

# Bringing State-Separating Proofs to EasyCrypt

## A Security Proof for Cryptobox

François Dupressoir  
*(University of Bristol)*

Konrad Kohbrok  
*(Aalto University)*

Sabine Oechsner  
*(University of Edinburgh)*

CSF 2022

# Cryptographic security

Security against computationally bounded adversaries

- Initially studied for primitives and simple schemes
- Extended to larger protocols built from such primitives

Approaches to formal verification:

- Focused on protocols (e.g. Cryptoverif, Squirrel)
  - Rely on strong assumptions or manual reasoning about primitives
- Focused on primitives (e.g. Certicrypt, EasyCrypt, CryptHOL)
  - Don't scale well to larger constructions

Challenge: combining formal reasoning at different levels

# State-separating proofs (SSP) [BDFKK18]

SSP addresses this challenge on paper:

- Modular reasoning at different levels of abstraction
- Primitives and bigger constructions are treated in a uniform way

How to take advantage of SSP?

- Formalize SSP (SSProve [AHR+21])
- Incorporate SSP reasoning style in existing tool (EasyCrypt)
  - Maintain existing proof support from EasyCrypt
  - Add flexibility of proof style

# Our contributions

## Systematic mapping from SSP to EC concepts

- To take advantage of SSP reasoning style without actually formalising SSP
- SSP as guide for developing mechanized proofs

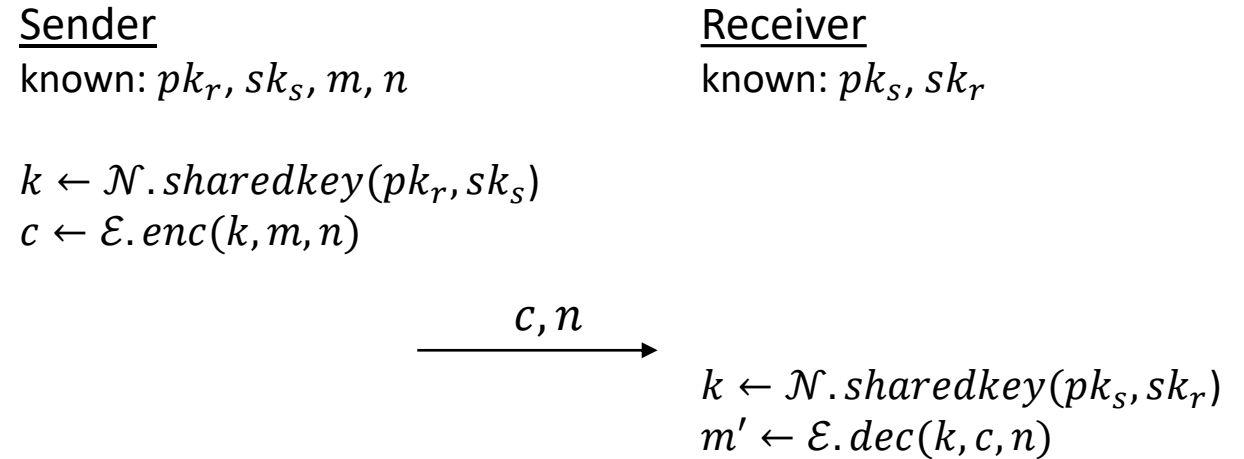
## Case study in EasyCrypt: Cryptobox

- PKAE security of multiple concurrent instances in the presence of corruption
- First mechanized security proof of Cryptobox

# Cryptobox

# Cryptobox

- Default way of achieving "secure encryption from public keys", e.g. implemented in NaCl
- Cryptobox = NIKE + AE [DH76]



Here: Cryptobox security as nonce-based PKAE scheme

- Multi-instance setting with corruption
- Reduce to standard assumptions on NIKE and AE schemes

# State-separating proofs (SSP)

# Code-based game-playing [BR04/06]

Game = set of oracles with shared state

Security as distinguishing advantage of adversary with oracle access

For all PPT  $\mathcal{A}$ ,  $|\Pr[\mathcal{A}^{GPKAE_{\mathcal{P}}^0} = 1] - \Pr[\mathcal{A}^{GPKAE_{\mathcal{P}}^1} = 1]|$  neglig.

Security proof: sequence of game hops  
e.g. from  $GPKAE_{\mathcal{P}}^0$  to  $GPKAE_{\mathcal{P}}^1$

Game  $GPKAE_{\mathcal{P}}^b$ :

State: PK, SK, ...

