Cracking HiTag2 Crypto

Weaponising Academic Attacks for Breaking and Entering

DC4420

Kev Sheldrake

rtfcode@gmail.com || @kevsheldrake

github/rtfcode rtfc.org.uk

Kev Sheldrake

DC4420

- Hacker
- Researcher
- Reverse engineer
- Maker

- Embedded systems
- Cryptography
- RFID
- Software defined radio
- Tool dev

Why copy 125KHz RFID tags?

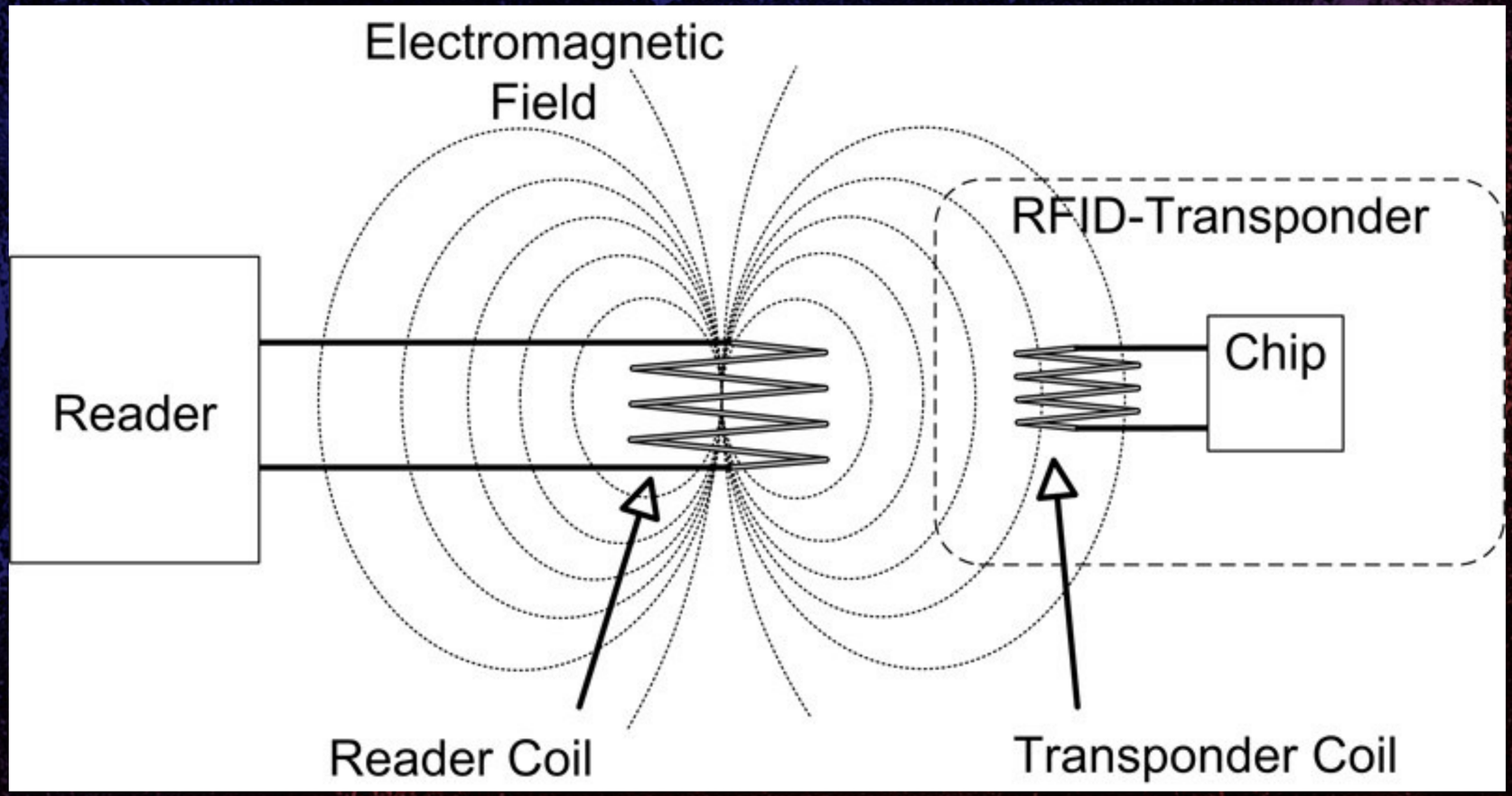
- Building access
- Car de-immobilization / ignition

- 125KHz tags are very low power
- No crypto / weak crypto
- Low level security
- Often used to secure high value assets

Simple 125KHz RFID tag

- Notionally a single page of data
- Typically 26 bits to 88 bits
- Public (continuously transmitted)
- UID contains synchronisation bits

How simple 125KHz RFID works



Wikipedia: User Maxpayne473

Tag is powered by field

Tag and reader communicate by modulating field

Data modulation and encoding

- Tag data modulation:
  - Amplitude Shift Keying (ASK)
  - Frequency Shift Keying (FSK)
  - Phase Shift Keying (PSK)
- Reader data modulation:
  - Pulse Width Modulation (PWM)
- Encoding: Manchester, biphase, raw

‘Secure’ 125KHz RFID

DC4420

- 2-way communications
- Keyed and nonce-based (randomised):
  - Authentication
  - Encryption
- Can’t replay
- Can’t copy

Contents

- HiTag2 password mode
- HiTag2 crypto mode
  - Pseudo-random number generator (PRNG)
  - Stream cipher
  - Authentication
  - Encryption
  - Three attacks from ‘Gone in 360 Seconds’
  - Attack from ‘Lock It and Still Lose It’

DC4420

Exclusions

- HiTag2 Public Modes A, B and C with/without password and crypto modes
- Mapping between internal and external formats
- Back-end communications protocols
- The depths of the research
- Read the code for implementation details
- Car-specific details

DC4420

Academic Paper

Gone in 360 Seconds: Hijacking with Hitag2

Roel Verdult Flavio D. Garcia

Institute for Computing and Information Sciences
Radboud University Nijmegen, The Netherlands.
{rverdult,flaviog}@cs.ru.nl

Josep Balasch

KU Leuven ESAT/COSIC and IBBT
Kasteelpark Arenberg 10, 3001 Heverlee, Belgium
josep.balasch@esat.kuleuven.be

Abstract

An electronic vehicle immobilizer is an anti-theft device which prevents the engine of the vehicle from starting unless the corresponding transponder is present. Such a transponder is a passive RFID tag which is embedded in the car key and wirelessly authenticates to the vehicle. It prevents a perpetrator from hot-wiring the vehicle or starting the car by forcing the mechanical lock. Having such an immobilizer is required by law in several countries. Hitag2, introduced in 1996, is currently the most widely used transponder in the car immobilizer industry. It is used by at least 34 car makes and fitted in more than 200 different car models. Hitag2 uses a proprietary stream cipher with 48-bit keys for authentication and confidentiality. This article reveals several weaknesses in the design of the cipher and presents three practical attacks that recover the secret key using only wireless communication. The most serious attack recovers the secret key from a car in less than six minutes using ordinary hardware. This attack allows an adversary to bypass the cryptographic authentication, leaving only the mechanical key as safeguard. This is even more sensitive on vehicles where the physical key has been replaced by a keyless entry system based on Hitag2. During our experiments we managed to recover the secret key and start the engine of many vehicles from various makes using our transponder emulating device. These experiments also revealed several implementation weaknesses in the immobilizer units.

1 Introduction

In the past, most cars relied only on mechanical keys to prevent a hijacker from stealing the vehicle. Since the '90s most car manufacturers incorporated an electronic car immobilizer as an extra security mechanism in their vehicles. From 1995 it is mandatory that all cars sold in the EU are fitted with such an immobilizer device, according to European directive 95/56/EC. Similar regulations apply to other countries like Australia, New Zealand (AS/NZS 4601:1999) and Canada (CAN/ULC S338-98). An electronic car immobilizer consists of two main components: a small transponder chip which is embedded in (the plastic part of) the car key, see Figure 1; and a reader which is located somewhere in the dashboard of the vehicle and has an antenna coil around the ignition, see Figure 2.

Figure 1: Car keys with a Hitag2 transponder/chip

The transponder is a passive RFID tag that operates at a low frequency wave of 125 kHz. It is powered up when it comes in proximity range of the electronic field of the reader. When the transponder is absent, the immobilizer unit prevents the vehicle from starting the engine.

Figure 2: Immobilizer unit around the ignition barrel

A distinction needs to be made with remotely operated central locking system, which opens the doors, is battery powered, operates at a ultra-high frequency (UHF) of 433 MHz, and only activates when the user pushes a

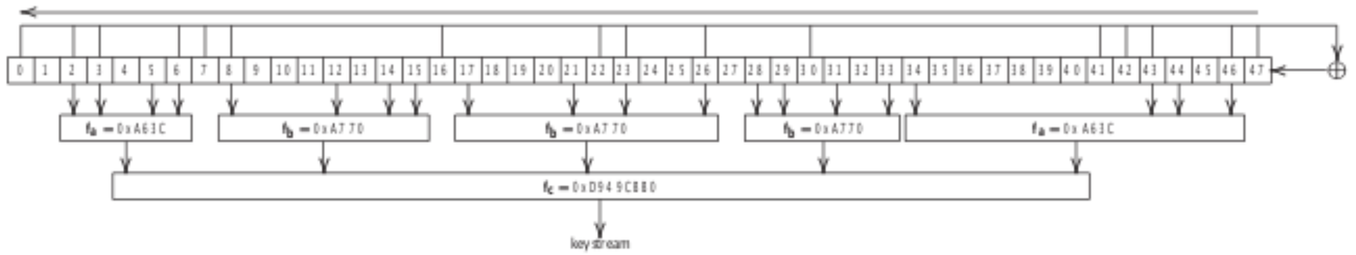


Figure 11: Structure of the Hitag2 stream cipher, based on [47]

3.6 Cipher Initialization

The following precisely defines the initialization of the cipher and the generation of the LFSR-stream $\mathbf{a}_0\mathbf{a}_1\ldots$ and the keystream $\mathbf{b}_0\mathbf{b}_1\ldots$.

Definition 3.5. **Given a key $\mathbf{k} = \mathbf{k}_0\ldots\mathbf{k}_{47} \in \mathbb{F}_2^{48}$, an identifier $\mathbf{id} = \mathbf{id}_0\ldots\mathbf{id}_{31} \in \mathbb{F}_2^{32}$, a reader nonce $\mathbf{n_R} = \mathbf{n_{R_0}}\ldots\mathbf{n_{R_{31}}} \in \mathbb{F}_2^{32}$, a reader answer $\mathbf{a_R} = \mathbf{a_{R_0}}\ldots\mathbf{a_{R_{31}}} \in \mathbb{F}_2^{32}$, and a transponder answer $\mathbf{a_T} = \mathbf{a_{T_0}}\ldots\mathbf{a_{T_{31}}} \in \mathbb{F}_2^{32}$, the internal state of the cipher at time $\mathbf{i}$ is $\alpha_\mathbf{i} := \mathbf{a_i}\ldots\mathbf{a_{47+i}} \in \mathbb{F}_2^{48}$. Here the $\mathbf{a_i} \in \mathbb{F}_2$ are given by**

$$\begin{aligned}
\mathbf{a_i} &:= \mathbf{id_i} && \forall \mathbf{i} \in [0,31]\\
\mathbf{a_{32+i}} &:= \mathbf{k_i} && \forall \mathbf{i} \in [0,15]\\
\mathbf{a_{48+i}} &:= \mathbf{k_{16+i}} \oplus \mathbf{n_{R_i}} && \forall \mathbf{i} \in [0,31]\\
\mathbf{a_{80+i}} &:= \mathbf{L}(\mathbf{a_{32+i}}\ldots\mathbf{a_{79+i}}) && \forall \mathbf{i} \in \mathbb{N}.
\end{aligned}$$

Furthermore, we define the keystream bit $\mathbf{b_i} \in \mathbb{F}_2$ at time $\mathbf{i}$ by

$$\mathbf{b_i} := \mathbf{f}(\mathbf{a_i}\ldots\mathbf{a_{47+i}}) \qquad \forall \mathbf{i} \in \mathbb{N}.$$

Define $\{\mathbf{n_R}\}, \{\mathbf{a_R}\}_\mathbf{i}, \{\mathbf{a_T}\}_\mathbf{i} \in \mathbb{F}_2$ by

$$\begin{aligned}
\{\mathbf{n_R}\}_\mathbf{i} &:= \mathbf{n_{R_i}} \oplus \mathbf{b_i} && \forall \mathbf{i} \in [0,31]\\
\{\mathbf{a_R}\}_\mathbf{i} &:= \mathbf{a_{R_i}} \oplus \mathbf{b_{32+i}} && \forall \mathbf{i} \in [0,31]\\
\{\mathbf{a_T}\}_\mathbf{i} &:= \mathbf{a_{T_i}} \oplus \mathbf{b_{64+i}} && \forall \mathbf{i} \in [0,31].
\end{aligned}$$

Note that the $\mathbf{a_i}$, $\alpha_\mathbf{i}$, $\mathbf{b_i}$, $\{\mathbf{n_R}\}_\mathbf{i}$, $\{\mathbf{a_R}\}_\mathbf{i}$, and $\{\mathbf{a_T}\}_\mathbf{i}$ are formally functions of $\mathbf{k}$, $\mathbf{id}$, and $\mathbf{n_R}$. Instead of making this explicit by writing, e.g., $\mathbf{a_i}(\mathbf{k},\mathbf{id},\mathbf{n_R})$, we just write $\mathbf{a_i}$ where $\mathbf{k}$, $\mathbf{id}$, and $\mathbf{n_R}$ are clear from the context.

