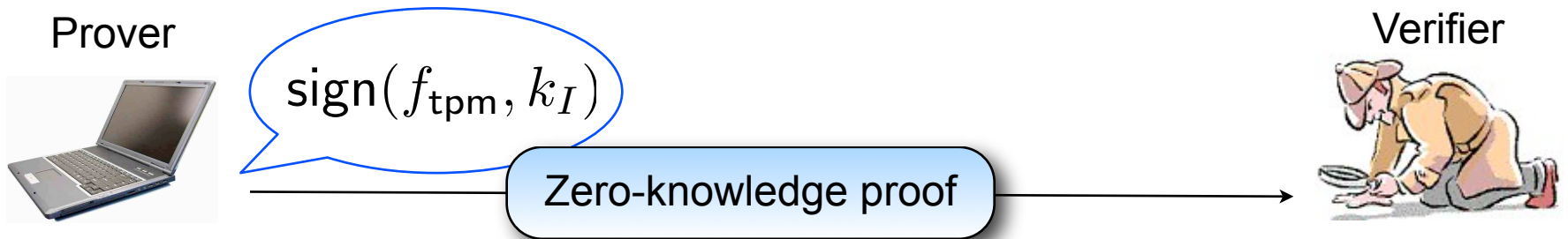
$$\text{zk}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1}(f_{\text{tpm}}, \text{sign}(f_{\text{tpm}}, k_I); \text{vk}(k_I), m)$$

$$\text{chk}(\alpha_2, \beta_1) = \alpha_1$$

[Backes, Maffei & Unruh, S&P 2008] [Backes & Unruh, CSF 2008]

In order to construct a formal specification of DAA in a process calculus (with the standard DY assumptions), we need a way to express zero knowledge proofs.

Idealization of Zero-knowledge



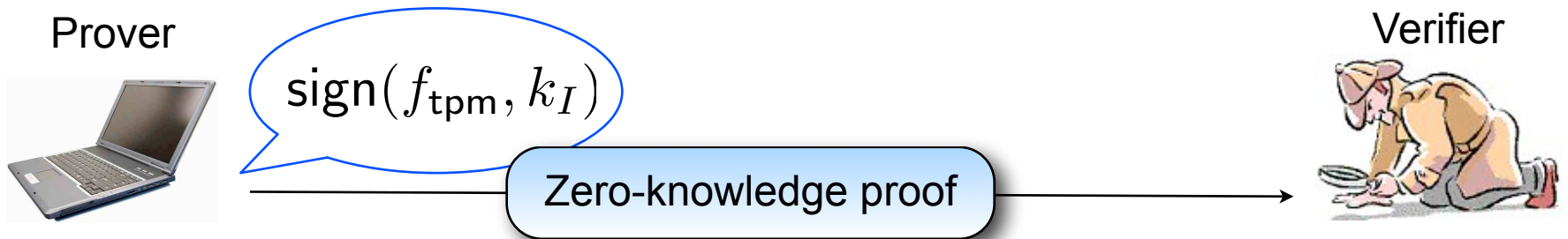
$$\text{zk}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1}(f_{\text{tpm}}, \text{sign}(f_{\text{tpm}}, k_I); \text{vk}(k_I), m)$$

Verification: $\text{chk}(\alpha_2, \beta_1) = \alpha_1$

[Backes, Maffei & Unruh, S&P 2008] [Backes & Unruh, CSF 2008]

In order to construct a formal specification of DAA in a process calculus (with the standard DY assumptions), we need a way to express zero knowledge proofs.

Idealization of Zero-knowledge



$$\text{zk}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1}(f_{\text{tpm}}, \text{sign}(f_{\text{tpm}}, k_I); \text{vk}(k_I), m)$$

Verification: $\text{chk}(\text{sign}(f_{\text{tpm}}, k_I), \text{vk}(k_I)) = f_{\text{tpm}}$ ✓

[Backes, Maffei & Unruh, S&P 2008] [Backes & Unruh, CSF 2008]

In order to construct a formal specification of DAA in a process calculus (with the standard DY assumptions), we need a way to express zero knowledge proofs.

