White-Box and Asymmetrically Hard Crypto Design

Alex Biryukov

University of Luxembourg

18-May-2019

SNT
securityandtrust.lu

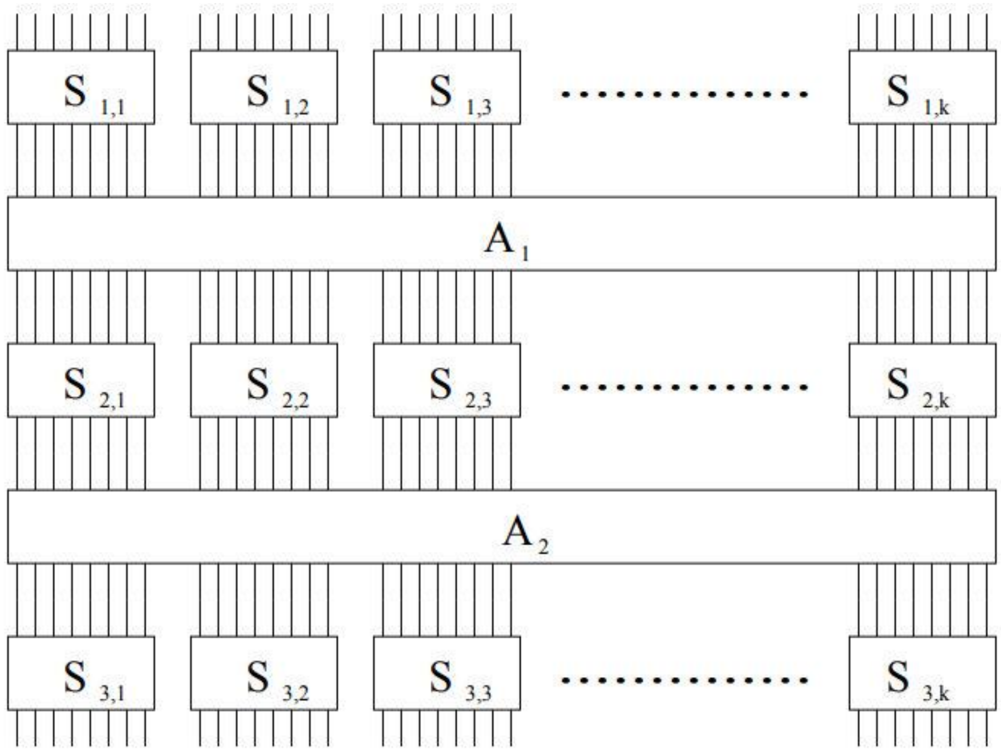
slides from Whibox'19 workshop

UNIVERSITÉ DU LUXEMBOURG

Plan of the talk

- The ASASA story
- Resource Hardness Framework
- Other ideas

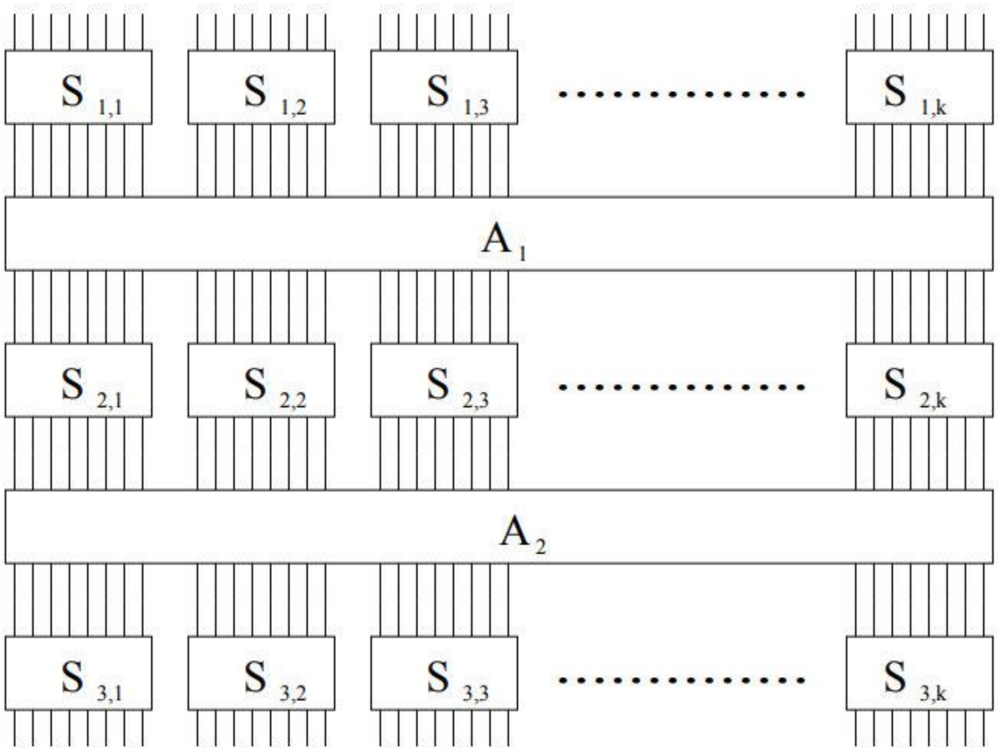
Structural cryptanalysis of SASAS*



- Scheme with unknown keyed S-boxes and Affine mappings
- For 128-bit block, 8-bit S-boxes, secret key-size is 2^{17} bits