Theorem 3.7. **In the situation from Definition 3.5, we have**

$$\begin{aligned}
\mathbf{a_{32+i}} &= \mathbf{R}(\mathbf{a_{33+i}}\ldots\mathbf{a_{80+i}}) && \forall \mathbf{i} \in \mathbb{N}\\
\mathbf{a_i} &= \mathbf{id_i} && \forall \mathbf{i} \in [0,31].
\end{aligned}$$

Proof. Straightforward, using Definition 3.5 and Equation (1). □

If an attacker manages to recover the internal state of the LFSR $\alpha_\mathbf{i} = \mathbf{a_i}\mathbf{a_{i+1}}\ldots\mathbf{a_{i+47}}$ at some time $\mathbf{i}$, then she can repeatedly apply Theorem 3.7 to recover $\mathbf{a_0}\mathbf{a_1}\ldots\mathbf{a_{79}}$ and, consequently, the keystream $\mathbf{b_0}\mathbf{b_1}\mathbf{b_2}\ldots$. By having eavesdropped $\{\mathbf{n_R}\}$ from the authentication protocol, the adversary can further calculate

$$\mathbf{n_{R_i}} = \{\mathbf{n_R}\}_\mathbf{i} \oplus \mathbf{b_i} \qquad \forall \mathbf{i} \in [0,31].$$

Finally, the adversary can compute the secret key as follows

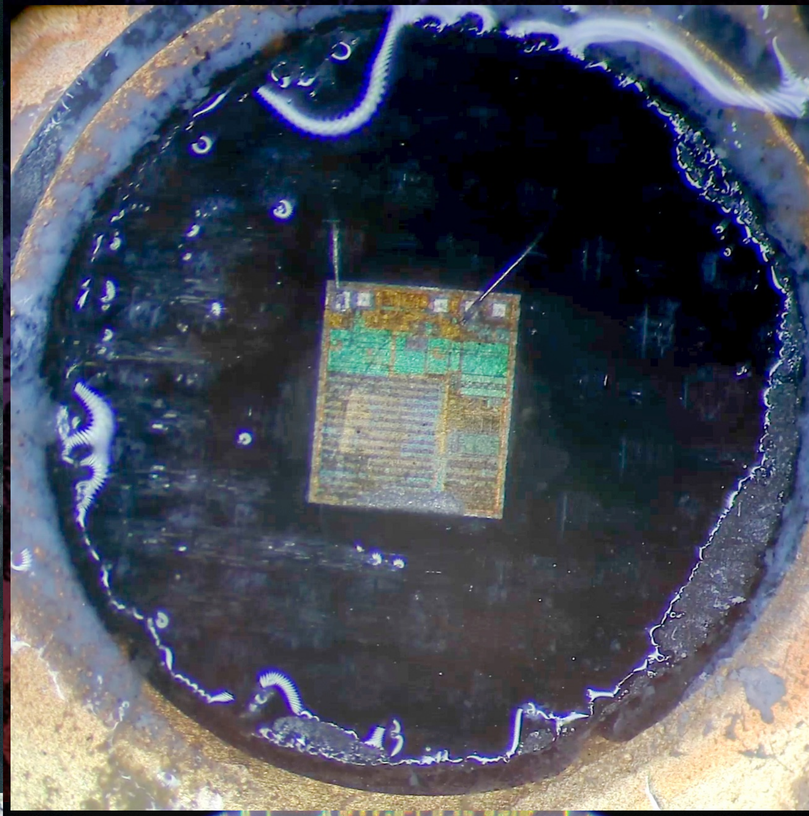
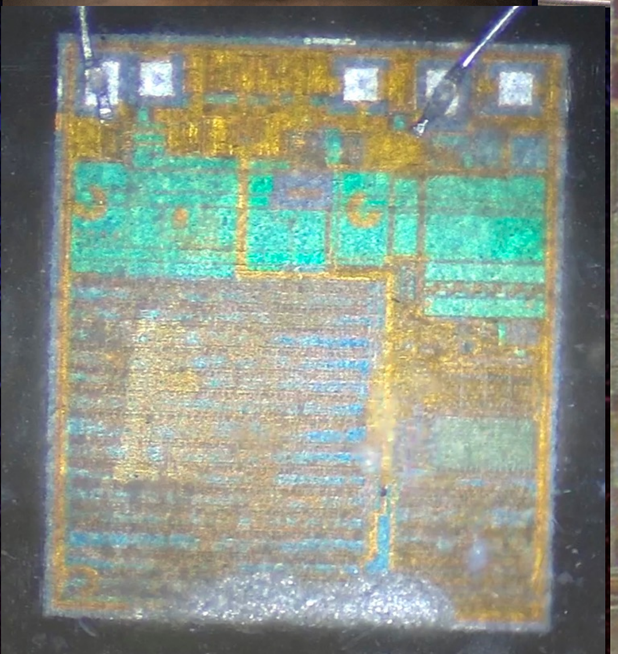
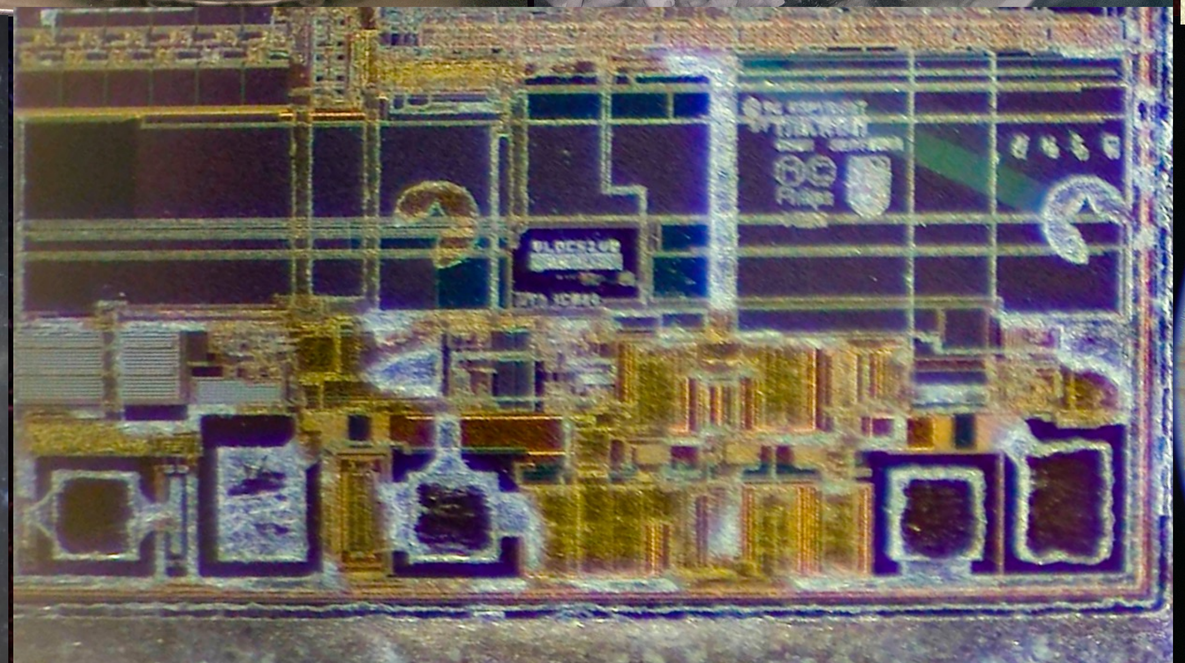
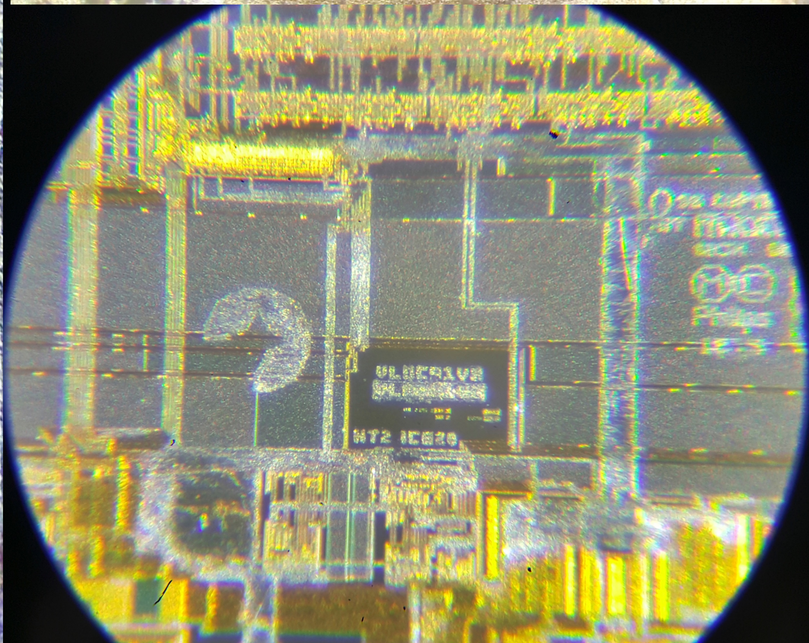
$$\begin{aligned}
\mathbf{k_i} &= \mathbf{a_{32+i}} && \forall \mathbf{i} \in [0,15]\\
\mathbf{k_{16+i}} &= \mathbf{a_{48+i}} \oplus \mathbf{n_{R_i}} && \forall \mathbf{i} \in [0,31].
\end{aligned}$$

