

When is a PKCS#11 Configuration Secure?

Sibylle Fröschle¹ Nils Sommer²

¹University of Oldenburg
Germany

²MWR InfoSecurity
UK

July 21, 2010

Public Key Cryptographic Standard (PKCS) #11

- ▶ Generic API for a wide range of devices and use cases.



Smartcard



eToken



Hardware Security Module

Security API

Export Key
Import Key
Generate Key
Encrypt Data
Decrypt Data
...



Alice

Public Key Cryptographic Standard (PKCS) #11

- ▶ Generic API for a wide range of devices and use cases.



Smartcard



eToken



Hardware Security Module

Security API

Export Key
Import Key
Generate Key
Encrypt Data
Decrypt Data
...



Alice

“How can you be sure that there isn't some chain of 17 transactions which will leak a clear key?” Anderson 2000

PKCS #11

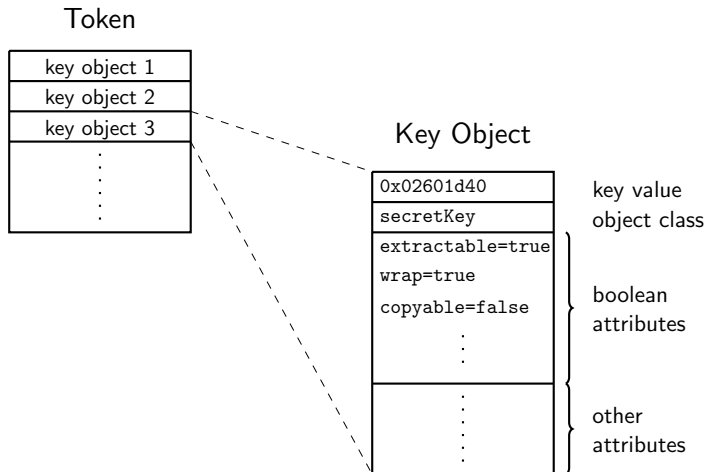
- ▶ Logical view:

Token

key object 1
key object 2
key object 3
⋮

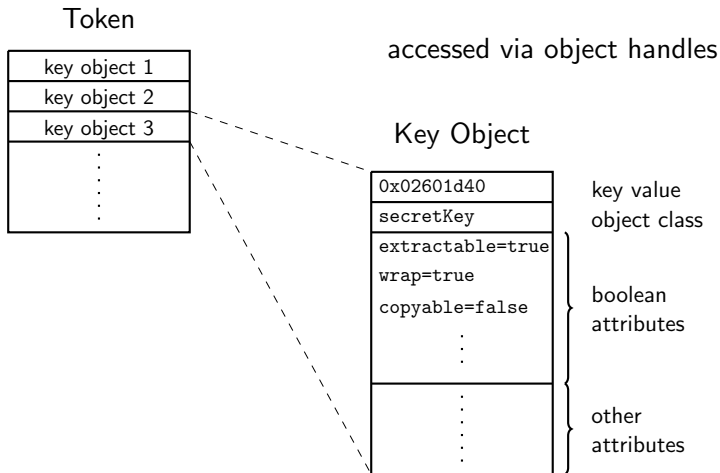
PKCS #11

► Logical view:



PKCS #11

► Logical view:

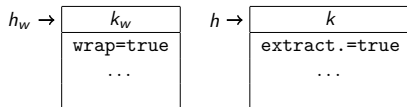


PKCS#11 Some Commands

- ▶ Export key k encrypted under k_w

H \rightarrow T: `wrap h_w h`

T \rightarrow H: `{ k }` _{k_w}

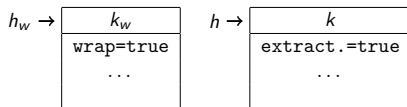


PKCS#11 Some Commands

- ▶ Export key k encrypted under k_w

H \rightarrow T: wrap h_w h

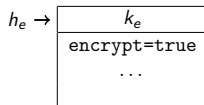
T \rightarrow H: $\{k\}_{k_w}$



- ▶ Encrypt data d under k_e

H \rightarrow T: encrypt h_e d

T \rightarrow H: $\{d\}_{k_e}$

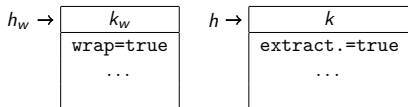


PKCS#11 Some Commands

- ▶ Export key k encrypted under k_w

H → T: wrap h_w h

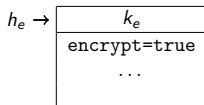
T → H: $\{k\}_{k_w}$



- ▶ Encrypt data d under k_e

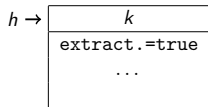
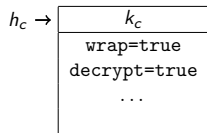
H → T: encrypt h_e d