*Biryukov, Shamir, Structural Cryptanalysis of SASAS, Eurocrypt'2001

Structural cryptanalysis of SASAS*



- For 128-bit block, 8-bit S-boxes, secret key-size is 2^{17} bits
- **Multiset attack** complexity is 2^{16} chosen texts and 2^{28} time

*Biryukov, Shamir, Structural Cryptanalysis of SASAS, Eurocrypt'2001

Structural cryptanalysis of SASAS

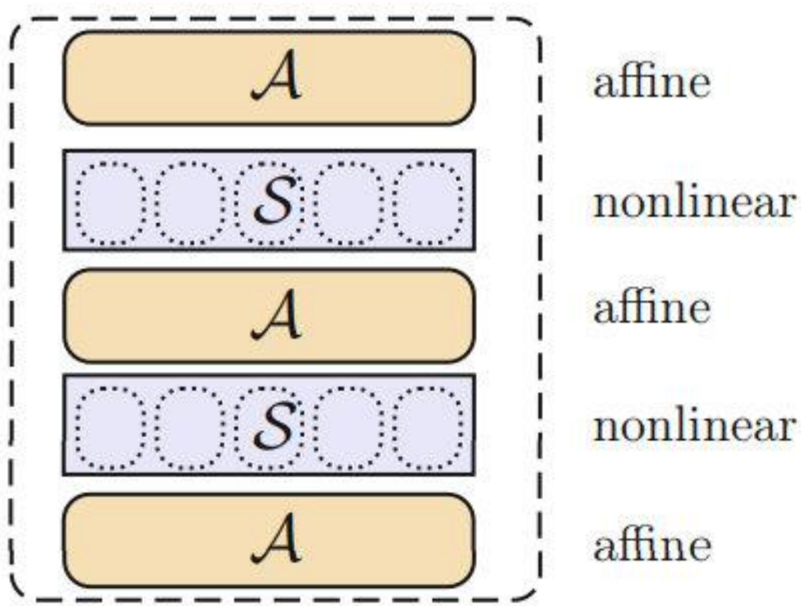
- What this has to do with WBC?

Structural cryptanalysis of SASAS

- Many early obfuscations were broken because SASAS and shorter ciphers are structurally very weak (and simple ASA was used in many WBC schemes)
- Strong diffusion in ciphers prevents from building tables with more rounds since lookup tables explode

The ASASA attempt*

- One scheme we couldn't break in 2001 was ASASA (with bijective S-boxes)
- (ASASA with non-bij. S-boxes was proposed as PK scheme by PatarinGoubin'97 and broken by Ding-Feng'99, Biham'00)



*Biryukov, Bouillaguet,Khovratovich, Cryptographic Schemes based on ASASA.., AC'2014

The ASASA attempt*

- Defined strong and weak white box crypto in [BBK'14] a la [Wyseur'09]

(Strong WBC=PK, i.e. no ability to decrypt, was the main goal of the paper, also now called one-wayness (OW))

- Built strong and weak WBC from ASASA
- Strong WBC was based on multivariate crypto, expanding S-boxes+noise

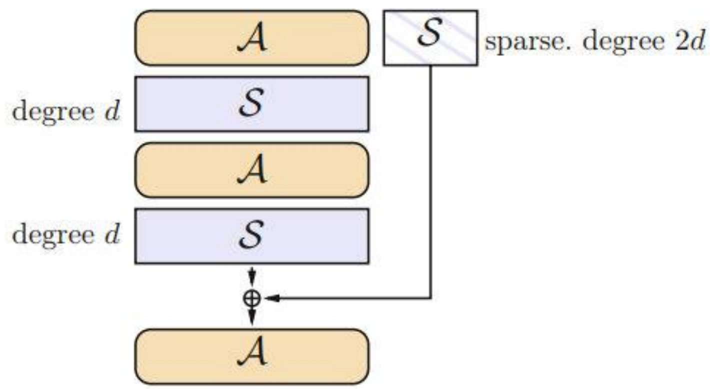


Fig. 2. Small perturbations to defeat decomposition attacks as injection of sparse high-degree polynomials

*Biryukov, Bouillaguet,Khovratovich, Cryptographic Schemes based on ASASA.., AC'2014