4 Hitag2 weaknesses

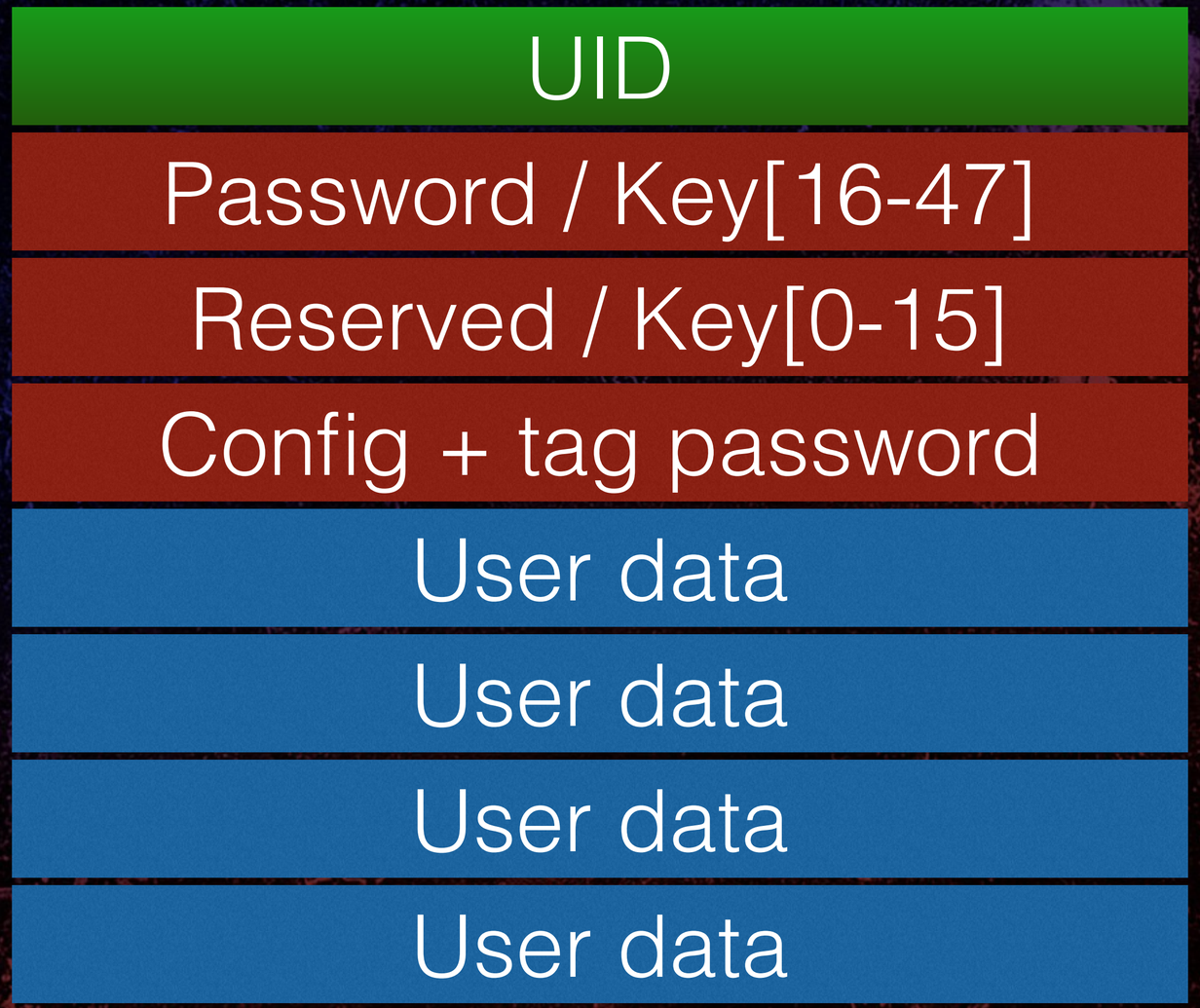
This section describes three weaknesses in the design of Hitag2. The first one is a protocol flaw while the last two concern the cipher's design. These weaknesses will later be exploited in Section 5.

DC4420

HiTag2 Overview

HiTag2 125KHz RFID tags



- 8 pages
- 32 bits each
- UID is public

HiTag2 auth and encryption

- Pre-shared secrets
- Password mode: uses pre-shared passwords for authentication
- Crypto mode: uses pre-shared keys for authentication and encryption
- Reader has same keys/passwords as tags
- Mutual authentication

HiTag2 password mode

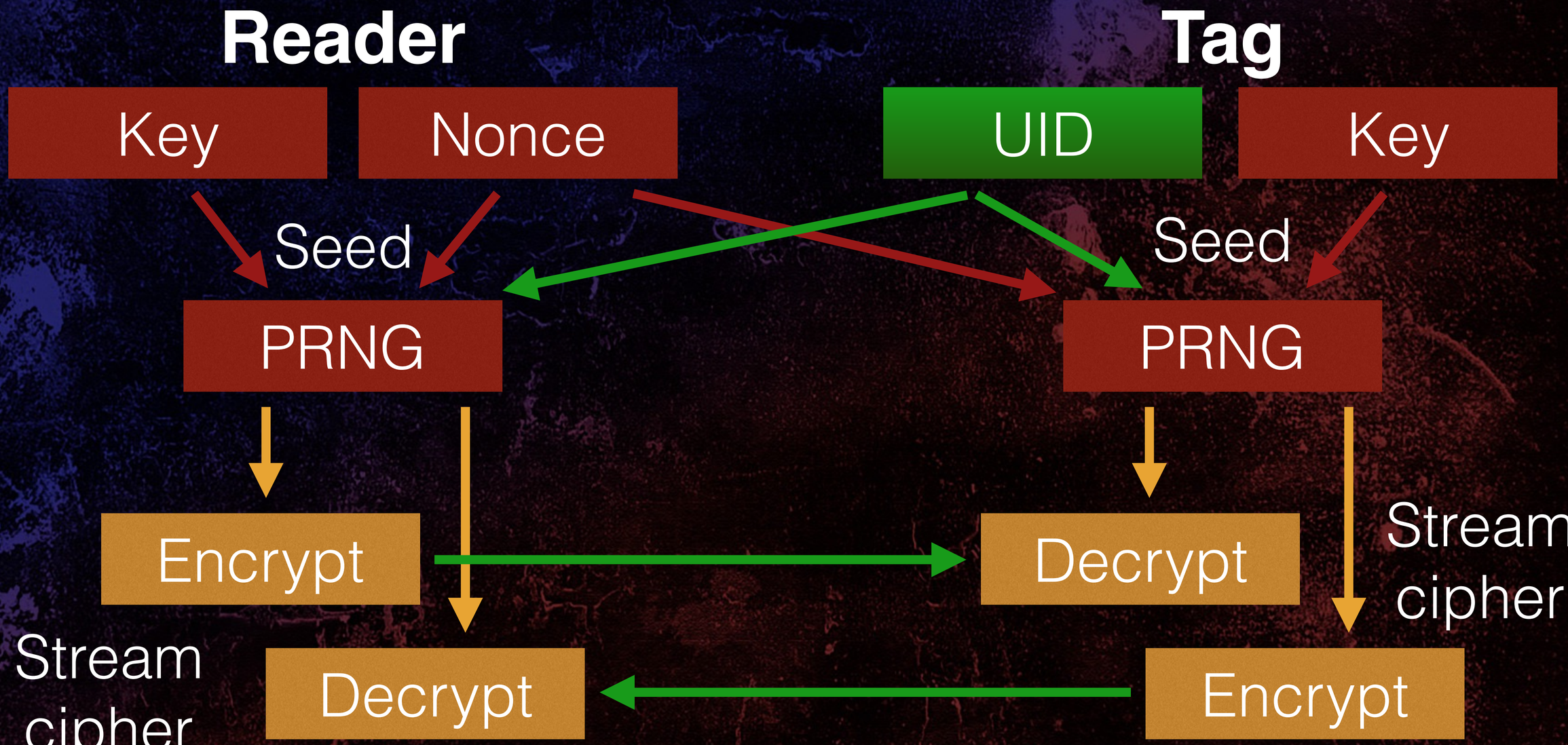
DC4420

- Reader sends START_AUTH (11000)
- Tag sends UID (p0) - [in clear]
- Reader sends password (p1) - [in clear!]
- Tag sends config & tag password (p3) - [in clear!]
- Reader sends commands - [in clear!]
- Tag responds - [in clear!]

HiTag2 crypto mode

- Reader sends START_AUTH (11000)
- Tag sends UID (p0) - [in clear]
- Reader sends nonce (nR) - [encrypted]
- Reader sends 0xFFFFFFFF (aR) - [encrypted]
- Tag sends config & tag password (p3) - [encrypted]
- Reader sends commands - [encrypted]
- Tag responds - [encrypted]

HiTag2 crypto overview



HiTag2 encryption

- Stream cipher
- Seeded PRNG
- data XOR (PRNG output) = encrypted data
- encrypted data XOR (PRNG output) = data

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

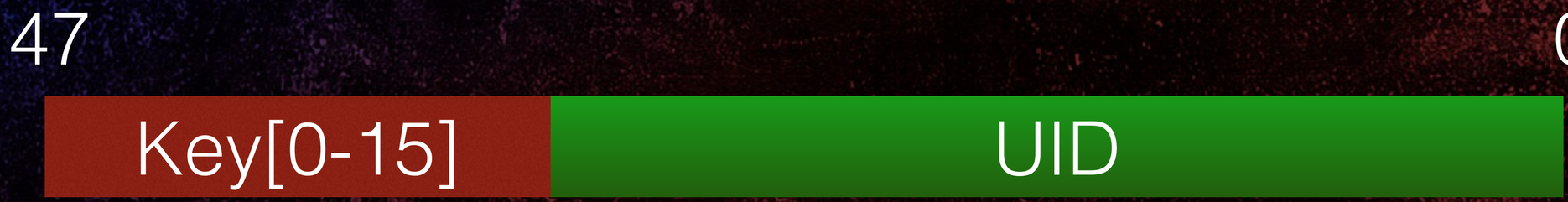
DC4420

HiTag2 PRNG and encryption

HiTag2 encryption

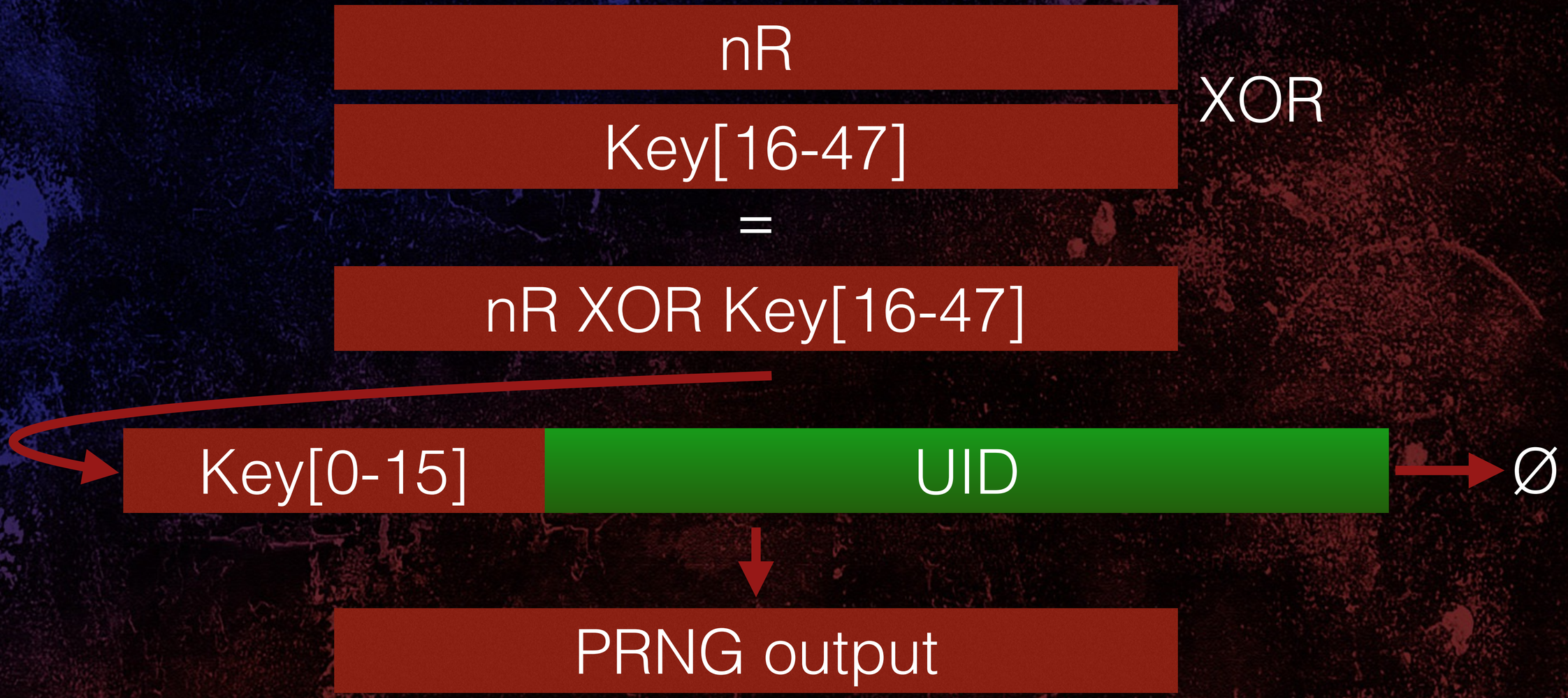
DC4420

- 48 bit Linear Feedback Shift Register (LFSR) Pseudo-Random Number Generator (PRNG)
- Seed with: UID (32 bits) + Key[0-15] (16 bits)