DAA Signing Protocol (simplified)

Prover



$$\text{zk}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1}(f_{\text{tpm}}, \text{sign}(f_{\text{tpm}}, k_I); \text{vk}(k_I), m)$$

Verifier



new k_I

new f_{tpm}

Prover | Verifier | Issuer

Prover = new m



$$\text{out}(c, \text{zk}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1}(f_{\text{tpm}}, \text{sign}(f_{\text{tpm}}, k_I); \text{vk}(k_I), m))$$

Verifier = in(c, x).



$$\text{let } y_m = \text{ver}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1}(x; \text{vk}(k_I)) \text{ then}$$

matching will
help typing

!!! DAA extremely simplified in my example -- just the signing protocol

Security Annotations

Prover



Verifier



$$\text{zk}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1}(f_{\text{tpm}}, \text{sign}(f_{\text{tpm}}, k_I); \text{vk}(k_I), m)$$

assume $\forall m. (\exists x_f. \text{Send}(x_f, m) \wedge \text{OkTPM}(x_f) \Rightarrow \text{Authenticate}(m))$ |

new k_I

new f_{tpm}

Prover | Verifier | Issuer

authorization policy
(in some logic)

Prover = new m



assume $\text{Send}(f_{\text{tpm}}, m)$ |

out($c, \text{zk}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1}(f_{\text{tpm}}, \text{sign}(f_{\text{tpm}}, k_I); \text{vk}(k_I), m)$)

Verifier = in(c, x).



let $y_m = \text{ver}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1}(x; \text{vk}(k_I))$

assert $\text{Authenticate}(y_m)$

Safety

Asserts are entailed by
the current assumes

!!! In the authorization policy “OkTPM(xf)” is assumed by the Issuer

!!! The property verified by our type system is that in all protocol runs the assertions are logically entailed by the current assumptions

Basic Types

assume $\forall m. (\exists x_f. \text{Send}(x_f, m) \wedge \text{OkTPM}(x_f) \Rightarrow \text{Authenticate}(m))$ |

new k_I

new f_{tpm}

Prover | Verifier | Issuer

Prover = new m



assume $\text{Send}(f_{\text{tpm}}, m)$ |

$\text{out}(c, \text{zk}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1}(f_{\text{tpm}}, \text{sign}(f_{\text{tpm}}, k_I); \text{vk}(k_I), m))$

Verifier = $\text{in}(c, x)$.



let $y_m = \text{ver}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1}(x; \text{vk}(k_I))$ then

assert $\text{Authenticate}(y_m)$

[Fournet, Gordon & Maffei, CSF 2007]

!!! When checking a signature with the corresponding verification key, we can actually infer that the signed message is of type Private and the OkTPM predicate holds for this message.

!!! However, in the DAA protocol we do not send the signature to the verifier so that he can directly check it. We only prove to know such a valid signature.

Basic Types


assume $\forall m. (\exists x_f. \text{Send}(x_f, m) \wedge \text{OkTPM}(x_f) \Rightarrow \text{Authenticate}(m))$ |

new k_I

new f_{tpm}

Prover | Verifier | Issuer

Prover = new m : Un
 assume Send
 out($c, \text{zk}_{\text{chk}}(\dots); \text{vk}(k_I), m)$)

Verifier = in(c, x).
 let $y_m = \text{ver}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1 \dots (m_I)}$ then
 assert $\text{Authenticate}(y_m)$

Type of
messages known to
the attacker

[Fournet, Gordon & Maffei, CSF 2007]

!!! When checking a signature with the corresponding verification key, we can actually infer that the signed message is of type Private and the OkTPM predicate holds for this message.

!!! However, in the DAA protocol we do not send the signature to the verifier so that he can directly check it. We only prove to know such a valid signature.

Basic Types

assume $\forall m. (\exists x_f. \text{Send}(x_f, m) \wedge \text{OkTPM}(x_f) \Rightarrow \text{Authenticate}(m))$ |

new k_I

new $f_{\text{tpm}}: \text{Private}$

Prover | Verifier

Prover =



new n

assume \exists

out($c, \text{zk}_{\text{chk}(\alpha_2, \beta_1)=\alpha_1}(f_{\text{tpm}}, \text{sign}(f_{\text{tpm}}, k_I)); \text{vk}(k_I), m)$)

Verifier = in(c, x).



let $y_m = \text{ver}_{\text{chk}(\alpha_2, \beta_1)=\alpha_1}(x; \text{vk}(k_I))$ then

assert $\text{Authenticate}(y_m)$

Type of messages
unknown to the attacker

[Fournet, Gordon & Maffei, CSF 2007]

!!! When checking a signature with the corresponding verification key, we can actually infer that the signed message is of type Private and the OkTPM predicate holds for this message.