The ASASA attempt*

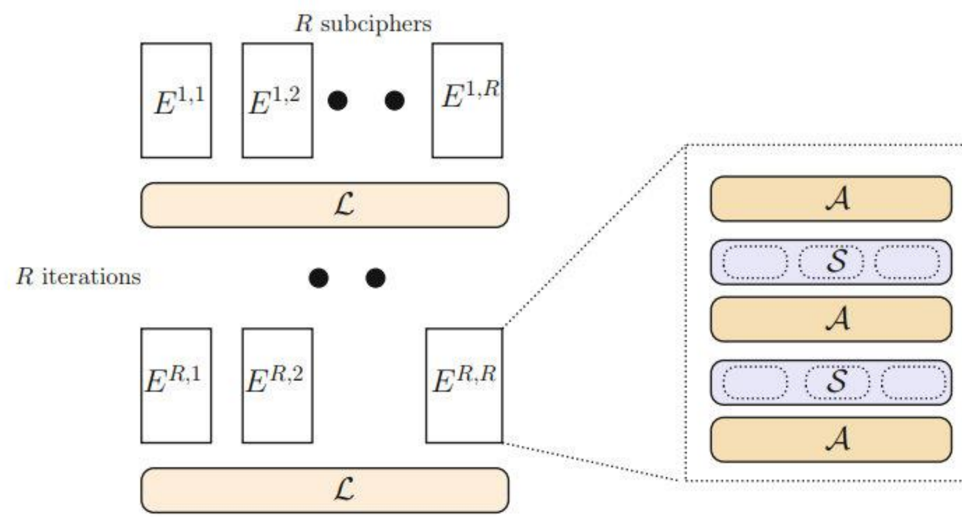
- Built strong and weak WBC from ASASA
- Strong WBC was based on multivariate crypto, expanding S-boxes+noise
- Strong and some weak WBC broken in 3 nice cryptanalytic papers [GPT'15,DDKL'15,MDFK'15]

*Biryukov, Bouillaguet,Khovratovich, Cryptographic Schemes based on ASASA.., AC'2014

The ASASA attempt

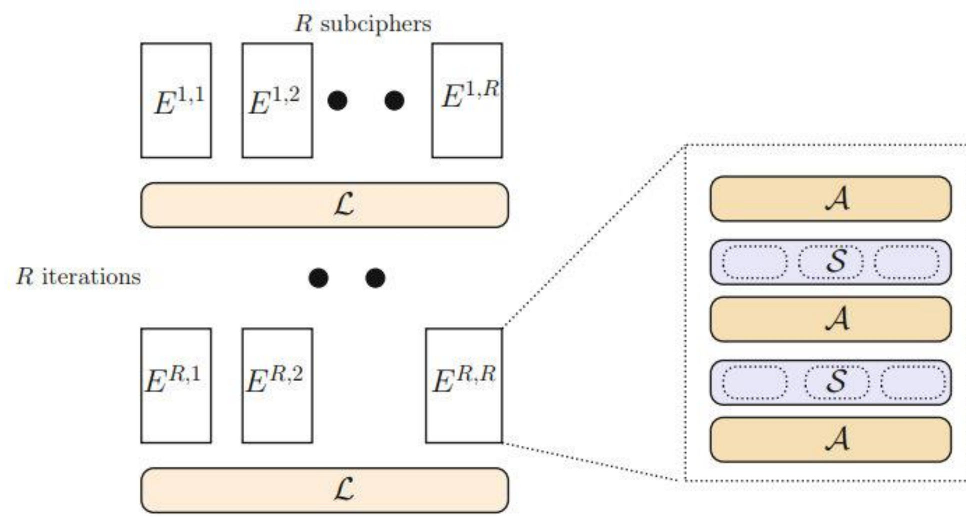
A few more details on our weak WBC scheme

- SPN, recursive approach, assuming ASASA or ASASASA mini-ciphers are secure against decomposition



The ASASA attempt

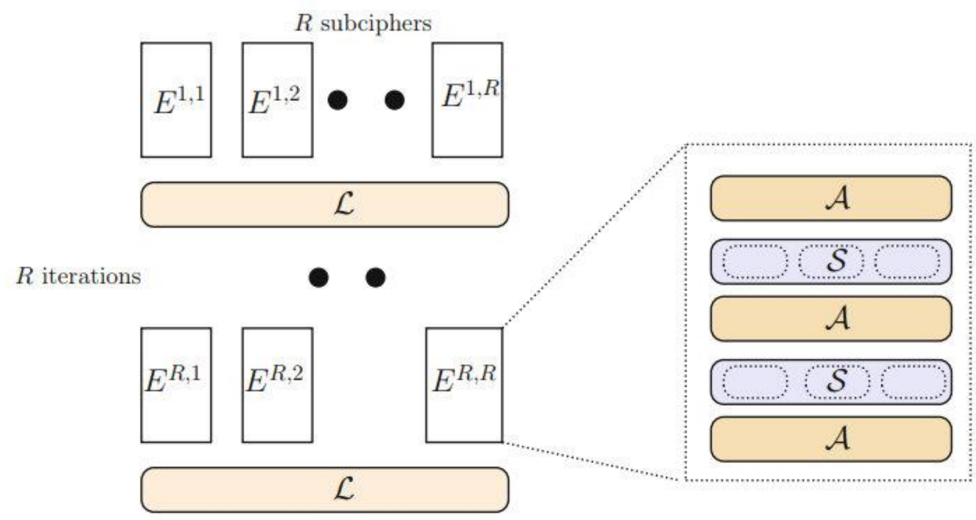
- ASASASA instances still unbroken
- Overall approach is valid, just needs more rounds *r*, description size grows linearly with *r*.