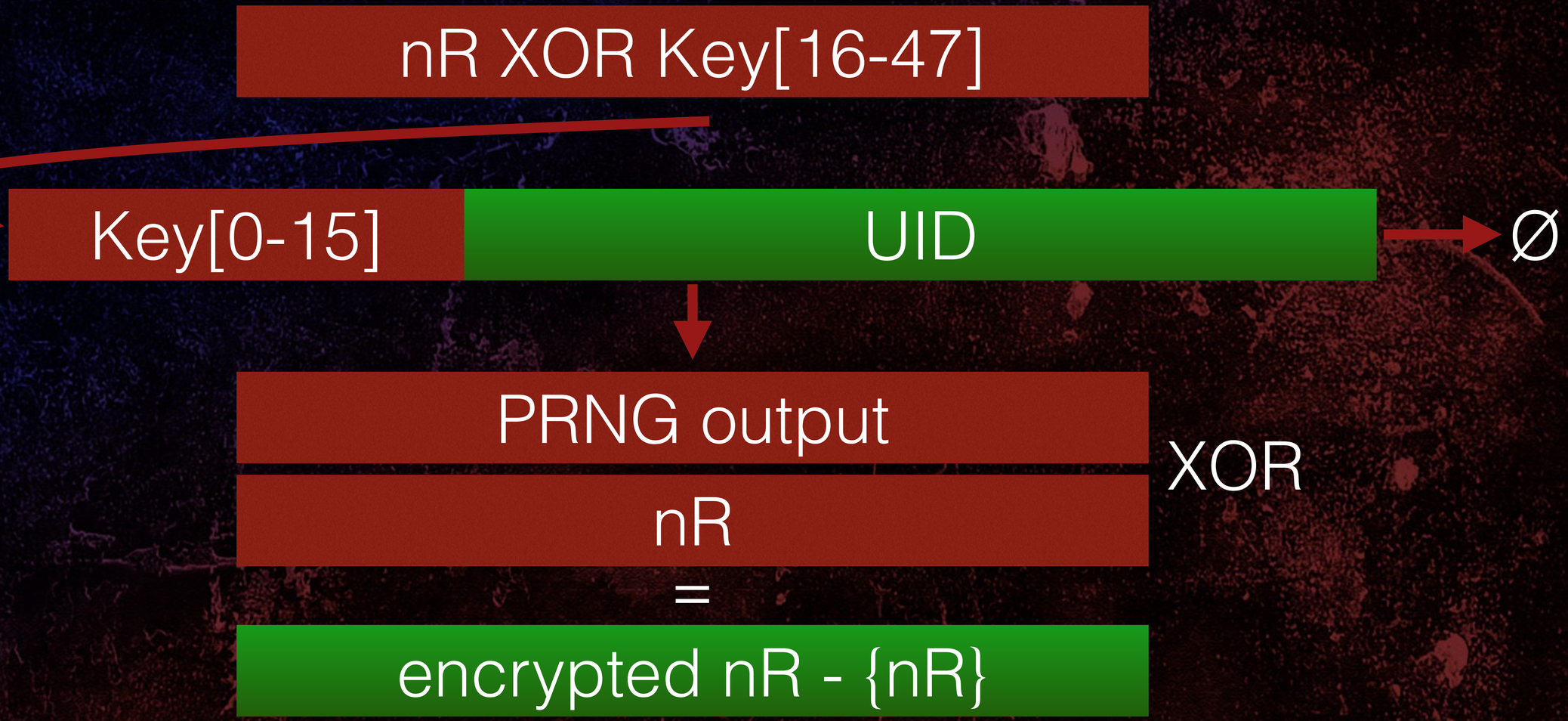
HiTag2 encryption

- Shift in: nR XOR Key[16-47] (32 bits), 1 bit at a time



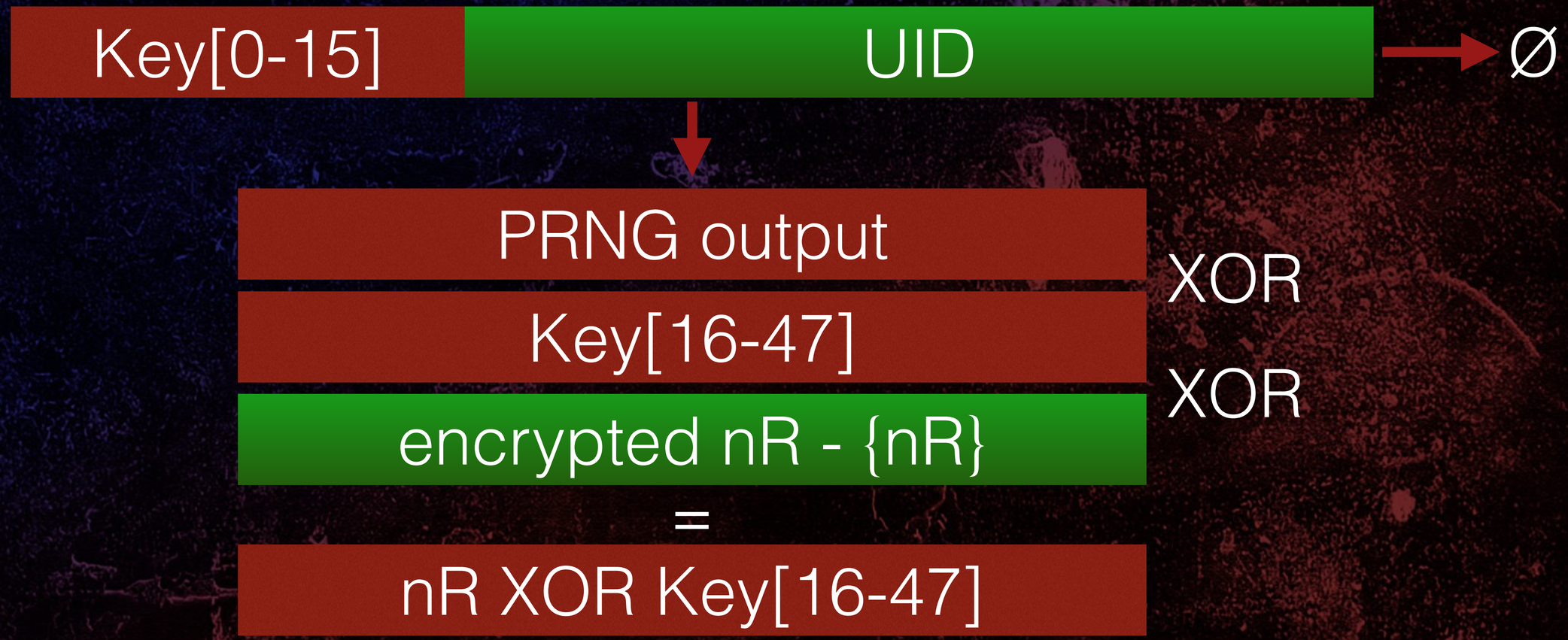
HiTag2 encryption

- Encrypt nR by XORing it with PRNG output, 1 bit at a time



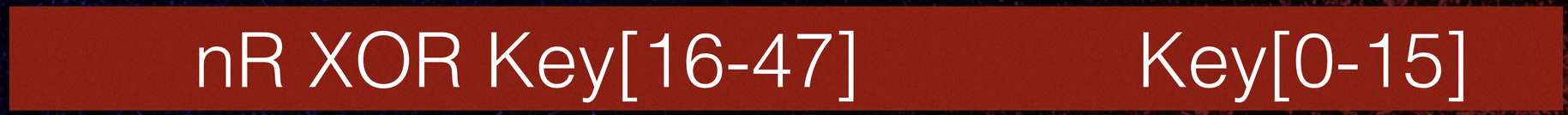
HiTag2 encryption

- Tag achieves same state (again, 1 bit at a time)