T → H: $\{d\}_{k_e}$

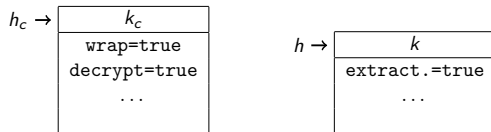


- ▶ Inverse commands unwrap, decrypt

Key Separation Attack (Clulow, 2003)



Key Separation Attack (Clulow, 2003)



Using key k_c in two conflicting roles to reveal key k ...

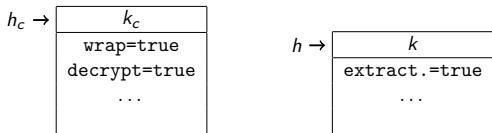
H \rightarrow T : `wrap` h_c h

T \rightarrow H : $\{k\}_{k_c}$

H \rightarrow T : `decrypt` h_c $\{k\}_{k_c}$

T \rightarrow H : k

Key Separation Attack (Clulow, 2003)



Using key k_c in two conflicting roles to reveal key k ...

H \rightarrow T : wrap h_c h

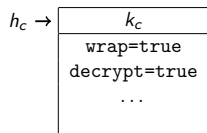
T \rightarrow H : $\{k\}_{k_c}$

H \rightarrow T : decrypt h_c $\{k\}_{k_c}$

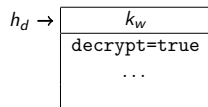
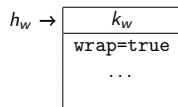
T \rightarrow H : k

More complicated versions exist ...

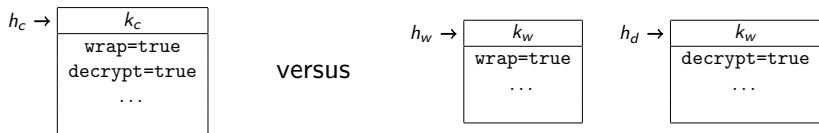
More Key Separation Attacks (Delaune et al. 08)



versus



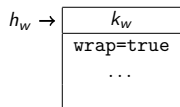
More Key Separation Attacks (Delaune et al. 08)



Commands to change attributes and make copies:

- ▶ SetAttributeValue
- ▶ CopyObject
- ▶ wrap and unwrap
- ▶ CreateObject (but key already known to intruder)

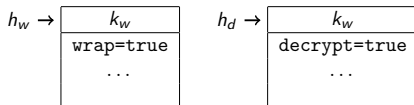
Manipulating the roles of a key



attribute template

H \rightarrow T : `copyObject h_w t`
where t specifies: `wrap=false, decrypt=true`

T \rightarrow H : h_d
where h_d is a new handle



- ▶ The success of these commands depends on the device

PKCS#11 Security

Many potential vulnerabilities!

Many potential vulnerabilities!

However:

- ▶ generic API for wide range of devices and use cases.
- ▶ up to the device, which functionality is indeed provided.

Many potential vulnerabilities!

However:

- ▶ generic API for wide range of devices and use cases.
- ▶ up to the device, which functionality is indeed provided.

Security is always up to the configuration!

PKCS#11 Configuration

1. Available commands
2. Additional constraints on commands
3. Vendor defined extensions
4. Pre-configured key objects

Experiments

Is the wrap/decrypt attack real?

How do configurations of real devices protect their keys?

- ▶ Aladdin eToken
- ▶ Siemens HiPath Scurity
- ▶ Estonian ID



Pre-configured Keys: all have public/private RSA key pair

Experiments

- Attack 1:** Generate a sensitive key on the token and attack it by the wrap/decrypt attack.
- Attack 2:** Run the wrap/decrypt attack against the existing private RSA key.

Experiments

Attack 1: Generate a sensitive key on the token and attack it by the wrap/decrypt attack.

Attack 2: Run the wrap/decrypt attack against the existing private RSA key.

Token	A	S	If failed reason for failure
Aladdin eToken	1	yes	
	2	no	existing key: extractable=false
HiPath Scurity	1	no	not possible to generate <i>any extractable</i> key
	2	no	no symmetric key with wrap/decrypt conflict existing key: extractable=false
Estonian ID	1	no	not possible to generate <i>any</i> key
	2	no	no key with wrap/decrypt conflict existing key: extractable=false

A ... Attack

S ... Successful?