!!! However, in the DAA protocol we do not send the signature to the verifier so that he can directly check it. We only prove to know such a valid signature.

Basic Types

assume $\forall m. (\exists x_f. \text{Send}(x_f, m) \wedge \text{OkTPM}(x_f) \Rightarrow \text{Authenticate}(m))$ |

new $k_I: \text{SigKey}(\langle x_f : \text{Private} \rangle \{ \text{OkTPM}(x_f) \})$

new $f_{\text{tpm}}: \text{Private}$

Prover | Verifier | Issuer

Prover = new $m: \text{Un}$



assume $\text{Send}(f_{\text{tpm}}, m)$ |

out($c, \text{zk}_{\text{chk}(\alpha_2, \beta_1)=\alpha_1}(f_{\text{tpm}}, \text{sign}(f_{\text{tpm}}, k_I); \text{vk}(k_I), m)$)

Verifier = in(c, x).



let $y_m = \text{ver}_{\text{chk}(\alpha_2, \beta_1)=\alpha_1}(x; \text{vk}(k_I))$ then

assert $\text{Authenticate}(y_m)$

Type of keys used
to sign Private messages for which
OkTPM holds

[Fournet, Gordon & Maffei, CSF 2007]

!!! When checking a signature with the corresponding verification key, we can actually infer that the signed message is of type Private and the OkTPM predicate holds for this message.


!!! However, in the DAA protocol we do not send the signature to the verifier so that he can directly check it. We only prove to know such a valid signature.


Basic Types



The type of the key allows us to “transfer” predicates from the prover to the verifier!

assume $\forall m. (\exists x_f. \text{Send}(x_f, m) \wedge \text{OkTPM}(x_f) \Rightarrow \text{Authenticate}(m))$ |
 new $k_I: \text{SigKey}(\langle x_f : \text{Private} \rangle \{ \text{OkTPM}(x_f) \})$
 new $f_{\text{tpm}}: \text{Private}$
 Prover | Verifier | Issuer

Prover = new $m: \text{Un}$

 assume $\text{Send}(f_{\text{tpm}}, m)$ |
 out($c, \text{zk}_{\text{chk}(\alpha_2, \beta_1)=\alpha_1}(f_{\text{tpm}}, \text{sign}(f_{\text{tpm}}, k_I); \text{vk}(k_I), m)$)

Verifier = in(c, x).

 let $y_m = \text{ver}_{\text{chk}(\alpha_2, \beta_1)=\alpha_1}(x; \text{vk}(k_I))$ then
 assert $\text{Authenticate}(y_m)$

[Fournet, Gordon & Maffei, CSF 2007]

!!! When checking a signature with the corresponding verification key, we can actually infer that the signed message is of type `Private` and the `OkTPM` predicate holds for this message.

!!! However, in the DAA protocol we do not send the signature to the verifier so that he can directly check it. We only prove to know such a valid signature.

Basic Types

But, the verifier can't use the key to check a certificate he never receives.

Worse, ZK don't necessarily rely on keys!



assume $\forall m. (\exists x_f. \text{Send}(x_f, m) \wedge \text{OkTPM}(x_f) \Rightarrow \text{Authenticate}(m))$ |

new k_I : **SigKey**($\langle x_f : \text{Private} \rangle \{ \text{OkTPM}(x_f) \}$)

new f_{tpm} : **Private**

Prover | Verifier | Issuer

Prover = new m : **Un**



assume $\text{Send}(f_{\text{tpm}}, m)$ |

out($c, \text{zk}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1}(f_{\text{tpm}}, \text{sign}(f_{\text{tpm}}, k_I); \text{vk}(k_I), m)$)

Verifier = in(c, x).



let $y_m = \text{ver}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1}(x; \text{vk}(k_I))$ then

assert $\text{Authenticate}(y_m)$

[Fournet, Gordon & Maffei, CSF 2007]

!!! When checking a signature with the corresponding verification key, we can actually infer that the signed message is of type **Private** and the **OkTPM** predicate holds for this message.

!!! However, in the DAA protocol we do not send the signature to the verifier so that he can directly check it. We only prove to know such a valid signature.