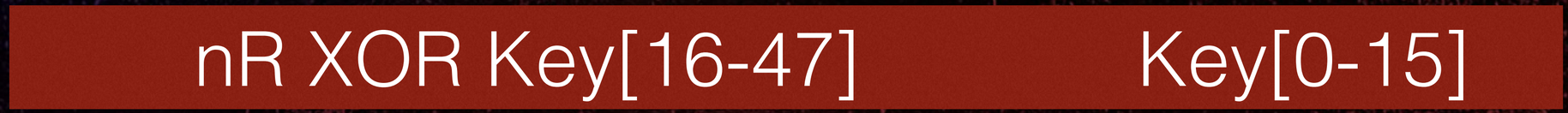


HiTag2 encryption

- Reader after sending {nR}

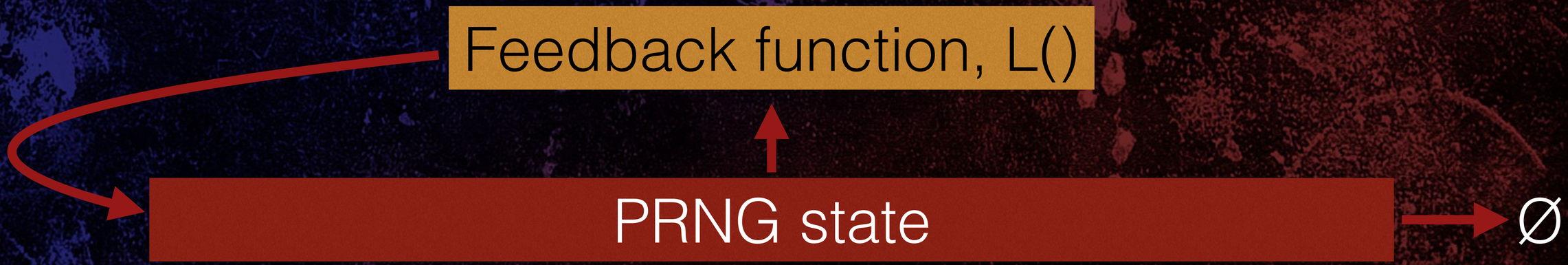


- Tag after receiving {nR}



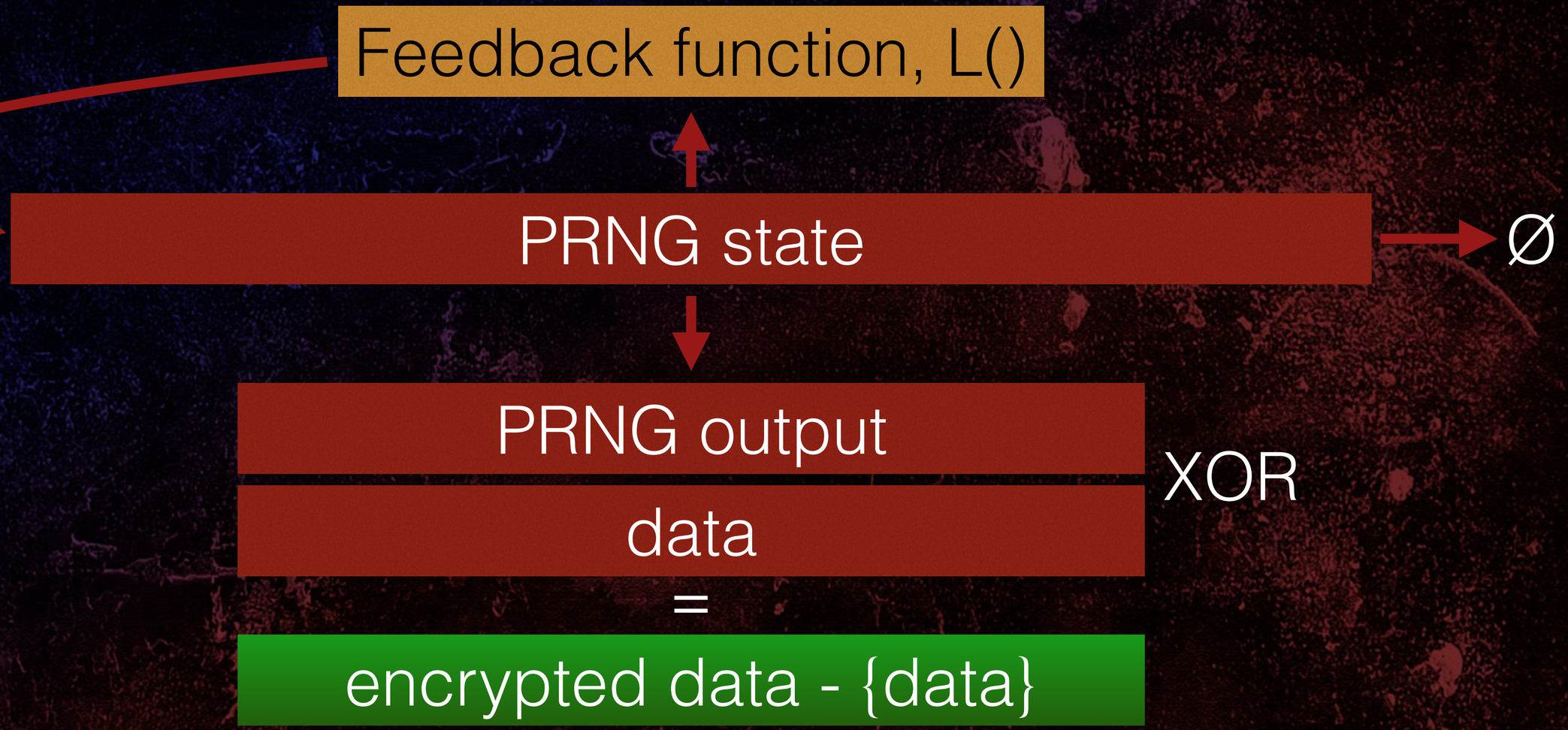
HiTag2 encryption

- Enable feedback



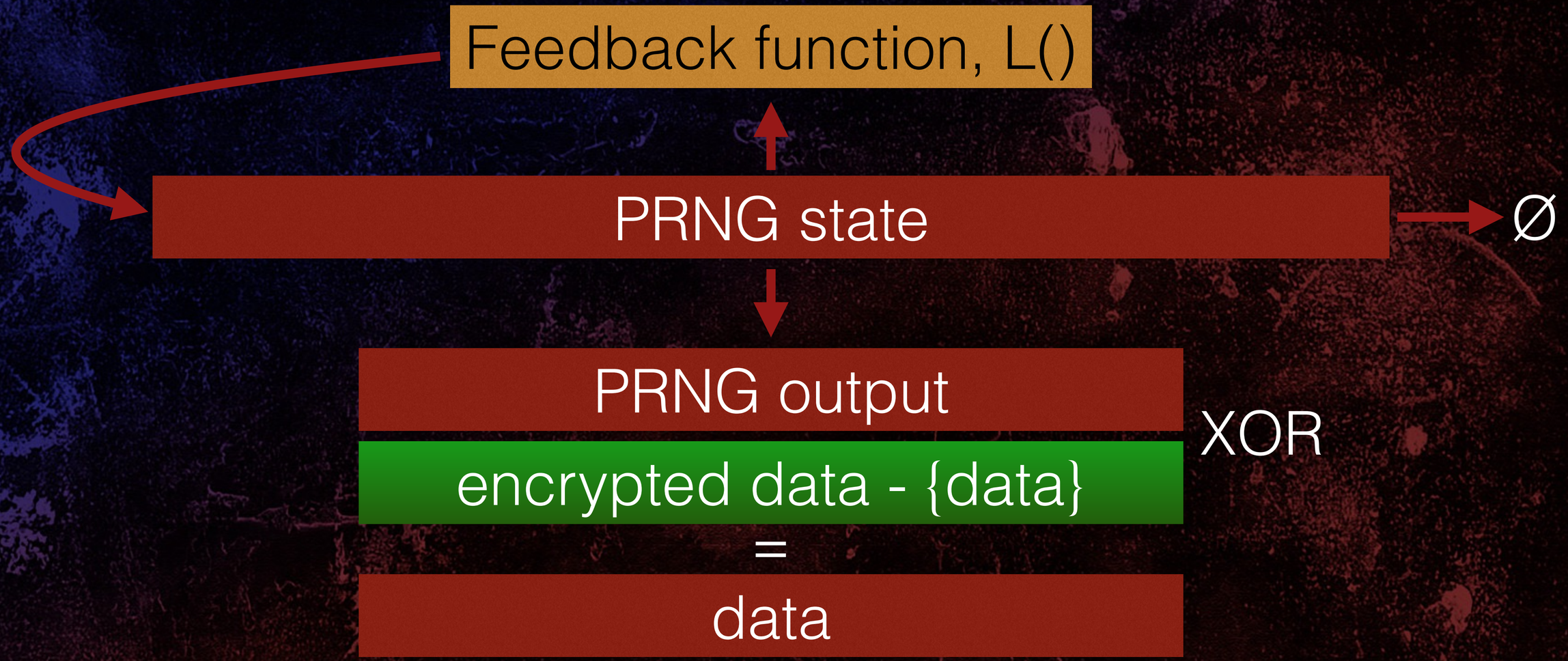
HiTag2 encryption

- Encrypt data by XORing it with PRNG output



HiTag2 encryption

- Decrypt data by XORing it with PRNG output



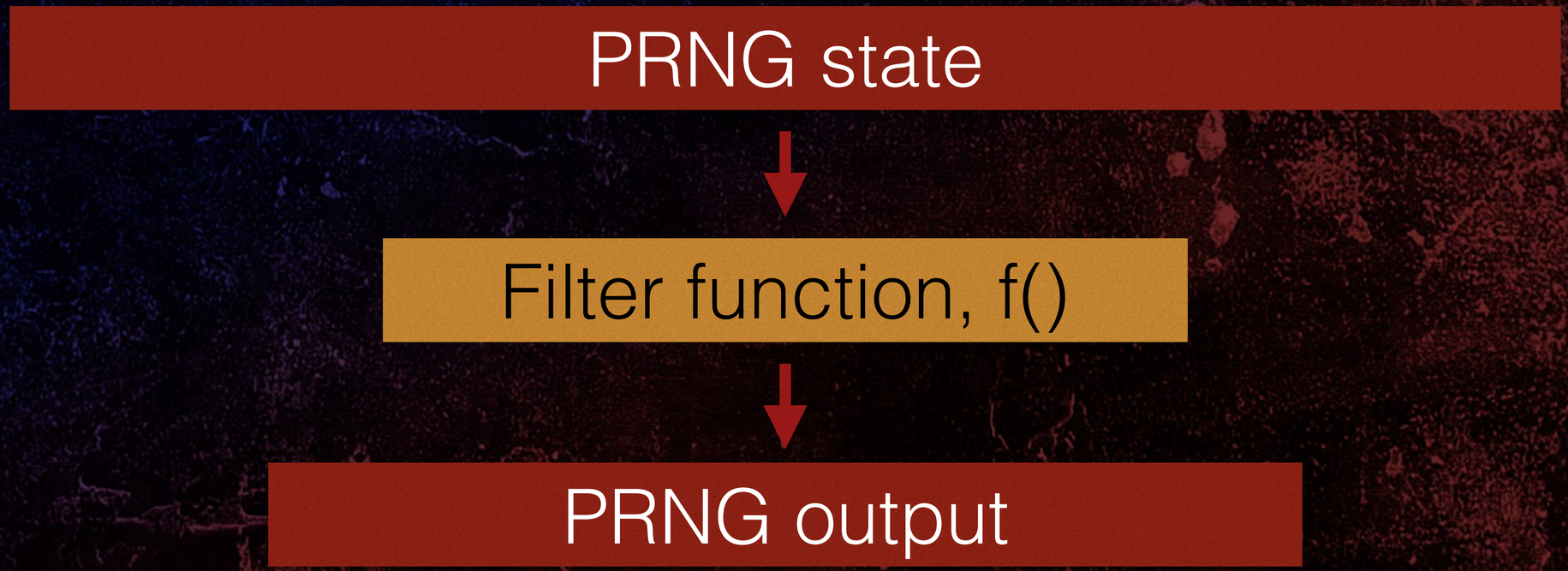
Feedback function, L()

DC4420

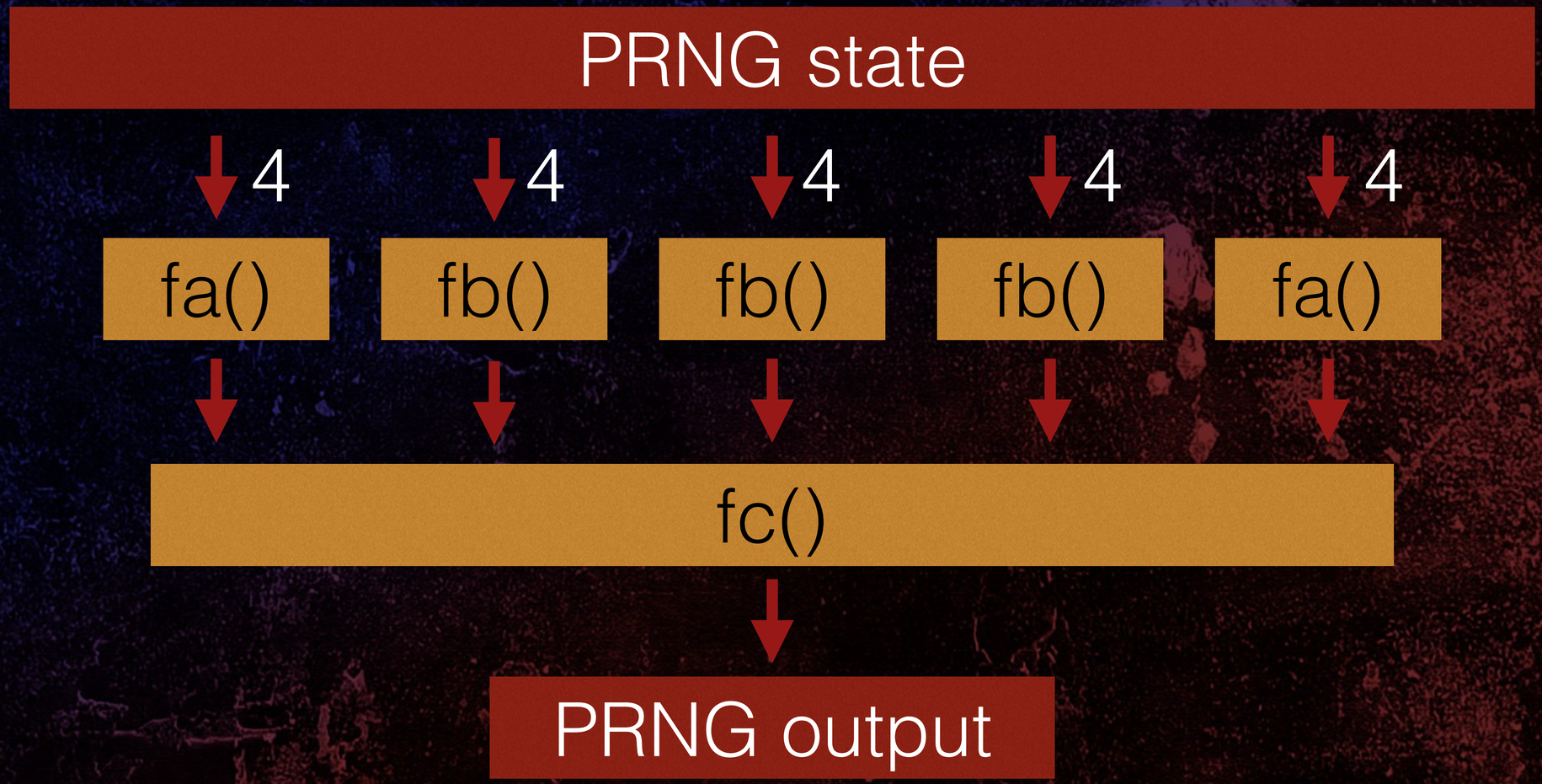
- Produces the new bit that is shifted into LFSR.
- Treating the LFSR state as an array of bits…
- L(x) = XOR(x[0], x[2], x[3], x[6], x[7], x[8], x[16], x[22], x[23], x[26], x[30], x[41], x[42], x[43], x[46], x[47]);