`extractable = false`

- ▶ Key objects with `extractable = false` are not vulnerable against any of the known attacks!
- ▶ Constraints by the standard concerning modifying and copying:

“...in the course of copying a secret key, a key’s `CKA_EXTRACTABLE` attribute may be changed from `CK_TRUE` to `CK_FALSE`, but not the other way around.”

Are keys generated with `extractable = false` always secure?
How to prove such a statement formally?

Challenges

How to prove such a statement formally?

1. Model-checking approach does not scale.
2. Need to consider features that cannot be modelled by the usual protocol models:
attribute templates, constraints on commands

Approach

Crystallize informal reasoning into novel approach:

1. Model the data types used by the security API by a **first-order language**.
2. Draw from the methodology used in program verification: **first-order linear time logic (FOLTL)** (Kröger, Merz)
3. Draw from the success of backwards analysis in security protocol analysis:
FOLTL extended by past operators and a **tableau proof method**.
4. Recover **monotonicity of state** by working with a bisimilar model.

Security API System

First-order labelled state transition system (FOSTS) Γ
over a temporal signature *TSIG*:

SIG a standard first-order signature

models the data types of the standard:

ASET: attribute set or template, t

MESG: keys, encryptions, $k, \{k\}_{k_e}$

R flexible predicate symbols

describe the states of the system

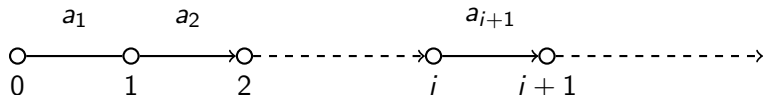
A action predicates

model the API commands and the intruder's own set
of actions

Linear-time semantics

- ▶ set of all possible execution sequences

Execution sequence:

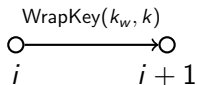


States = a set of flexible predications that hold at the state.

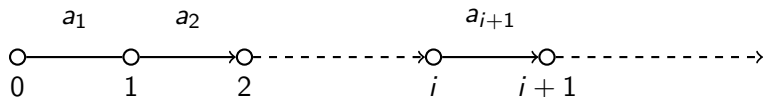
$\text{iknows}(m)$ “The intruder knows message m ”.

$\text{tcontains}(k, t)$ “The token contains a key object with value k
and attribute set t ”

Transitions are labelled by action predications:



FOLTL: atomic formulas



Non-flexible atomic formulas:

- ▶ $t.extractable = true$

$K, i \models \phi$ iff ϕ holds in the structure that interpretes SIG

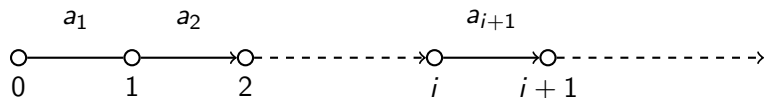
Flexible atomic formulas:

- ▶ $iknows(m), tcontains(k, t)$
- ▶ “As next action the key k will be exported under key k_w ”:

$exec\ WrapKey(k_w, k)$

$K, i \models \phi$ iff ϕ holds at s_i .

FOLTL: temporal formulas



- ▶ $\bigcirc\phi$ 'Next ϕ '

$$K, i \models \bigcirc\phi \quad \text{iff} \quad K, i+1 \models \phi$$

- ▶ $\Box\phi$ 'Always ϕ '

$$K, i \models \Box\phi \quad \text{iff} \quad K, j \models \phi \text{ for all } j \geq i$$

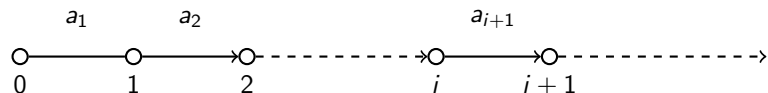
- ▶ $\ominus\phi$ 'Previous ϕ '

$$K, i \models \ominus\phi \quad \text{iff} \quad \text{if } i > 0 \text{ then } K, i-1 \models \phi$$

- ▶ $\exists\phi$ 'Has always been ϕ '

$$K, i \models \exists\phi \quad \text{iff} \quad K, j \models \phi \text{ for all } j \leq i$$

FOLTL

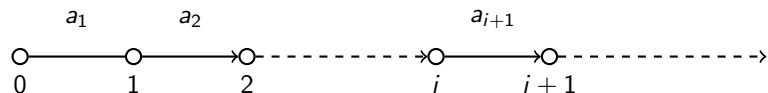


- ▶ Propositional connectives: \wedge , \vee , \rightarrow
- ▶ Quantifiers: \exists , \forall