The ASASA attempt

- ASASASA instances still unbroken
- Overall approach is valid, just needs more rounds.
- Motivated more reseach on weak-WBC and nice constructions

  SPACE [BI15], PuppyCipher [FKKM16], SPNBox [BIT16]



Weak white-box

- "We note that a white-box implementation can be useful as it forces the user to use the software at hand“, -Marc Joye’08

Weak white-box

- Incompressibility ≈ Space-hardness ≈ Code-hardness
- Generalize: Resource *R*-hardness

Force to use implementation with special properties:

- *Inefficient* in resource *R*
- Password-protected (access control)
- Tagged/watermarked (tracing)

Resource Hardness Framework*

Efficiency metrics for crypto algorithms:

- Speed (Time complexity, parallel or sequential)
- Code-size (ROM)
- Memory complexity (RAM)

Sometimes *inefficiency* of algorithms in these metrics is required

*Biryukov, Perrin, "Symmetrically and Asymmetrically Hard Cryptography, Asiacrypt'17

Resource Hardness Framework

Sometimes inefficiency of crypto algorithms in these metrics is required (*several research areas that do not always talk to each other*)

- Weak whitebox-crypto (code size hardness)
- Password hashing (memory hardness)
- Key derivation functions (KDF) (time hardness)
- Big key encryption (code size hardness)
- Time-lock puzzles, PoSW, VDFs (sequential time hardness)
- Proof-of-X (all kinds of hardness)

Resource Hardness Framework

Symmetric vs *Asymmetric* Resource hardness:

- Symmetric – computation is *R* hard for all the users
- Asymmetric – computation is easy for "privileged" users knowing the secret *K*

Resource Hardness Framework

	Time	Memory	Code size
Applications	KDF, time-lock	Password hashing, egalitarian computing	White-box crypto, big-key encryption
Symmetrically hard functions	PBKDF2 [Kal00]	Argon2 [BDK16], Balloon [BCGS16]	XKEY2 [BKR16], WHALE (Sec. 5.2)
Asymmetrically hard functions	RSA-lock [RSW96], SKIPPER (Sec. 5.1)	DIODON (Sec. 2.4.3)	White-box block ciphers [BBK14, BI15, FKKM16] [BIT16]

Table 1: Six types of hardness and their applications.

Resource Hardness Framework

Definition 2 (R-hardness). *We say that a function $f : \mathcal{X} \to \mathcal{Y}$ is* R-hard against 2^p-adversaries *for some tuple* $\mathsf{R} = (\rho, u, \epsilon(p))$ *with* $\rho \in \{\mathit{Time}, \mathit{Code}, \mathit{RAM}\}$ *if evaluating the function f using less than u units of the resource ρ and at most 2^p units of storage is possible only with probability $\epsilon(p)$. More formally, the probability for a 2^p-adversary to win the* efficient approximation game, *which is described below, must be upper-bounded by $\epsilon(p)$.*

1. *The challenger chooses a function f from a predefined set of functions requiring more than u units of ρ to be evaluated.*

2. *The challenger sends f to the adversary.*

3. *The adversary computes an approximation f' of f which, unlike f, can be computed using less than* u *units of the resource ρ.*

4. *The challenger picks an input x of $\mathcal{X}$ uniformly at random and sends it to the adversary.*

5. *The adversary wins if $f'(x) = f(x)$.*

*Generalized from definition of incompressibility from [FKKM16]