PRNG output

- Output is produced by the filter function, f()



Filter function, f()



HiTag2 crypto mode

- Reader sends START_AUTH (11000)
- Tag sends UID (p0) - [in clear]
- Reader sends nonce (nR) - [encrypted]
- Reader sends 0xFFFFFFFF (aR) - [encrypted]
- Tag sends config & tag password (p3) - [encrypted]
- Reader sends commands - [encrypted]
- Tag responds - [encrypted]

DC4420

HiTag2 Commands

HiTag2 commands

CM1	CM0	A4	A3	A2	Command
1	1	x	x	x	READ PAGE
0	1	x	x	x	READ PAGE INVERTED
1	0	x	x	x	WRITE PAGE
0	0	x	x	x	HALT

HiTag2 commands

Standard:

11000 00111

Extended:

...

DC4420

Attacks

Academic paper from 2012

- Gone in 360 Seconds: Hijacking with Hitag2
- Roel Verdult, Flavio D. Garcia, Josep Balasch
- Three attacks on HiTag2:
  - Nonce replay and length extension for key stream recovery
  - Time/memory trade off for key recovery
  - Cryptanalytic attack for key recovery

Academic paper from 2016

DC4420

- Lock It and Still Lose It - On the (In)Security of Automotive Remote Keyless Entry Systems
- Flavio D. Garcia, David Oswald, Timo Kasper, Pierre Pavlides
- New attack on HiTag2 crypto:
  - Fast correlation attack for key recovery

DC4420

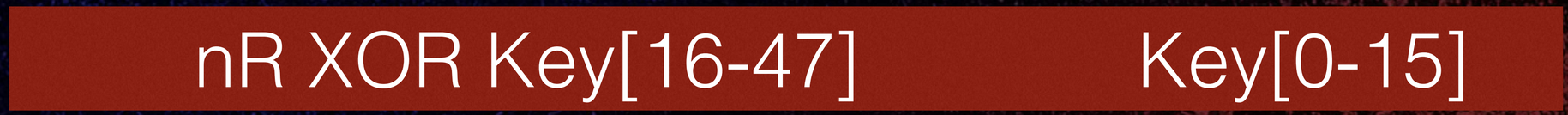
Nonce replay and length extension for key stream recovery

Nonce replay

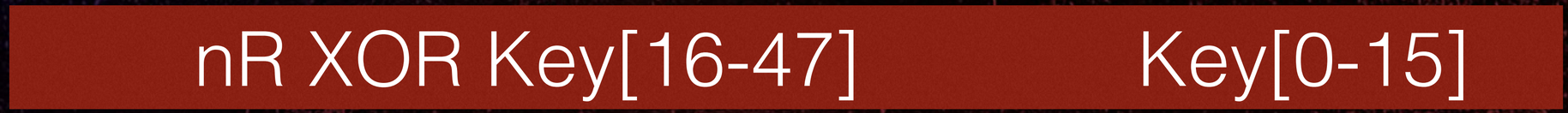
- All entropy comes from reader (nR)
- Reader presents encrypted nonce and challenge response ({nR} {aR})
- Capture these ({nR}, {aR}) values
- Emulate reader and replay same ({nR}, {aR}) for every communication for common UID (same tag)
- Initialise before every test =>
  - PRNG starts in same state each time!

HiTag2 encryption

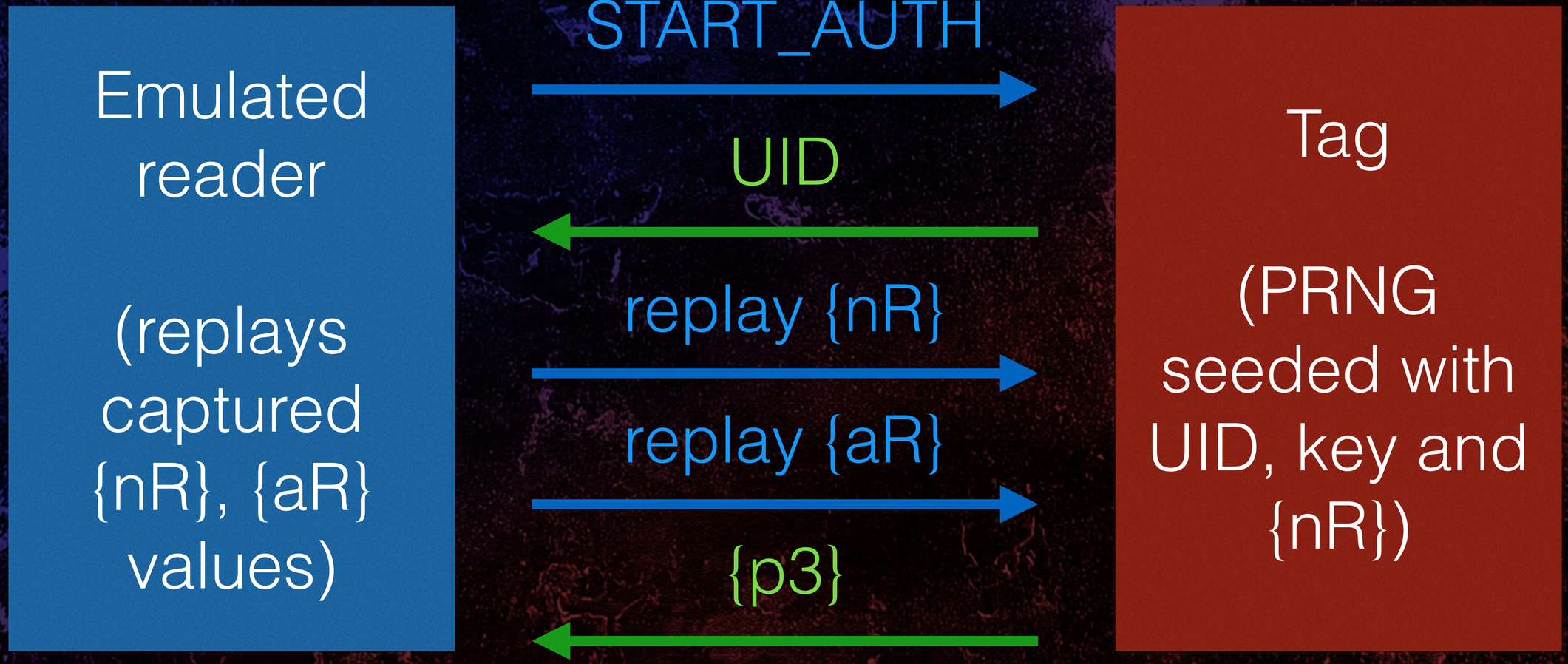
- Reader after sending {nR}



- Tag after receiving {nR}



Emulate reader



- Output of tag PRNG (key stream) depends solely on {nR}

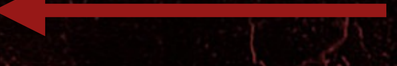
Nonce replay attack

{nR}
0x4A85B2DF

Tag

(PRNG seeded with UID, key and {nR})

PRNG output:
01011100011101000 1…



- Reset PRNG to same state for every session / guess

Find encrypted ‘read p0’ command

- ‘Read page’ commands are 10 bits long (2 x 5 bits)
- Commands are encrypted - appear random
- 1024 possibilities
- 16 valid read commands (8 normal, 8 inverted)
- Valid read commands return 32 bits of encrypted data
- Bad commands return UNENCRYPTED error response, 0xF402889C

Find encrypted ‘read p0’ command

DC4420

- Approach:
  - Find one encrypted ‘read page’ command
  - Bit-flip to find remaining 15 ‘read page’ commands
  - Test each one to find ‘read page 0’ command

Find one encrypted ‘read’ command

- for (i=0; i<1024; i++) {
  - setup(); [START_AUTH, UID, {nR}, {aR}, {p3}]
  - response = send(i);
  - if (response != ERROR_RESPONSE) {
    - return i;
  - }
- }

- Note:
  - Non-optimised
  - Only need to brute force 6 bits in reality

Find one encrypted ‘read’ command

CM1	CM0	A4	A3	A2	Command
1	1	x	x	x	READ PAGE
0	1	x	x	x	READ PAGE INVERTED

Valid encrypted command: 0b*EFGHIJKLMN*

cmd	a	**1**	b	c	d		~a	**0**	~b	~c	~d
enc	*E*	***F***	*G*	*H*	*I*		*J*	***K***	*L*	*M*	*N*

Find one encrypted ‘read’ command

- ‘read’ command is: a,1,b,c,d,~a,0,~b,~c,~d
- Encrypted cmd is: *E,F,G,H,I,J,K,L,M,N*
- We don’t know the key stream, but:
  - *(F,K)* must decrypt to (1,0) for a ‘read’
  - *(E,G,H,I)* decrypt to inv. of decryption of *(J,L,M,N)*
- We can fix *(J,L,M,N)* and try all values for *(E,G,H,I)*
- Therefore only need to brute force:
  - *(E,F,G,H,I,K)* = 6 bits = 64 max, 32 average

Find all encrypted ‘read’ commands

CM1	CM0	A4	A3	A2	Command
1	1	x	x	x	READ PAGE
0	1	x	x	x	READ PAGE INVERTED

Valid encrypted command: 0b*EFGHIJKLMN*

cmd	a	**1**	b	c	d		~a	**0**	~b	~c	~d
enc	*E*	***F***	*G*	*H*	*I*		*J*	***K***	*L*	*M*	*N*

Flip ‘inverted’ bit

enc	*E=0*	*F=0*	*G=1*	*H=1*	*I=1*	*J=0*	*K=0*	*L=0*	*M=0*	*N=0*
ks	0	1	0	1	1	1	0	0	1	1
cmd	0	1	1	0	0	1	0	0	1	1

‘read inverted’ ‘page 4’ ~‘read page 4 inverted’

enc	***1***	*0*	*1*	*1*	*1*	***1***	*0*	*0*	*0*	*0*
ks	**0**	1	0	1	1	**1**	0	0	1	1
cmd	**1**	1	1	0	0	**0**	0	0	1	1

‘read’ ‘page 4’ ~‘read page 4’

DC4420

Flip ‘page’ bit

enc	*E=0*	*F=0*	*G=1*	*H=1*	*I=1*		*J=0*	*K=0*	*L=0*	*M=0*	*N=0*
ks	0	1	0	1	1		1	0	0	1	1
cmd	0	1	1	0	0		1	0	0	1	1

‘read inverted’ ‘page 4’ ~‘read page 4 inverted’

enc	*0*	*0*	*1*	*1*	***0***		*0*	*0*	*0*	*0*	***1***
ks	0	1	0	1	**1**		1	0	0	1	**1**
cmd	0	1	1	0	**1**		1	0	0	1	**0**