GEN() :

... // gen. honest key pair

CSETPK(pk) :

... // register corrupt pk

PKENC(pks, pkr, m, n) :

... // behaviour depends on  $b$

PKDEC(pkr, pks, c, n) :

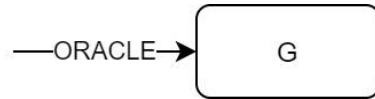
... // behaviour depends on  $b$



# State-separating proofs (SSP) [BDFKK18]

Structure code-based games to achieve modularity and composition

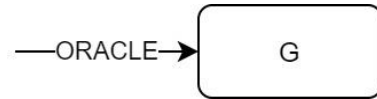
- Game = set of oracles with state



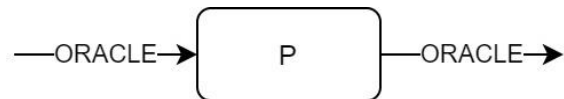
# State-separating proofs (SSP) [BDFKK18]

Structure code-based games to achieve modularity and composition

- Game = set of oracles with state



- **Package** = set of oracles with state that can call oracles



- State separation between packages
- Package composition (associative)

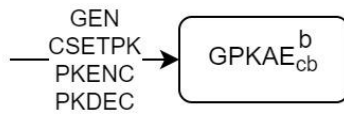


- Adversary: special package



# Cryptobox and SSP

Games  $GPKAE_{\mathcal{P}}^b$  for  $\mathcal{P} = \text{Cryptobox}$



Sender

known:  $pk_r, sk_s, m, n$

$k \leftarrow \mathcal{N}.sharedkey(pk_r, sk_s)$

$c \leftarrow \mathcal{E}.enc(k, m, n)$

Receiver

known:  $pk_s, sk_r$

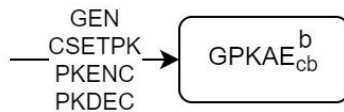
$c, n$

$k \leftarrow \mathcal{N}.sharedkey(pk_s, sk_r)$

$m' \leftarrow \mathcal{E}.dec(k, c, n)$

# Cryptobox and SSP

Games  $GPKAE_{\mathcal{P}}^b$  for  $\mathcal{P} = \text{Cryptobox}$



Sender

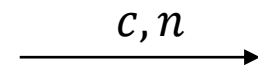
known:  $pk_r, sk_s, m, n$

$k \leftarrow \mathcal{N}.sharedkey(pk_r, sk_s)$

$c \leftarrow \mathcal{E}.enc(k, m, n)$

Receiver

known:  $pk_s, sk_r$

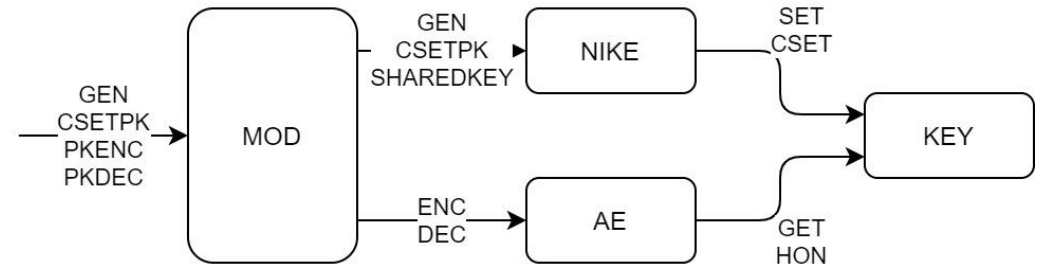


$k \leftarrow \mathcal{N}.sharedkey(pk_s, sk_r)$

$m' \leftarrow \mathcal{E}.dec(k, c, n)$

Alternative representation of PKAE game for proof

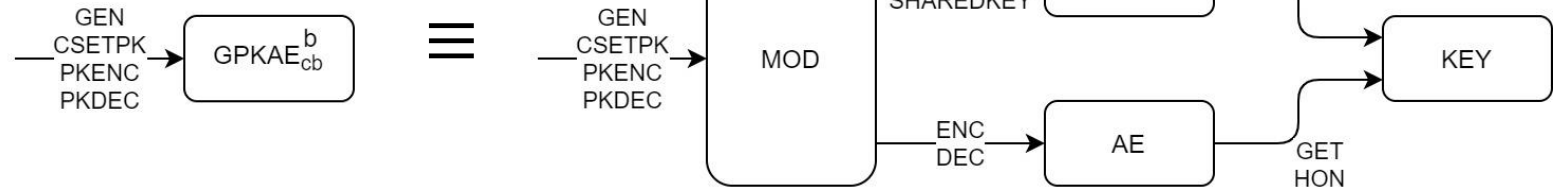
- Highlights (in)dependencies between building blocks: NIKE and AE
- Shared state: shared symmetric key