Typing Zero-knowledge Proofs

Our solution:

User gives a type to each statement proved by ZK

!!! Don't panic ... this is the only ZK type I'm going to show !!!

- For each statement in the protocol the user needs to annotate such a type
- The logical formula that is going to be transferred by the proof – where the private messages are existentially quantified

Typing Zero-knowledge Proofs

Prover



Verifier



$$zk_{chk(\alpha_2, \beta_1) = \alpha_1}(f_{tpm}, \text{sign}(f_{tpm}, k_I); vk(k_I), m)$$

Type of
public messages

$$ZK_{chk(\alpha_2, \beta_1) = \alpha_1} (\langle y_k : \text{VerKey}(\dots), y_m : \text{Un} \rangle \{ \exists x_f, x_s. \text{Send}(x_f, y_m) \wedge \text{OkTPM}(x_f) \})$$

!!! Don't panic ... this is the only ZK type I'm going to show !!!

- For each statement in the protocol the user needs to annotate such a type
- The logical formula that is going to be transferred by the proof - where the private messages are existentially quantified

Typing Zero-knowledge Proofs

Prover



Verifier



$$zk_{chk(\alpha_2, \beta_1) = \alpha_1}(f_{tpm}, \text{sign}(f_{tpm}, k_I); vk(k_I), m)$$

$$ZK_{chk(\alpha_2, \beta_1) = \alpha_1} (\langle y_k : \text{VerKey}(\dots), y_m : \text{Un} \rangle \{ \exists x_f, x_s. \text{Send}(x_f, y_m) \wedge \text{OkTPM}(x_f) \})$$

Logical formula where
the secret witnesses are
existentially quantified

!!! Don't panic ... this is the only ZK type I'm going to show !!!

- For each statement in the protocol the user needs to annotate such a type
- The logical formula that is going to be transferred by the proof - where the private messages are existentially quantified

Type-checking the Prover

$$\text{ZK}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1} \left(\langle y_k : \text{VerKey}(\langle x : \text{Private} \rangle \{ \text{OkTPM}(x) \}), y_m : \text{Un} \rangle \right. \\ \left. \{ \exists x_f, x_s. \text{Send}(x_f, y_m) \wedge \text{OkTPM}(x_f) \} \right)$$

$\Gamma = \dots$
 $k_I : \text{SigKey}(\langle x_f : \text{Private} \rangle \{ \text{OkTPM}(x_f) \}),$
 $m : \text{Un},$
 \dots
 $\text{OkTPM}(f_{\text{tpm}}),$
 $\text{Send}(f_{\text{tpm}}, m)$



Type of public messages

Prover = \dots
 $\text{out}(c, \text{zk}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1}(f_{\text{tpm}}, \text{sign}(f_{\text{tpm}}, k_I); \text{vk}(k_I), m))$



!!! Begin !!! Since type-checking is compositional we are going to check the prover and the verifier independently. We start with the prover.

!!! the type system ensures that the formula in the ZK type is entailed in the prover's environment



Type-checking the Prover

$$\text{ZK}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1} \left(\begin{array}{l} \langle y_k : \text{VerKey}(\langle x : \text{Private} \rangle \{ \text{OkTPM}(x) \}), y_m : \text{Un} \rangle \\ \{ \exists x_f, x_s. \text{Send}(x_f, y_m) \wedge \text{OkTPM}(x_f) \} \end{array} \right)$$

$\Gamma = \dots$
 $k_I : \text{SigKey}(\langle x_f : \text{Private} \rangle \{ \text{OkTPM}(x_f) \}),$
 $m : \text{Un},$
 \dots
 $\text{OkTPM}(f_{\text{tpm}}),$
 $\text{Send}(f_{\text{tpm}}, m)$



- Type of public messages
- Logical formula entailed

Prover = \dots
  $\text{out}(c, \text{zk}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1}(f_{\text{tpm}}, \text{sign}(f_{\text{tpm}}, k_I); \text{vk}(k_I), m))$

!!! Begin !!! Since type-checking is compositional we are going to check the prover and the verifier independently. We start with the prover.