“The token contains a key object with value k and attribute extractable set to true”:

$$\text{hasRole}(k, \text{extractable}) \equiv \exists t. \text{tcontains}(k, t) \wedge t.\text{extractable} = \text{true}$$

FOLTL



- ▶ Propositional connectives: $\wedge, \vee, \rightarrow$
- ▶ Quantifiers: \exists, \forall

“The token contains a key object with value k and attribute extractable set to true”:

$$\text{hasRole}(k, \text{extractable}) \equiv \exists t. \text{tcontains}(k, t) \wedge t.\text{extractable} = \text{true}$$

Validity for execution sequences and FOSTSs:

- ▶ $K \models \phi$ iff $K, i \models \phi$ for all i .
- ▶ $\Gamma \models \phi$ iff $K \models \phi$ for all execution sequences K of Γ .

Cryptoki theory

Exec axioms:

... enabling conditions for the commands and consequences for the next state

- ▶ $\text{exec WrapKey}(k_w, k) \rightarrow \text{hasRole}(k, \text{extractable}) \wedge \text{hasRole}(k_w, \text{wrap})$
- ▶ $\text{exec GenerateKey}(k, t) \rightarrow \bigcirc \neg \text{iknows}(k) \wedge \dots$

Obtain axioms:

... how the intruder can obtain a new situation beneficial for his attack:

- ▶ ObtainKey:

$$\neg \text{iknows}(k) \wedge \bigcirc \text{iknows}(k) \rightarrow$$

$$\exists k_d. \text{exec Decrypt}(k_d, k) \vee \exists k_d. \text{exec IntrDecrypt}(k_d, k)$$

Monotonicity axioms:

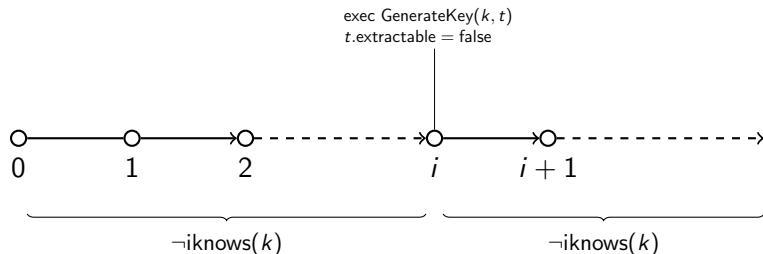
- ▶ $\text{iknows}(m) \rightarrow \square \text{iknows}(m)$
- ▶ $\text{tcontains}(k, t) \rightarrow \square \text{tcontains}(k, t)$

Secrecy of initially unextractable keys

“If a key is generated on the token with `extractable=false` then the key has always been and will always remain secret.”

exec `GenerateKey(k, t) ∧ (t.extractable = false)` \rightarrow $\Box \neg \text{iknows}(k) \wedge \square \neg \text{iknows}(k)$

Theorem: For all Cryptoki systems Γ , $\Gamma \models \phi$



Proof

1. Assume the contrary:

$\tau_0 : \text{exec GenerateKey}(k, t) \wedge t.\text{extractable} = \text{false}$

$\tau_I : \text{iknows}(k)$



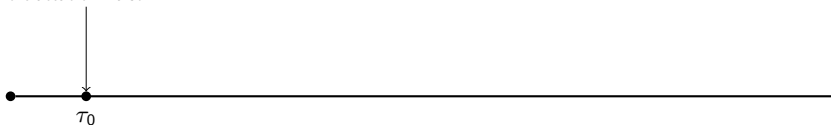
Proof

1. Assume the contrary:

$\tau_0 : \text{exec GenerateKey}(k, t) \wedge t.\text{extractable} = \text{false}$

$\tau_I : \text{iknows}(k)$

exec GenerateKey(k, t)
 $t.\text{extractable} = \text{false}$



Proof

1. Assume the contrary:

$\tau_0 : \text{exec GenerateKey}(k, t) \wedge t.\text{extractable} = \text{false}$