‘read inverted’ ‘page 5’ ~‘read page 5 inverted’

DC4420

Find all encrypted ‘read’ commands

cmd	a	**1**	b	c	d	~a	**0**	~b	~c	~d
enc	*E*	***F***	*G*	*H*	*I*	*J*	***K***	*L*	*M*	*N*

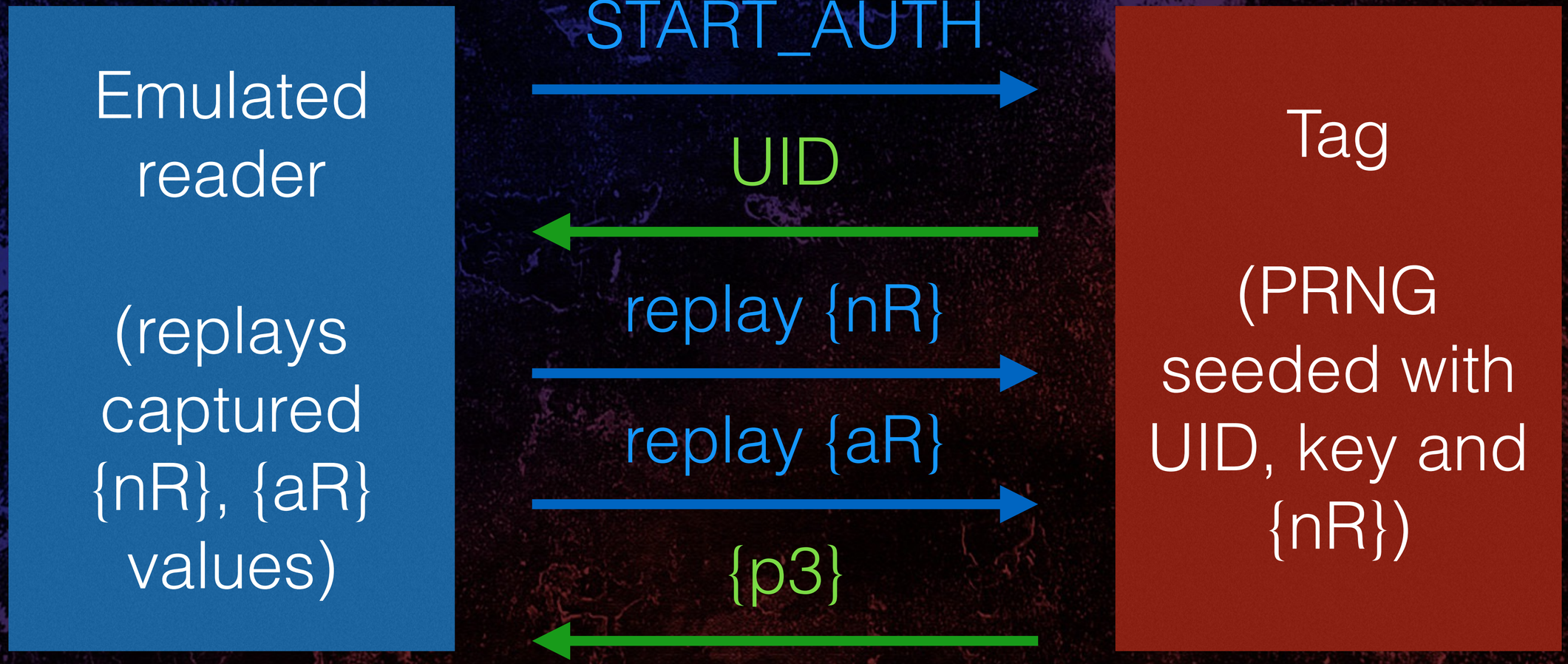
0	*E*	***F***	*G*	*H*	*I*	*J*	***K***	*L*	*M*	*N*
1	*E*	***F***	*G*	*H*	*~I*	*J*	***K***	*L*	*M*	*~N*
2	*E*	***F***	*G*	*~H*	*I*	*J*	***K***	*L*	*~M*	*N*
3	*E*	***F***	*G*	*~H*	*~I*	*J*	***K***	*L*	*~M*	*~N*
4	*E*	***F***	*~G*	*H*	*I*	*J*	***K***	*~L*	*M*	*N*
…	…	**…**	…	…	…	…	**…**	…	…	…
14	*~E*	***F***	*~G*	*~H*	*I*	*~J*	***K***	*~L*	*~M*	*N*
15	*~E*	***F***	*~G*	*~H*	*~I*	*~J*	***K***	*~L*	*~M*	*~N*

DC4420

Find ‘read p0’ command

- for (i=0; i<16; i++) {
  - setup(); [START_AUTH, UID, {nR}, {aR}, {p3}]
  - if (test_guess_and_recover_key_stream(i, ks))
    - return TRUE;
- }
- return FALSE;

Emulate reader



- Output of tag PRNG (key stream) depends solely on {nR}

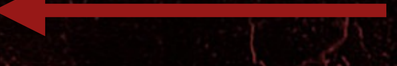
Nonce replay attack

{nR}
0x4A85B2DF

Tag

(PRNG seeded with UID, key and {nR})

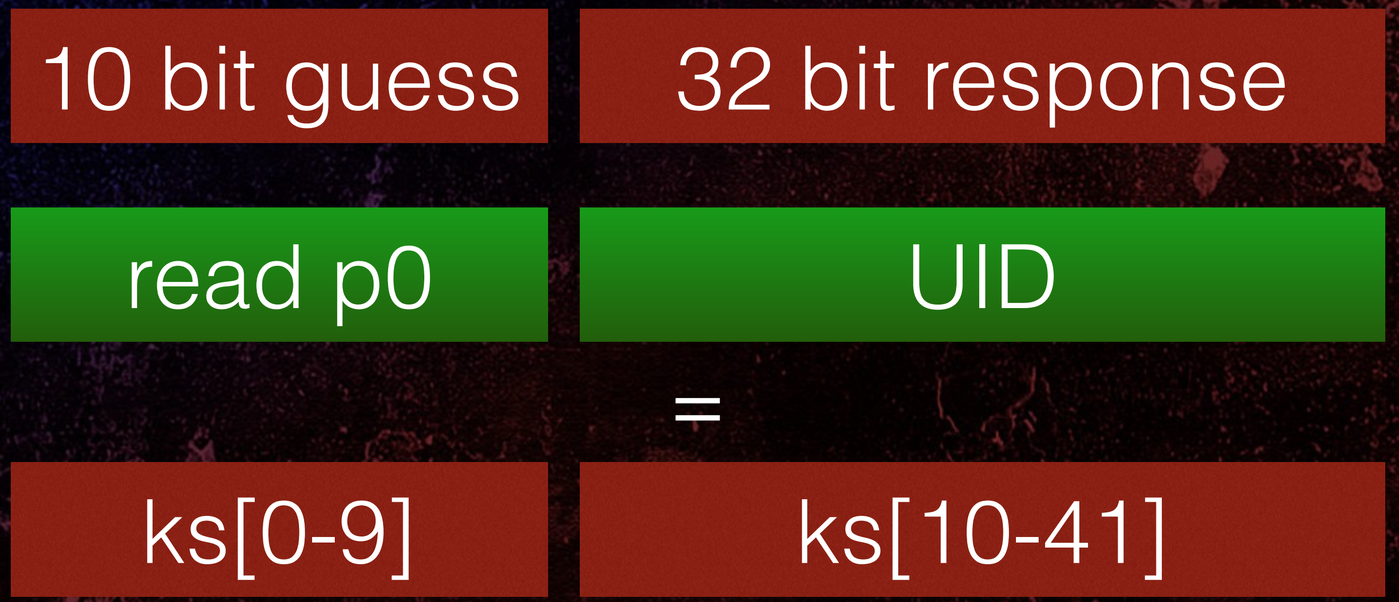
PRNG output:
01011100011101000 1…



- Reset PRNG to same state for every session / guess

Test ‘read p0’ command guesses

- Assume guess is correct
- Setup() [START_AUTH, UID, {nR}, {aR}, {p3}]



(read p0 = 0b11000 00111)

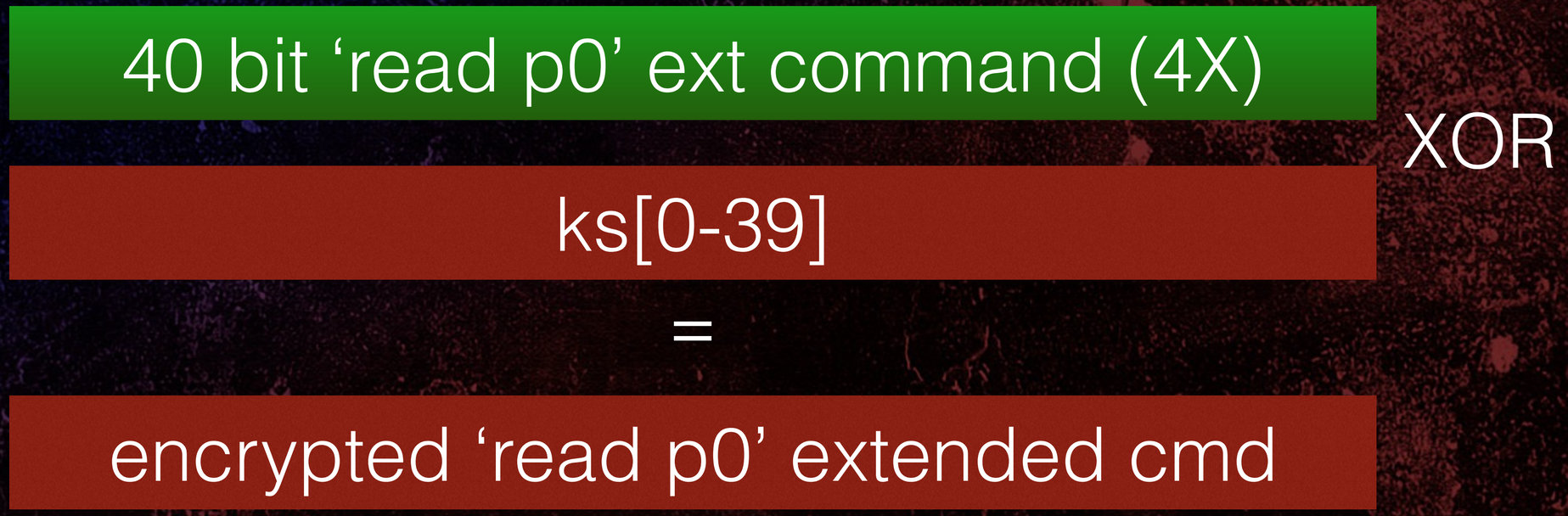
Test ‘read p0’ command guesses

DC4420

- Build 40 bit extended ‘read p0’ command:
  - ext_cmd = 0b1100000111 (‘read p0’) X 4
  - = 0b1100000111 1100000111 1100000111 1100000111

Test ‘read p0’ command guesses

- Assume guess is correct
- Setup() [START_AUTH, UID, {nR}, {aR}, {p3}]