!!! the type system ensures that the formula in the ZK type is entailed in the prover's environment

Type-checking the Verifier

$$\text{ZK}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1} \left(\begin{array}{l} \langle y_k : \text{VerKey}(\langle x : \text{Private} \rangle \{ \text{OkTPM}(x) \}), y_m : \text{Un} \rangle \\ \{ \exists x_f, x_s. \text{Send}(x_f, y_m) \wedge \text{OkTPM}(x_f) \} \end{array} \right)$$

$$\Gamma = \dots$$

$$k_I : \text{SigKey}(\langle x_f : \text{Private} \rangle \{ \text{OkTPM}(x_f) \}),$$


$$\dots$$

$$\forall m. ((\exists x_f. \text{Send}(x_f, m) \wedge \text{OkTPM}(x_f) \Rightarrow \text{Authenticate}(m)),$$

If verification succeeds, can we assume the formula in ZK type?

In general not, the proof can come from untyped adversary!

Verifier = $\text{in}(c, x).$

 $\text{let } y_m = \text{ver}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1}(x; \text{vk}(k_I)) \text{ then}$
 $\text{assert } \text{Authenticate}(y_m)$

!!! TODO: relate this to signatures? there checking the signature with the verification key of the sender gives you strong guarantees ... we would like to have the same for zk, but it doesn't seem evident how to do that

Type-checking the Verifier

$$\text{ZK}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1} \left(\begin{array}{l} \langle y_k : \text{VerKey}(\langle x : \text{Private} \rangle \{ \text{OkTPM}(x) \}), y_m : \text{Un} \rangle \\ \{ \exists x_f, x_s. \text{Send}(x_f, y_m) \wedge \text{OkTPM}(x_f) \} \end{array} \right)$$



Conceptual issue


How can we know whether the zero-knowledge proof comes from an honest participant or from the adversary?!

Type-checking the Verifier

$$\text{ZK}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1} \left(\begin{array}{l} \langle y_k : \text{VerKey}(\langle x : \text{Private} \rangle \{ \text{OkTPM}(x) \}), y_m : \text{Un} \rangle \\ \{ \exists x_f, x_s. \text{Send}(x_f, y_m) \wedge \text{OkTPM}(x_f) \} \end{array} \right)$$

$$\begin{aligned} \Gamma = & \dots \\ & k_I : \text{SigKey}(\langle x_f : \text{Private} \rangle \{ \text{OkTPM}(x_f) \}) \\ & \dots \\ & \forall m. ((\exists x_f. \text{Send}(x_f, m) \wedge \text{OkTPM}(x_f)) \rightarrow \text{OkTPM}(m)) \\ & \text{chk}(\alpha_2, \beta_1) = \alpha_1 \end{aligned}$$

The statement
instantiated with the matched
messages is valid
(by the semantics)

Verifier = $\text{in}(c, x).$

 $\text{let } y_m = \text{ver}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1}(x; \text{vk}(k_I)) \text{ then}$
 $\text{assert Authenticate}(y_m)$




Type-checking the Verifier

$$\text{ZK}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1} \left(\langle y_k : \text{VerKey}(\langle x : \text{Private} \rangle \{ \text{OkTPM}(x) \}), y_m : \text{Un} \rangle \right. \\ \left. \{ \exists x_f, x_s. \text{Send}(x_f, y_m) \wedge \text{OkTPM}(x_f) \} \right)$$

$\Gamma = \dots$
 $k_I : \text{SigKey}(\langle x_f : \text{Private} \rangle \{ \text{OkTPM}(x_f) \})$
 \dots
 $\forall m. ((\exists x_f. \text{Send}(x_f, m) \wedge \text{OkTPM}(x_f)) \rightarrow \text{Auth}(m, k_I))$
 $\text{chk}(x_s, \text{vk}(k_I)) = x_f$

The statement
instantiated with the matched
messages is valid
(by the semantics)

Verifier = $\text{in}(c, x).$

 $\text{let } y_m = \text{ver}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1}(x; \text{vk}(k_I)) \text{ then}$
 $\text{assert Authenticate}(y_m)$




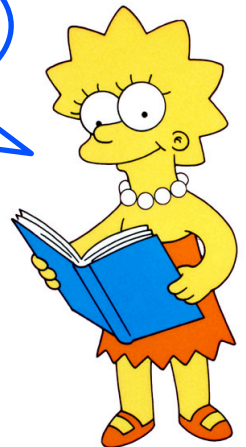
Type-checking the Verifier

$$\text{ZK}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1} \left(\begin{array}{l} \langle y_k : \text{VerKey}(\langle x : \text{Private} \rangle \{ \text{OkTPM}(x) \}), y_m : \text{Un} \rangle \\ \{ \exists x_f, x_s. \text{Send}(x_f, y_m) \wedge \text{OkTPM}(x_f) \} \end{array} \right)$$