$\tau_1 : \text{iknows}(k)$

exec GenerateKey(k, t)
 $t.\text{extractable} = \text{false}$



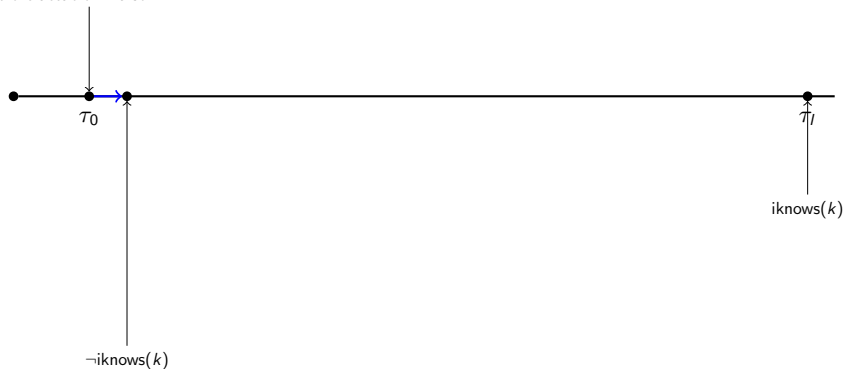
Proof

1. Assume the contrary:

$\tau_0 : \text{exec GenerateKey}(k, t) \wedge t.\text{extractable} = \text{false}$

$\tau_1 : \text{iknows}(k)$

exec GenerateKey(k, t)
 $t.\text{extractable} = \text{false}$

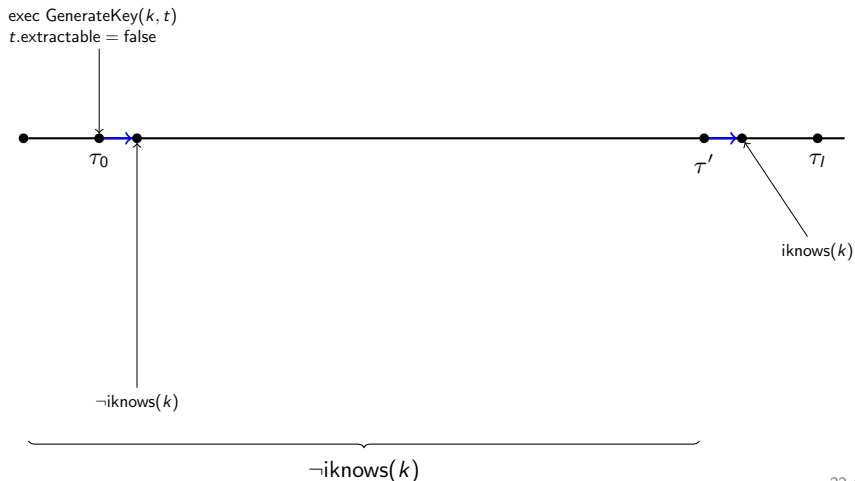


Proof

1. Assume the contrary:

$\tau_0 : \text{exec GenerateKey}(k, t) \wedge t.\text{extractable} = \text{false}$

$\tau_I : \text{iknows}(k)$



2. How has the intruder obtained k ?

ObtainKey: $\neg \text{iknows}(k) \wedge \bigcirc \text{iknows}(k) \rightarrow$
 $\exists k_d. \text{exec Decrypt}(k_d, k) \vee \exists k_d. \text{exec IntrDecrypt}(k_d, k)$

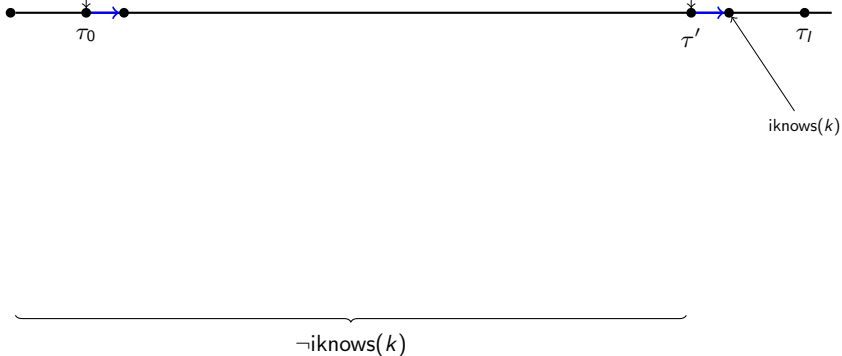


2. How has the intruder obtained k ?

ObtainKey: $\neg \text{iknows}(k) \wedge \bigcirc \text{iknows}(k) \rightarrow$
 $\exists k_d. \text{exec Decrypt}(k_d, k) \vee \exists k_d. \text{exec IntrDecrypt}(k_d, k)$