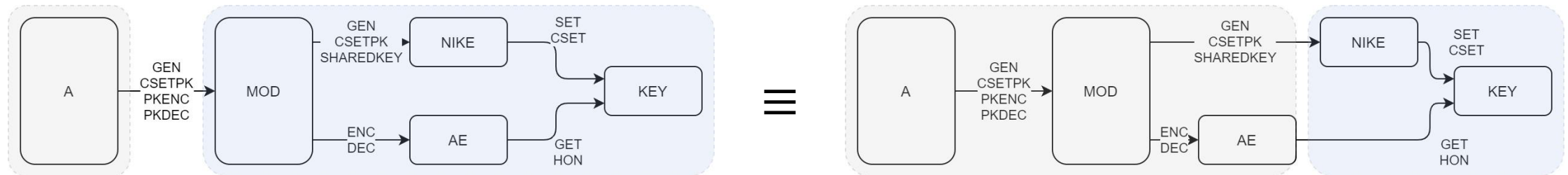
# Cryptobox and SSP

Types of proof steps:

- Code equivalence



- Reductions as shifting package boundaries
  - e.g. turn PKAE adversary A into NIKE adversary B



$A \rightarrow \text{GPKAE}$

$B \rightarrow \text{GNIKE}$

# Mapping SSPs to EasyCrypt

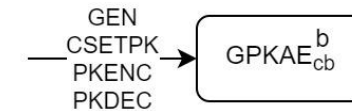
# EasyCrypt

- Interactive proof assistant for reasoning about probabilistic programs
- Module system
  - Modular reductions
  - Modular construction of adversaries

Goal: recreate SSP reasoning style

# Mapping SSPs to EasyCrypt

SSP	EasyCrypt
Package	Module



```
module type GPKE = {... // proc. }
```

```
module GPKE0 (P : PKAE) = {
  ...           // procedures
}
```

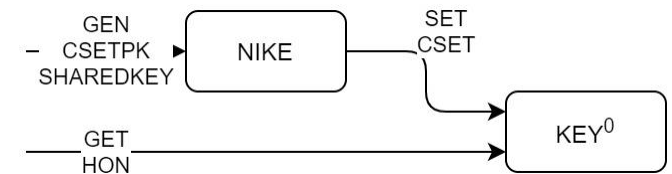


```
module NIKE (K : NIKE_in) = {
  proc gen() = {...}
  proc csetpk(pk : pkey) = {...}
  proc sharedkey(pk:pkey,sk:skey)
    = {...} // may query K's procedures
}
```



# Mapping SSPs to EasyCrypt

SSP	EasyCrypt
Package	Module
Package composition	Module parameter (sequential), wiring module (parallel)

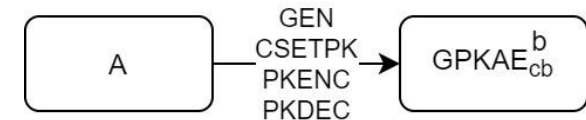


```
module GNIKE (N : NIKE)
  (K : KEY) = {
    include N(K) [gen, csetpk, sharedkey]
    include K[get, hon]
  }
```

```
module GNIKE0 = GNIKE(NIKE, K0)
```

# Mapping SSPs to EasyCrypt

SSP	EasyCrypt
Package	Module
Package composition	Module parameter (sequential), wiring module (parallel)
Game	Fully instantiated module
Adversary	Abstract module



```

module type A_pkae (G : GPKAE) = {
  proc run() : bool
}
  
```

Distinguishing advantage of  $A : A\_pkae$ :