$\Gamma = \dots$
 $k_I : \text{SigKey}(\langle x_f : \text{Private} \rangle \{ \text{OkTPM}(x_f) \}),$
 \dots
 $\forall m. ((\exists x_f. \text{Send}(x_f, m) \wedge \text{OkTPM}(x_s) \rightarrow \text{Authenticate}(m)),$
 $\text{chk}(x_s, \text{vk}(k_I)) = x_f$
 $x_f : \text{Private}$

The type of $\text{vk}(k_I)$ gives us the type of x_f (existentially quantified)


 Verifier = $\text{in}(c, x).$
 let $y_m = \text{ver}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1}(x; \text{vk}(k_I))$ then
 assert $\text{Authenticate}(y_m)$



Type-checking the Verifier

$$\text{ZK}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1} \left(\begin{array}{l} \langle y_k : \text{VerKey}(\langle x : \text{Private} \rangle \{ \text{OkTPM}(x) \}), y_m : \text{Un} \rangle \\ \{ \exists x_f, x_s. \text{Send}(x_f, y_m) \wedge \text{OkTPM}(x_f) \} \end{array} \right)$$

$\Gamma = \dots$

$k_I : \text{SigKey}(\langle x_f : \text{Private} \rangle \{ \text{OkTPM}(x_f) \})$

\dots

$\forall m. ((\exists x_f. \text{Send}(x_f, m) \wedge \text{OkTPM}(x_f)))$

$\text{chk}(x_s, \text{vk}(k_I)) = x_f$

$x_f : \text{Private}$

The prover is honest, since he knows a message of type Private!



Verifier = $\text{in}(c, x).$




let $y_m = \text{ver}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1}(x; \text{vk}(k_I))$ then
assert $\text{Authenticate}(y_m)$

Type-checking the Verifier

$$\text{ZK}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1} \left(\begin{array}{l} \langle y_k : \text{VerKey}(\langle x : \text{Private} \rangle \{ \text{OkTPM}(x) \}), y_m : \text{Un} \rangle \\ \{ \exists x_f, x_s. \text{Send}(x_f, y_m) \wedge \text{OkTPM}(x_f) \} \end{array} \right)$$

$\Gamma = \dots$
 $k_I : \text{SigKey}(\langle x_f : \text{Private} \rangle \{ \text{OkTPM}(x_f) \})$
 \dots
 $\forall m. ((\exists x_f. \text{Send}(x_f, m) \wedge \text{OkTPM}(x_f)))$
 $\text{chk}(x_s, \text{vk}(k_I)) = x_f$
 $x_f : \text{Private}$

We can now exploit the formula in the ZK type


Verifier = $\text{in}(c, x).$

 $\text{let } y_m = \text{ver}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1}(x; \text{vk}(k_I)) \text{ then}$
 $\text{assert Authenticate}(y_m)$



Type-checking the Verifier

$$\text{ZK}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1} \left(\begin{array}{l} \langle y_k : \text{VerKey}(\langle x : \text{Private} \rangle \{ \text{OkTPM}(x) \}), y_m : \text{Un} \rangle \\ \{ \exists x_f, x_s. \text{Send}(x_f, y_m) \wedge \text{OkTPM}(x_f) \} \end{array} \right)$$

$\Gamma = \dots$
 $k_I : \text{SigKey}(\langle x_f : \text{Private} \rangle \{ \text{OkTPM}(x_f) \}),$
 \dots
 $\forall m. ((\exists x_f. \text{Send}(x_f, m) \wedge \text{OkTPM}(x_f) \Rightarrow \text{Authenticate}(m)),$
 $\text{chk}(x_s, \text{vk}(k_I)) = x_f$
 $x_f : \text{Private}$
 $\exists x_f, x_s. \text{Send}(x_f, y_m) \wedge \text{OkTPM}(x_f)$

Verifier = $\text{in}(c, x).$

 $\text{let } y_m = \text{ver}_{\text{chk}(\alpha_2, \beta_1) = \alpha_1}(x; \text{vk}(k_I)) \text{ then}$
 $\text{assert } \text{Authenticate}(y_m)$