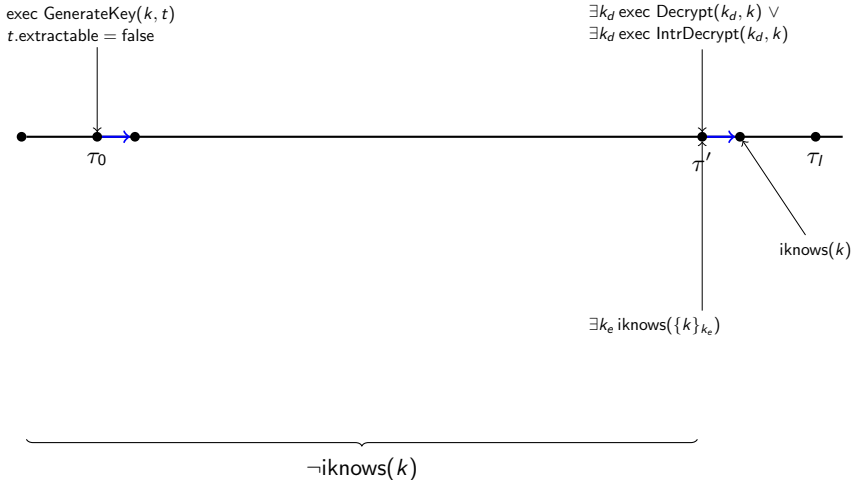
exec GenerateKey(k, t)
 $t.\text{extractable} = \text{false}$

$\exists k_d \text{exec Decrypt}(k_d, k) \vee$
 $\exists k_d \text{exec IntrDecrypt}(k_d, k)$



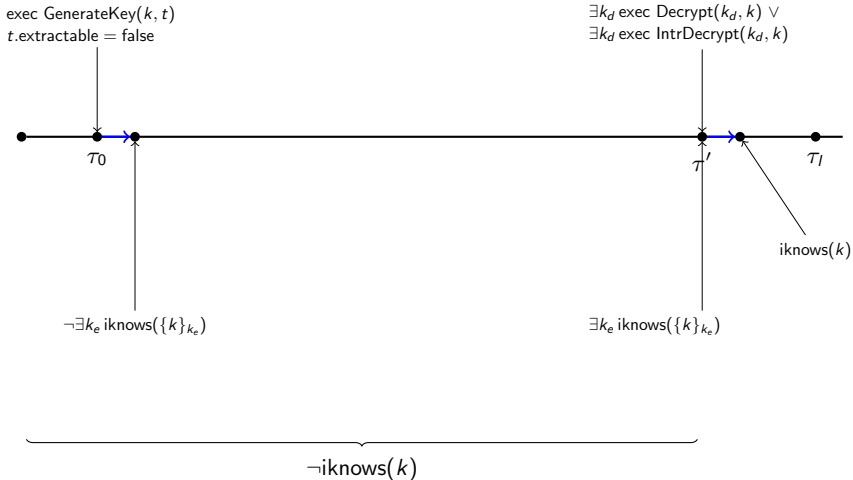
2. How has the intruder obtained k ?

ObtainKey: $\neg \text{iknows}(k) \wedge \bigcirc \text{iknows}(k) \rightarrow$
 $\exists k_d. \text{exec Decrypt}(k_d, k) \vee \exists k_d. \text{exec IntrDecrypt}(k_d, k)$



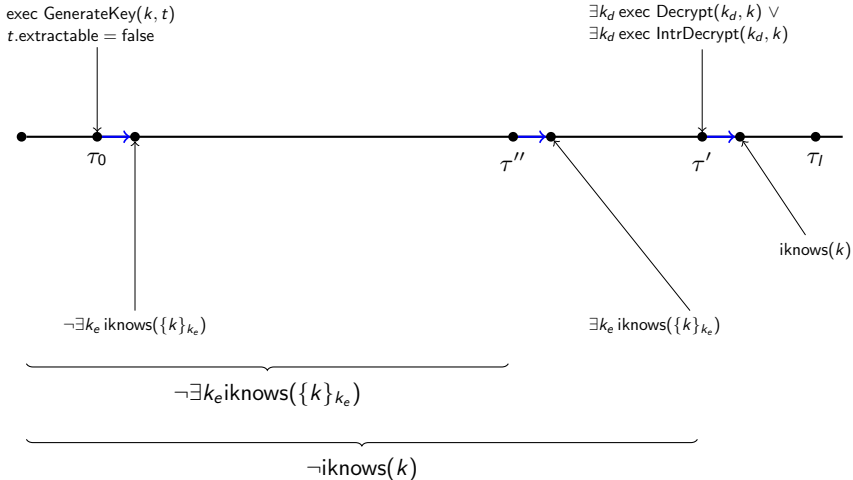
2. How has the intruder obtained k ?