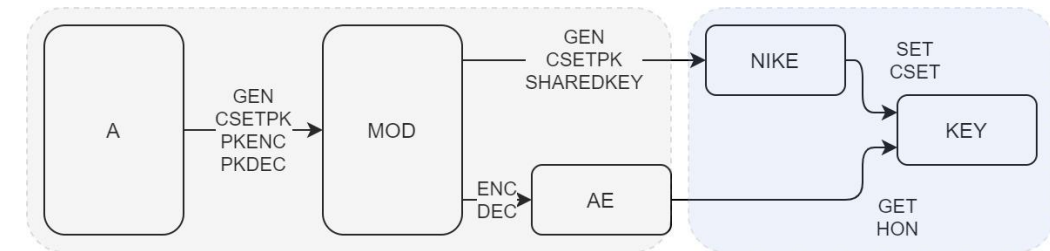
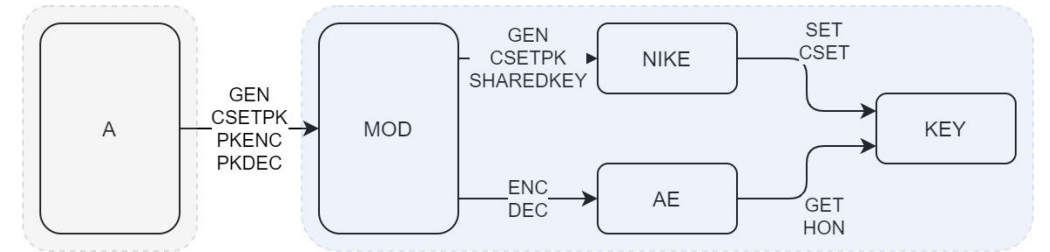
$$|\Pr[ A(GPKAE0).run() @m : res ] - \Pr[ A(GPKAE1).run() @m : res ]|$$

# Mapping SSPs to EasyCrypt

<b>SSP</b>	<b>EasyCrypt</b>
Package	Module
Package composition	Module parameter (sequential), wiring module (parallel)
Game	Fully instantiated module
Adversary	Abstract module
Implicit initialization	?
Implicit state separation	?

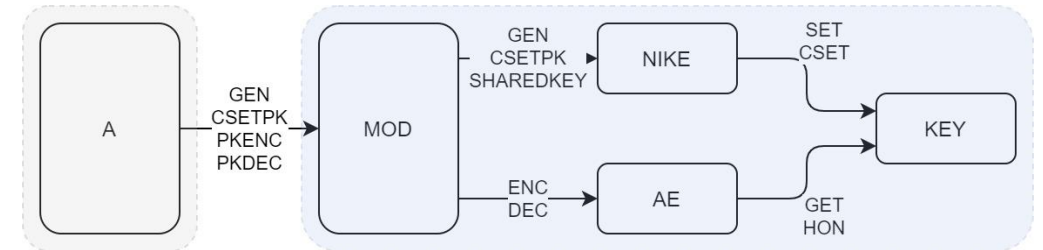
# Mapping SSPs to EasyCrypt

SSP	EasyCrypt
Package	Module
Package composition	Module parameter (sequential), wiring module (parallel)
Game	Fully instantiated module
Adversary	Abstract module
Implicit initialization	?
Implicit state separation	?
Shifting package boundaries	Redefining module boundaries



# Mapping SSPs to EasyCrypt

SSP	EasyCrypt
Package	Module
Package composition	Module parameter (sequential), wiring module (parallel)
Game	Fully instantiated module
Adversary	Abstract module
Implicit initialization	?
Implicit state separation	?
Shifting package boundaries	Redefining module boundaries



```

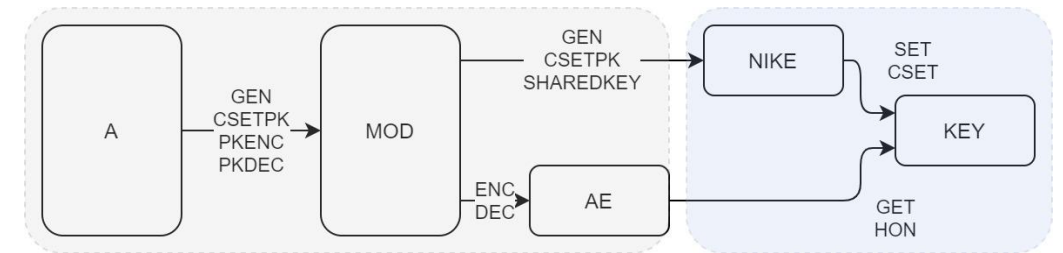
module GMOD (N : NIKE)
  (AE : AE) : GPKAE = {...}

module GMOD0 = GMOD(NIKE(K0), AE(K0))

consider A(GMOD0)
  