Resource Hardness Framework

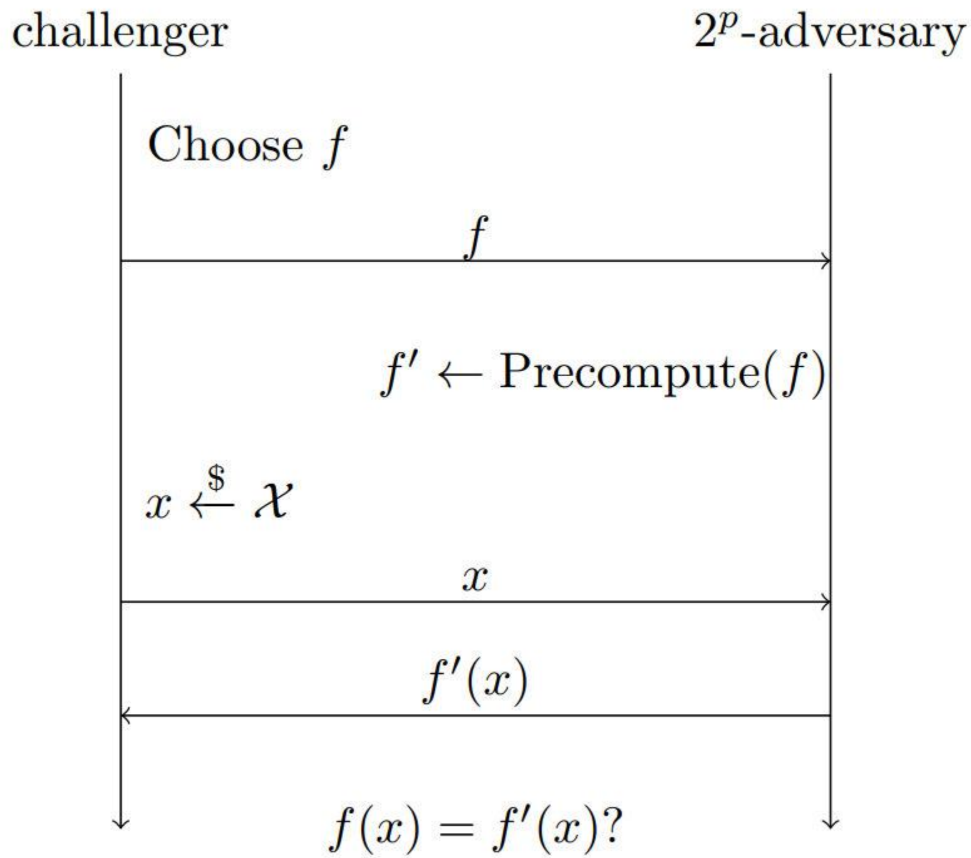


Figure 2: The game corresponding to the definition of $(\rho, u, \epsilon(p))$-hardness against 2^p-adversaries.

Resource Hardness Framework

- How to achieve required *R*-hardness?
- The framework allows us to construct primitives with any hardness type:

the idea of *plugs* with specific hardness type

Plugs: Time-Hardness

Symmetric:

- IterHash (t,n) – iterates t-bit hash n times ($n < 2^{t/2}$ to avoid cycles)

Asymmetric

- RSAlock(t,n) (time-lock) *n* squarings mod N, N=pq ≈ 2^t

$$\mathrm{RSAlock}_n^t(x) = x^{2^n} \mod N$$

Secret owner first computes $e=2^n \bmod (p-1)(q-1)$

Then he computes $x^e \bmod N$ (or CRT)

Plugs: Code-Hardness

Symmetric:

- BigLUT (t,v) – a table with 2^t random *v*-bit entries

Asymmetric

- BcCounter(t,v) = $E_k(0^{v-t}||x)$, E_k is a *v*-bit block cipher with secret key *k*, $|k| \geq v$

  Secret owner knows *k*

Hardness for the common user:

$$(\mathrm{Code}, 2^p, 2^{p-t})\text{-hard}$$

Plugs: Code-Hardness

Symmetric:

- BigLUT (t,v) – a table with 2^t random v-bit entries

Asymmetric

- BcCounter(t,v) = $E_k(0^{v-t}||x)$, E_k is a *v*-bit block cipher with secret key *k,* $|k| \geq v$, $|x|=t$, $t < v$

  Secret owner knows *k*

Improvement for small *t:* (parallel application of *l* tables |x| = v)

$$f(x_0||...||x_{\ell-1}) \;=\; \oplus_{i=0}^{\ell-1} E_k(\mathrm{byte}(i)||0^{n-t-8}||x_i)$$

Hardness for the common user:

$$\big(\mathrm{Code}, 2^p, \max(2^{p-v}, (2^{p-t}/\ell)^\ell)\big)\text{-hard.}$$

Plugs: Memory-Hardness

Symmetric:

- Argon2(t,M) with input size *t* and memory size *M* (memory hard password hashing function)

Asymmetric

- Diodon (more details later)

Our collection of *R*-hard plugs

Hardness	Symmetric	Asymmetric
Time	$\mathrm{IterHash}_{\eta}^{t}$ $(\mathsf{Time}, \eta, 2^{p-t})$	$\mathrm{RSAlock}_{\eta}^{t}$ $(\mathsf{Time}, \eta, 2^{p-t})$
Memory	Argon2 $(\mathsf{RAM}, M/5, 2^{p-t})$	Diodon $(\mathsf{RAM}, M/10, 2^{p-t})$
Code	BigLUT_{v}^{t} $(\mathsf{Code}, 2^{p}, 2^{p-t})$	$\mathrm{BcCounter}_{v}^{t}$ $(\mathsf{Code}, 2^{p}, 2^{p-t})$

Table 2: Possible plugs, i.e. sub-components for our constructions which we assume to be R-hard against 2^p-adversaries.

Modes of Plug Usage

The plugs can be used in different modes

- Plug-then-randomize (PTR)
- Hard block cipher mode (HBC)
- Hard sponge mode (HSp)

Mode: Plug-then-Randomize

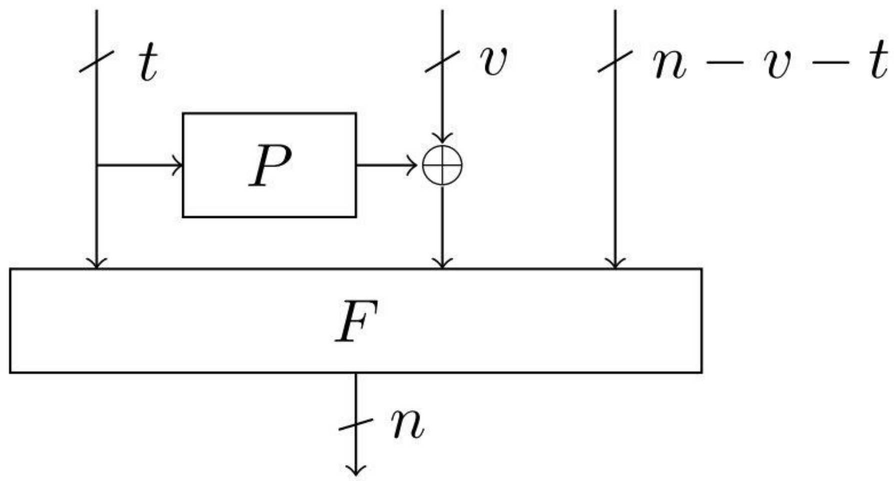


Figure 3: Evaluating the plugged function $(F \cdot P)$

Here F is a random (permutation) oracle

Iterate to increase hardness:

$(\rho, u, \max(\epsilon(p)^r, 2^{p-n}))$*-hard against* 2^p*-adversaries*

Mode: Hard block cipher

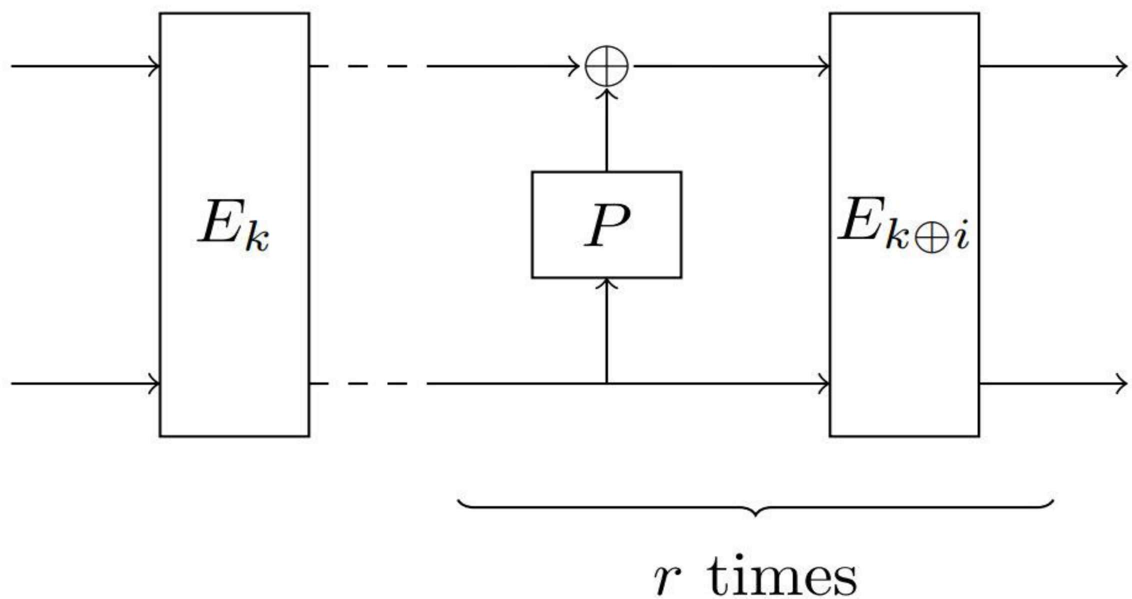


Figure 4: The HBC block cipher mode.

- Given related-key-secure n-bit block cipher E_k, $k \geq n$

$(\rho, u, \max(\epsilon(p)^r, 2^{p-n}))$-*hard against* 2^p-*adversaries*

Example: Time-hard block cipher *Skipper*

Algorithm 5 SKIPPER encryption

Inputs: n-bit plaintext x; k-bit key k; RSA modulus N

Output: n-bit ciphertext y

$y \leftarrow \mathrm{AES}_k(x)$

for all $i \in \{1, 2\}$ **do**

  $y_1 \mid\mid y_2 \leftarrow y$, where $|y_1| = 88$ and $|y_2| = 40$

  $y_2 \leftarrow y_2 \oplus T_{40}(y_1^{2^\eta} \mod N)$

  $y \leftarrow \mathrm{AES}_{k \oplus i}(y_1 || y_2)$

end for

return y

- The plug is: $(\mathrm{Time}, \eta, 2^{-40})$-hard   *Skipper* is: $\left(\mathrm{Time}, \eta, \max\left(2^{48-128}, (2^{-40})^2\right)\right)$-hard

Hard Sponge Mode (HSp)