ObtainKey: $\neg \text{iknows}(k) \wedge \bigcirc \text{iknows}(k) \rightarrow$
 $\exists k_d. \text{exec Decrypt}(k_d, k) \vee \exists k_d. \text{exec IntrDecrypt}(k_d, k)$



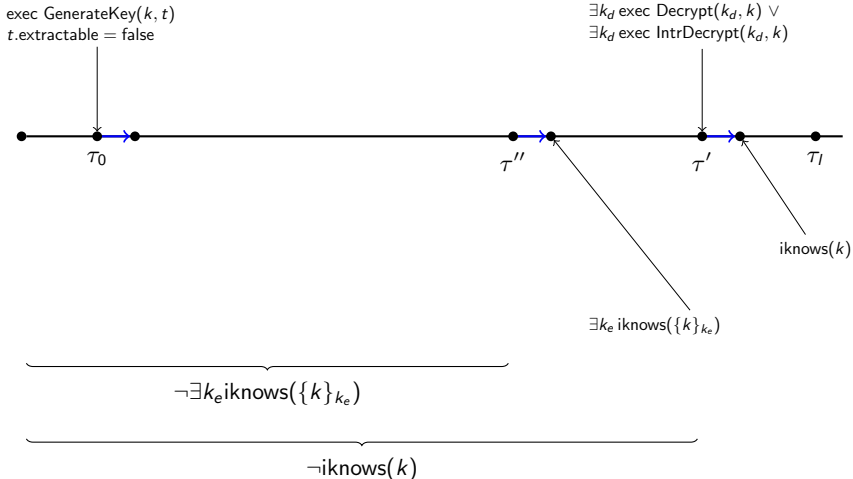
2. How has the intruder obtained k ?

ObtainKey: $\neg \text{iknows}(k) \wedge \bigcirc \text{iknows}(k) \rightarrow$
 $\exists k_d. \text{exec Decrypt}(k_d, k) \vee \exists k_d. \text{exec IntrDecrypt}(k_d, k)$



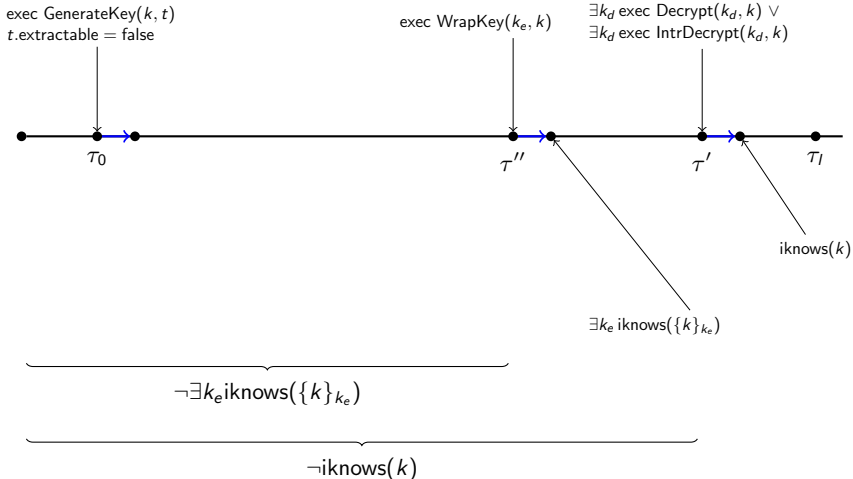
3. How has the intruder obtained the encryption?

$$\neg \text{iknows}(k) \wedge \neg \text{iknows}(\text{enc}(k_e, k)) \wedge \bigcirc \text{iknows}(\text{enc}(k_e, k)) \rightarrow \text{exec WrapKey}(k_e, k)$$



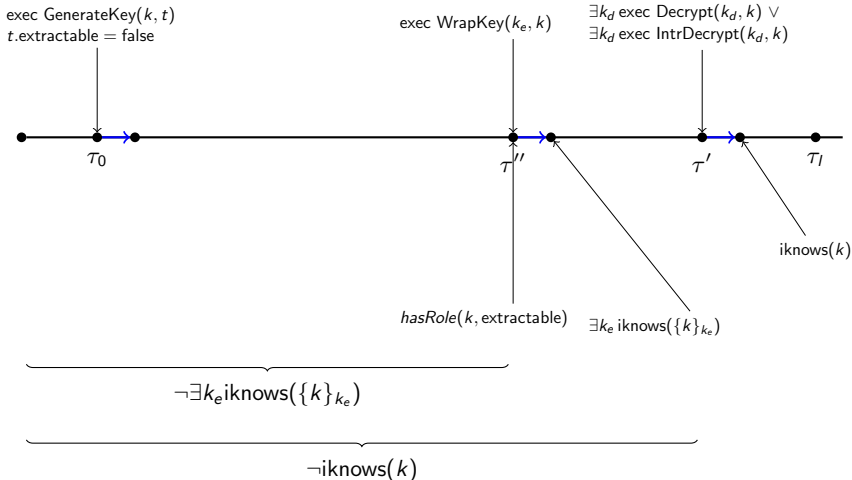
3. How has the intruder obtained the encryption?