Take Home

- ▶ ZK proofs are given *dependent types* where the witnesses are *existentially quantified*
- ▶ *The prover* can only prove statements for which the formula in the ZK type holds
- ▶ *The verifier* can assume the formula in the ZK type
 - if the formula is entirely derived from the proved statement (most often much too weak)
 - if he can somehow infer that the proof was constructed by an *honest prover (type-checked)*

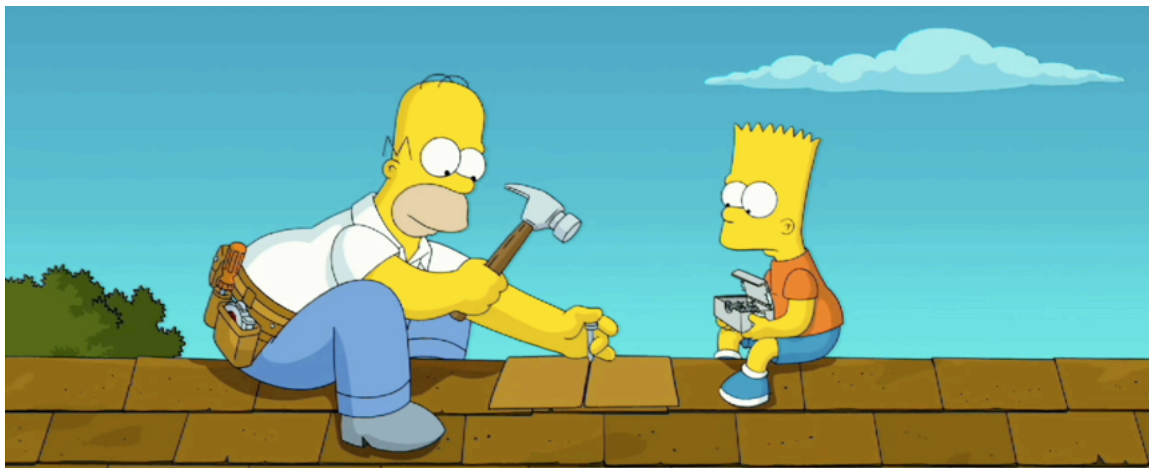


--- removed:

- we statically infer this by looking at the statement and the type of the matched public messages
- use fact that adversary doesn't know Private msg.

Implementation

- ▶ Type-checker written in O'CamI (~5000 LOC)
- ▶ Uses automatic prover for discharging FOL formulas
- ▶ Extensible - very easy to add arbitrary primitives + types
- ▶ Efficient - the complete analysis of DAA takes 0.7s
- ▶ Available under the Apache License:
<http://www.infsec.cs.uni-sb.de/projects/zk-typechecker/>
- ▶ Kudos to Stefan Lorenz, Kim Pecina and Thorsten Tarrach



- we implemented our type system in O'CamI and use first-order theorem prover to discharge proof obligations
- the type-checker is extensible, so it's very easy to add to extend it with arbitrary cryptographic primitives and base types
- the analysis of the complete DAA protocol takes less than a second (**not just the small simplified fragment I presented in this talk**)
- the implementation is available under the Apache license
- you can also find this link on my website
- last but not least, I would like to thank the students who helped us with the implementation

Ongoing Work



- ▶ *Type-checking a model of **Civitas***
 - Remote electronic voting system [Clarkson, Chong & Myers, S&P 2008]
- ▶ *Type-checking implementations of protocols that employ zero-knowledge*
 - Trying to extend [Bengtson et al., CSF 2008]
- ▶ *Improving security despite compromise*
 - Execute original protocol, but add zero-knowledge to prove correct behavior to remote parties
- ▶ *Idealizing interactive zero-knowledge proofs and analyzing the protocols that use them*

THANK YOU!

Currently we are working on various extensions of this work ...

- We are **verifying the protocol used by the Civitas remote electronic voting system** using our type-checker
- We are trying to apply some of the same ideas of our type system to verifying real **implementations** of security protocols that employ zero-knowledge proofs
- We are also working on an **automatic transformation to improve the security of protocols against corrupted participants**. (The main idea is to use zero-knowledge to prove correct behavior to remote parties.)
- Finally, we are **looking at ways to idealize interactive zero-knowledge proofs and to analyze the security of the protocols** that use such proofs. (The hope here is that one can capture communication patterns using session types for instance.)