- Sponges can be used to construct hash functions, stream ciphers, MACs and AE

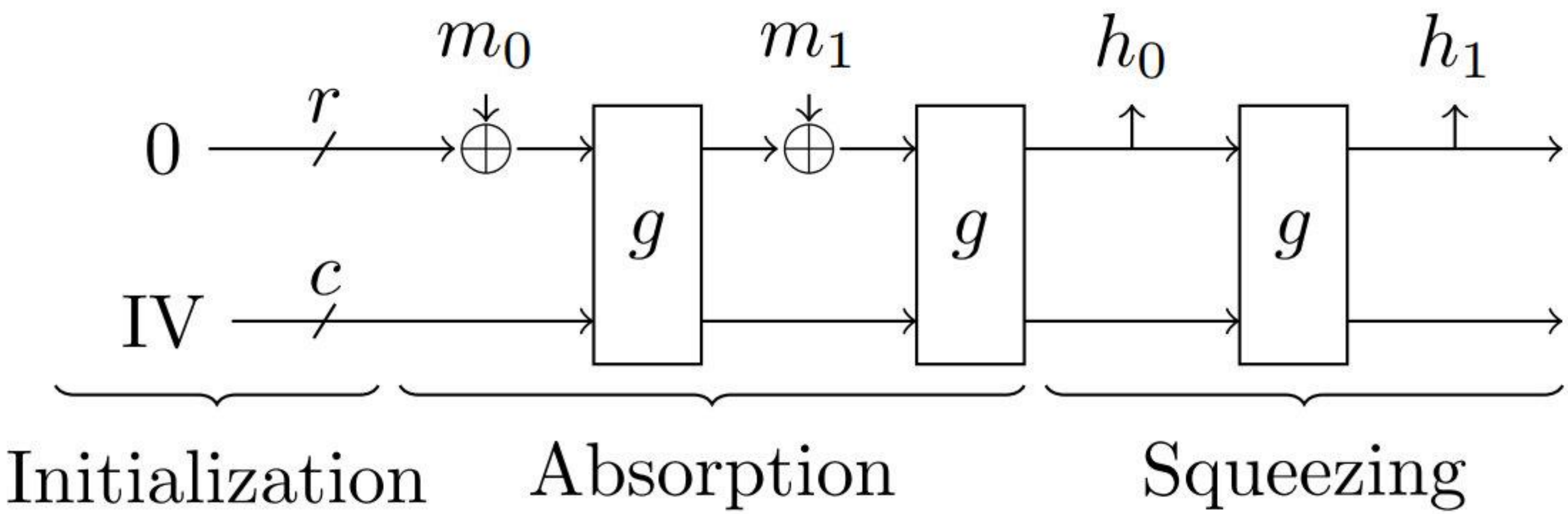
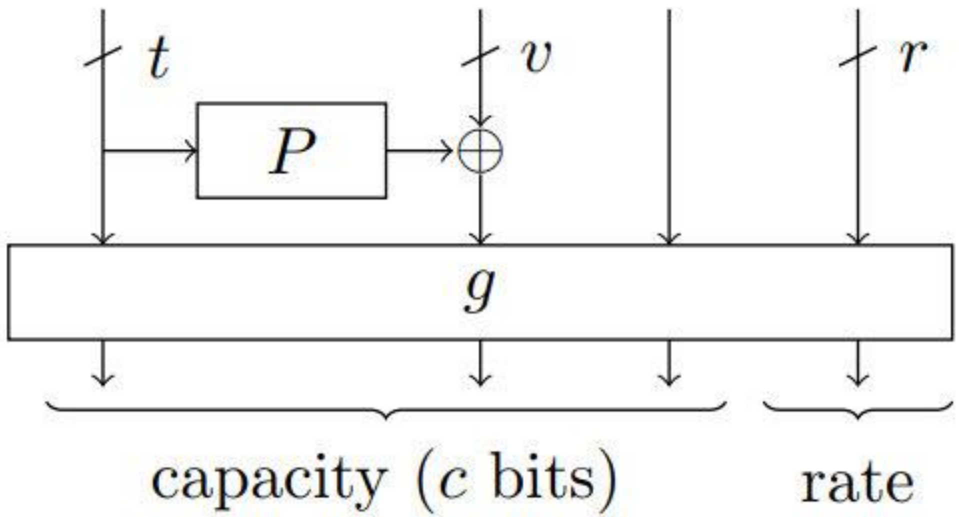


Figure 5: A sponge-based hash function.

Hard Sponge Mode (HSp)

- Iteratively use Plug-then-Randomize mode



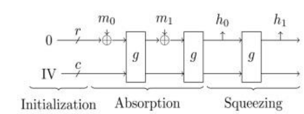


Figure 5: A sponge-based hash function.

- In the paper: Code-hard hash function based on Keccak which we called Whale.

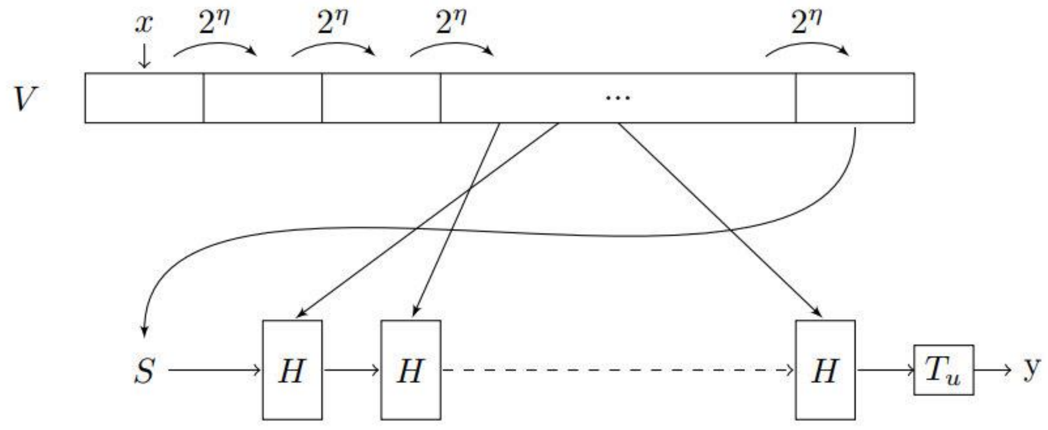
Example: Memory-Hard function *Diodon*

Algorithm 1 DIODON Asymmetrically memory-hard function

Inputs: t-bit block x; RSA modulus N of n_p bits; M, L;