$$\neg \text{iknows}(k) \wedge \neg \text{iknows}(\text{enc}(k_e, k)) \wedge \bigcirc \text{iknows}(\text{enc}(k_e, k)) \rightarrow \text{exec WrapKey}(k_e, k)$$

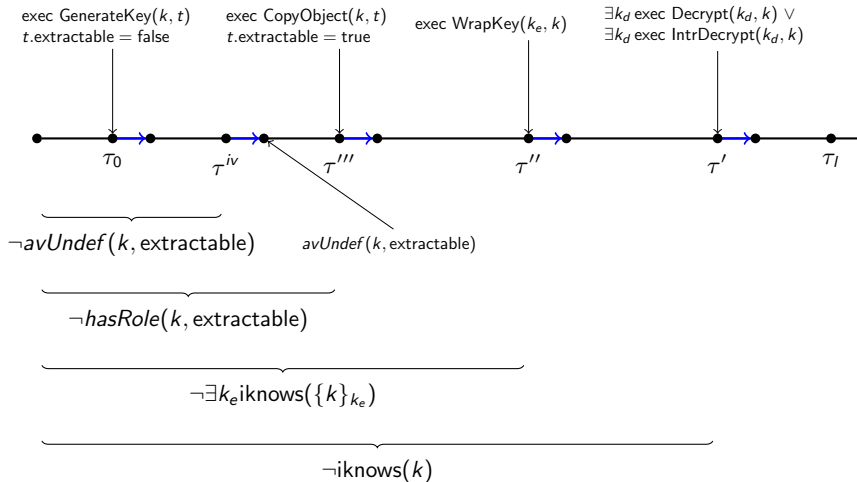


3. How has the intruder obtained the encryption?

$$\neg \text{iknows}(k) \wedge \neg \text{iknows}(\text{enc}(k_e, k)) \wedge \bigcirc \text{iknows}(\text{enc}(k_e, k)) \rightarrow \text{exec WrapKey}(k_e, k)$$



And so on ... until contradiction!



Last step shows a subtlety:

- ▶ Attributes can be undefined.
- ▶ It should not be possible to make a key object 'less defined' by unsetting a specified attribute to undefined.

$$\text{exec CopyObject}(k, t) \wedge t.a = \text{undefined} \rightarrow \text{avUndef}(k, a)$$

where

$$\text{avUndef}(k, a) \equiv \exists t. \text{tcontains}(k, t) \wedge t.a = \text{undefined}$$

- ▶ Otherwise attack possible!

Last step shows a subtlety:

- ▶ Attributes can be undefined.
- ▶ It should not be possible to make a key object 'less defined' by unsetting a specified attribute to undefined.

$$\text{exec CopyObject}(k, t) \wedge t.a = \text{undefined} \rightarrow \text{avUndef}(k, a)$$

where

$$\text{avUndef}(k, a) \equiv \exists t. \text{tcontains}(k, t) \wedge t.a = \text{undefined}$$

- ▶ Otherwise attack possible!

For robust configuration:

Attributes should always be given default values!

Formally ...

- ▶ Labelled formulas à la Gabbay's LDS approach

$\tau : F$ ' F is true at time point τ '

- ▶ Tableau proof system with special monotonicity rules

Final Remarks I

“Security APIs are like security protocols”

Final Remarks I

“Security APIs are like security protocols”

Only to a certain degree!

Some aspects our approach answers:

- ▶ Security APIs are typically generic.
Security is always up to the configuration!
- ▶ Global state, richer data structures, special features that cannot be modelled by protocol models.
- ▶ Security protocol model-checkers do not scale.

Paper: see FAST'10

Final Remarks II

Work in progress:

- ▶ Investigate `wrapWithTrusted` feature
- ▶ Notion of simulation to carry over results between different configurations.
- ▶ Move to untyped model (see Delaune et al. 08)
- ▶ Future: proof automation, algebraic properties, other APIs

How do configurations in general look like?

- ▶ This approach seems complementary to (Fröschle, Steel 09):
`Key-focused` versus `token-focused` configurations.
- ▶ `key-focused`: protect sensitive key objects from being wrapped and unwrapped 'arbitrarily'.
`extractable = false`, `wrapWithTrusted`
- ▶ `token-focused`: use robust attribute policy and a more secure version of `wrap/unwrap`