```

# Mapping SSPs to EasyCrypt

SSP	EasyCrypt
Package	Module
Package composition	Module parameter (sequential), wiring module (parallel)
Game	Fully instantiated module
Adversary	Abstract module
Implicit initialization	?
Implicit state separation	?
Shifting package boundaries	Redefining module boundaries



instantiate  $G_{MOD}$  only partially:

```
module R (G : GNIKE) = GMOD(G, AE(K0))
```

then instantiate with a NIKE game

```
consider A ( R (GNIKE0) )
```

```
NIKE adversary B (G : GNIKE) = A(R(G))
```

# Mapping SSPs to EasyCrypt

<b>SSP</b>	<b>EasyCrypt</b>
Package	Module
Package composition	Module parameter (sequential), wiring module (parallel)
Game	Fully instantiated module
Adversary	Abstract module
Implicit initialization	?
Implicit state separation	?
Shifting package boundaries	Redefining module boundaries

EasyCrypt: Designed for Halevi's style of code-based game-playing proofs [Halevi05]

- Experiment that initializes state, provides oracles and calls adversary
- Experiment can initialize state of other modules

# Mapping SSPs to EasyCrypt

SSP	EasyCrypt
Package	Module
Package composition	Module parameter (sequential), wiring module (parallel)
Game	Fully instantiated module
Adversary	Abstract module
Implicit initialization	Explicit initialization or restrict initial memories
Implicit state separation	
Shifting package boundaries	Redefining module boundaries

Options:

- Explicit initialization: composition issues
- Restricting initial memories: seems better for now

$$\forall m, \text{GPKAE0.PK}\{m\} = \text{empty} \wedge \dots \Rightarrow$$
$$|\text{Pr}[A(\text{GPKAE0}).\text{run}() @m : \text{res}]$$
$$- \text{Pr}[A(\text{GPKAE1}).\text{run}() @m : \text{res}]|$$

(but: explicit initialization sometimes needed)



# Mapping SSPs to EasyCrypt

SSP	EasyCrypt
Package	Module
Package composition	Module parameter (sequential), wiring module (parallel)
Game	Fully instantiated module
Adversary	Abstract module
Implicit initialization	Explicit initialization or restrict initial memories
Implicit state separation	Explicit state separation
Shifting package boundaries	Redefining module boundaries

Explicitly require separate memories of modules

$$\begin{aligned} & \forall (A <: \text{ApkAE}\{GPKAE0, GPKAE1\}) \\ & \forall m, GPKAE0.PK\{m\} = \text{empty} \wedge \dots \Rightarrow \\ & |\text{Pr}[ A(GPKAE0).run() @m : res ] \\ & \quad - \text{Pr}[ A(GPKAE1).run() @m : res ]| \end{aligned}$$

# Cryptobox security in EasyCrypt

# Cryptobox security proof

- PKAE multi-instance security notion with corruption
  - Extension of [An01] PKAE security from two honest parties to many parties with corruption
  - Variant of [BT16] AE security notion that adds corruption
- Cryptobox implementation
- Security of Cryptobox is reduced to (single-instance) security of NIKE and AE schemes

Not mentioned yet: statistical equivalence steps

- Bound probability of public key collisions
  - Cryptobox provides plausible deniability: no ID related to public keys

What have we learned?

# Benefits of our approach

## Why SSP?

- Local reasoning
- Reasoning at different levels of abstraction

## Why EasyCrypt?

- Existing support from EasyCrypt: libraries, tactics, automation etc.
- Flexibility: switching between reasoning styles

## SSP as tool for proof development

- Proof discipline
- Sketching proof outline (dependencies, shared state, ...)
- Visualizing proof steps

# Potential for improvement

## State initialization

## Assertions

- SSP: to enforce "good" adversary behaviour
- EasyCrypt: modeled as explicit control flow

## Forward reasoning

- SSP: oracles close to the real-world interface of primitive
- In oracles: same oracle for honest and corrupt queries, distinguish through control flow
- In proof: case analysis and simplify execution path
- Forces forward reasoning when proving program equivalences, which goes against tool design
- Assertion modeling as control flow causes similar issue

# Conclusion

# Exploring SSPs to guide larger formalizations

## Map SSP concepts to EasyCrypt constructs

- Informal yet systematic
- Preserves key SSP features (local and modular reasoning)
- Can be combined with traditional reasoning style
- Identified friction points and potential for improvement

## New example: Cryptobox

- PKAE security proof in multi-instance setting with corruption

eprint: <https://ia.cr/2021/326>

Code: <https://gitlab.com/fdupress/ec-cryptobox/>