Output: u-bit output y

```
V_0 = x
for all i ∈ {1, ..., M − 1} do
    V_i = V_{i−1}^{2^η} mod N
end for
S = V_{M−1}
for all i ∈ {0, ..., L − 1} do
    j = S mod M
    S = H(S, V_j)
end for
return T_u(S)
```



Example: Memory-Hard function *Diodon*

Algorithm 2 DIODON for privileged users
Inputs: t-bit block x; RSA factors q, q'; η; M, T;
Output: u-bit output y

$e = 2^{(M-1)\times\eta} \mod (q-1)(q'-1)$
$S = x^e \mod (qq')$
for all $i \in \{0, ..., L-1\}$ **do**
  $j = S \mod M$
  $e_j = 2^{j\times\eta} \mod (q-1)(q'-1)$
  $S = H\big(S, (x^{e_j} \mod (qq'))\big)$
end for
return $T_u(S)$

Resource hardness Framework

Parameters	Conservative	Fast
t	128	128
u	128	128
n_p	2048	1024
η	2048	1
M	$4,000$	$8,000,000$
L	$4,000$	$20,000$
RAM (basic user)	1 Mb	1 Gb
Time (basic user)	10.00 s	9.87 s
Time (privileged)	13.49 s	10.65 s

n_p – bits in RSA modulus; *t,u* –input/output sizes; *M,L*-upper/lower chain length

Resource hardness Framework

Parameters	Conservative	Fast
t	128	128
u	128	128
n_p	2048	1024
η	2048	1
M	$4,000$	$8,000,000$
L	$4,000$	$20,000$
RAM (basic user)	1 Mb	1 Gb
Time (basic user)	10.00 s	9.87 s
Time (privileged)	13.49 s	10.65 s

Open problem: Diodon is based on scrypt which has lousy linear TM-tradeoff. Also slow due to RSA. Improve?

Few other things

R-hardness and code obfuscation

Using obfuscation idea from [BK'16*]:

- Compiler that runs some resource hard function *F(pwd,x)*
- Computes *R*-hard bits $F(pwd,x) = b_i$ and then makes code transformations:

for $b_i = 0$

if $x \oplus b_i$ **then** A
else B

if x **then** A
else B

if $x \oplus b_i$ **then** B
else A

for $b_i = 1$.

*Biryukov, Khovratovich, Egalitarian Computing, Usenix'16

R-hardness and code obfuscation

Using obfuscation idea from [BK'16]:

- Compiler that runs some resource hard function *F(pwd,x)*
- Computes *R*-hard bits $F(pwd,x) = b_i$ and then makes code transformations:

if x **then** A **else** B

for $b_i = 0$:
if $x \oplus b_i$ **then** A **else** B

for $b_i = 1$:
if $x \oplus b_i$ **then** B **else** A

- The user will have to run *R*-hard function *F(pwd,x)* at least once

R-hardness and code obfuscation

Using obfuscation idea from [BK'16]:

- Compiler that runs some resource hard function *F(pwd,x)*
- Computes *R*-hard bits $F(pwd,x) = b_i$ and then makes code transformations:

if x **then** A **else** B

for $b_i = 0$

if $x \oplus b_i$ **then** A **else** B

for $b_i = 1$.

if $x \oplus b_i$ **then** B **else** A

- This could work well for previously unseen code.

R-hardness and code obfuscation

Using obfuscation idea from [BK'16]:

- Compiler that runs some resource hard function *F(pwd,x)*
- Computes *R*-hard bits $F(pwd,x) = b_i$ and then makes code transformation:

if x **then** A
else B

for $b_i = 0$
if $x \oplus b_i$ **then** A
else B

for $b_i = 1$.
if $x \oplus b_i$ **then** B
else A

Would this approach work to make Incompressible, password protected INC-AES ?

R-hardness and code obfuscation

- Not really. Unless we already have *K*-unextractable/unbreakable UBK-AES.
- However it shows hope that at least in some cases UBK => INC

Related topics

Related research topics

- Code Obfuscation (for structure hiding)
- Cross-pollination with GreyBox crypto (for value hiding)
- IO
- Malicious crypto – adversarial crypto design
- PK crypto based on new ideas

Open problems

- Can we design a WBC-friendly cipher?
- Would Even-Mansour cipher be a good candidate?
- Design Diodon-like asymmetric memory hard functions with non-linear TM tradeoffs and faster operations
- INC-PWD-AES?

End

(and we are hiring postdocs on WBC and other topics)

cryptolux.org