Test ‘read p0’ command guesses

- If guess was correct, response != ERROR_RESPONSE

encrypted ‘read p0’ ext cmd

response

XOR

UID

=

ks[40-71]

ks[0-39] must be valid

Length extension

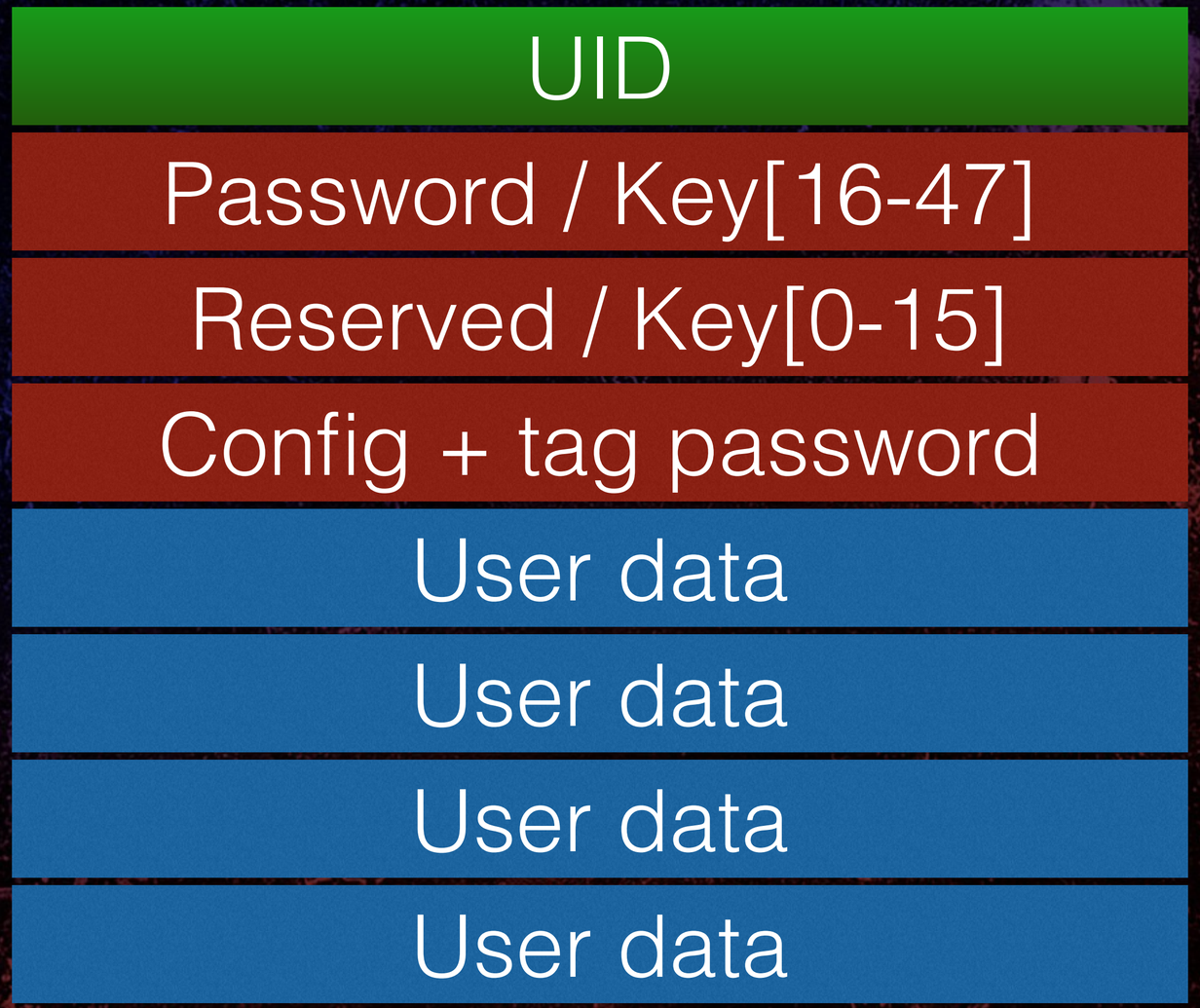
- We now have:
  - A replayable {nR}, {aR} pair (nonce replay)
  - The encrypted ‘read page0’ 40 bit extended command, following setup()
  - The encrypted 32 bit UID (p0) response
  - 72 bits of key stream following setup():
    - 40 bit command + 32 bit response

Read page data

DC4420

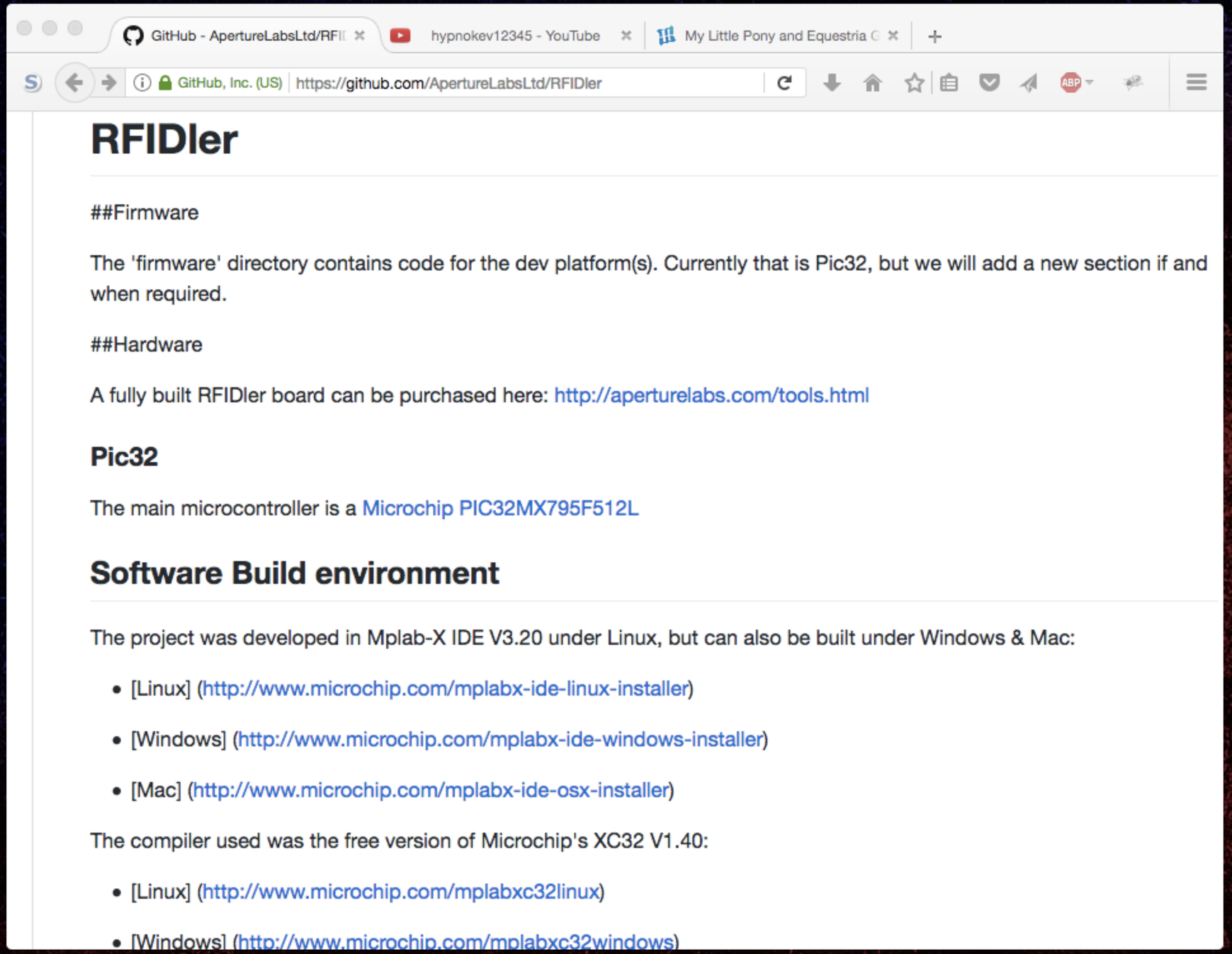
- for (i=0; i<8; i++) {
  - setup(); [START_AUTH, UID, {nR}, {aR}, {p3}]
  - enc_cmd = ‘read page i command’ XOR ks[0-9];
  - enc_response = send(enc_cmd);
  - ‘page i’ = enc_response XOR ks[10-41];
- }

HiTag2 125KHz RFID tags



- 8 pages
- 32 bits each
- UID is public

DC4420



RFIDler

##Firmware

The 'firmware' directory contains code for the dev platform(s). Currently that is Pic32, but we will add a new section if and when required.

##Hardware

A fully built RFIDler board can be purchased here: http://aperturelabs.com/tools.html

Pic32

The main microcontroller is a Microchip PIC32MX795F512L

Software Build environment

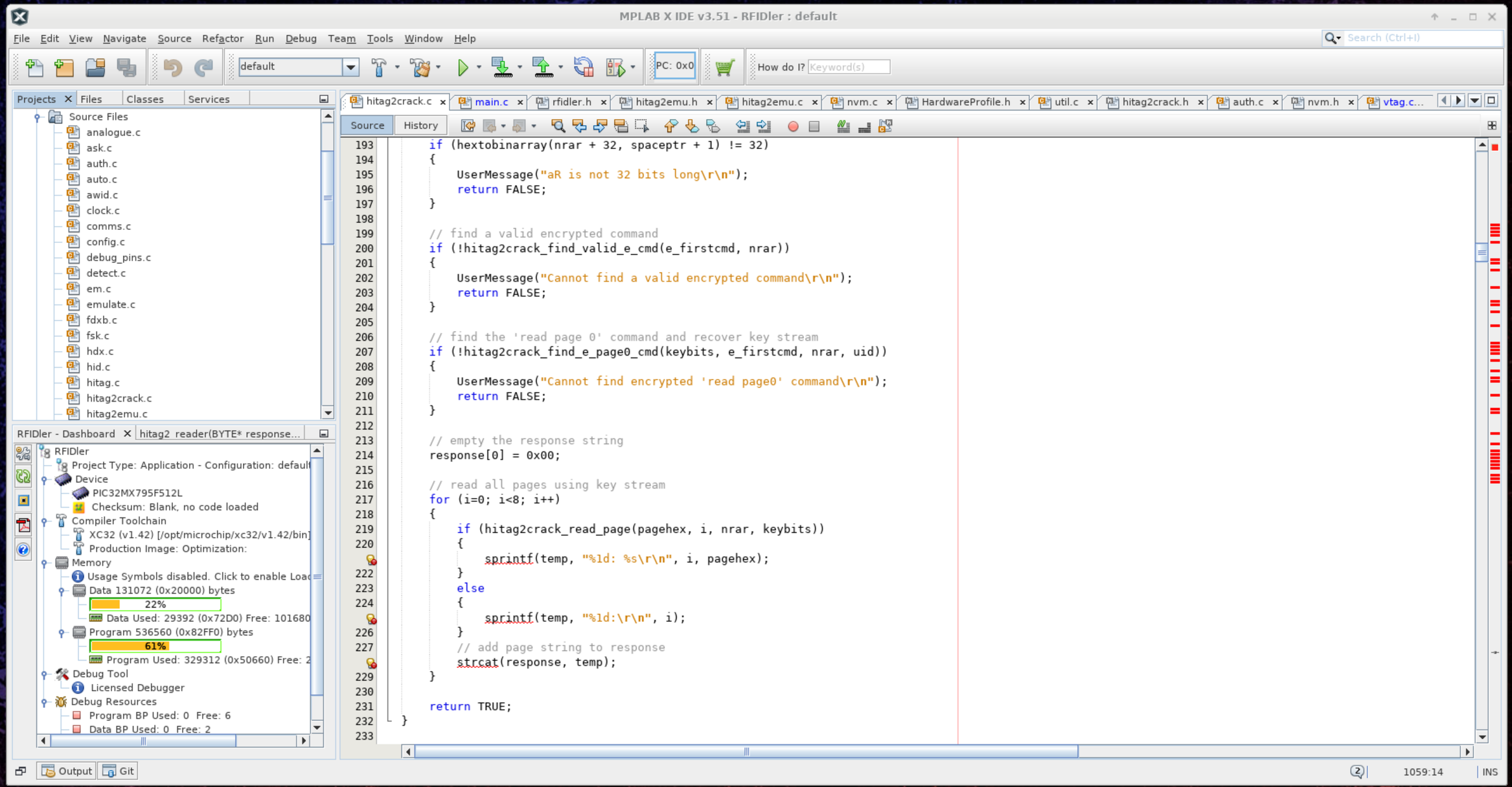
The project was developed in Mplab-X IDE V3.20 under Linux, but can also be built under Windows & Mac:

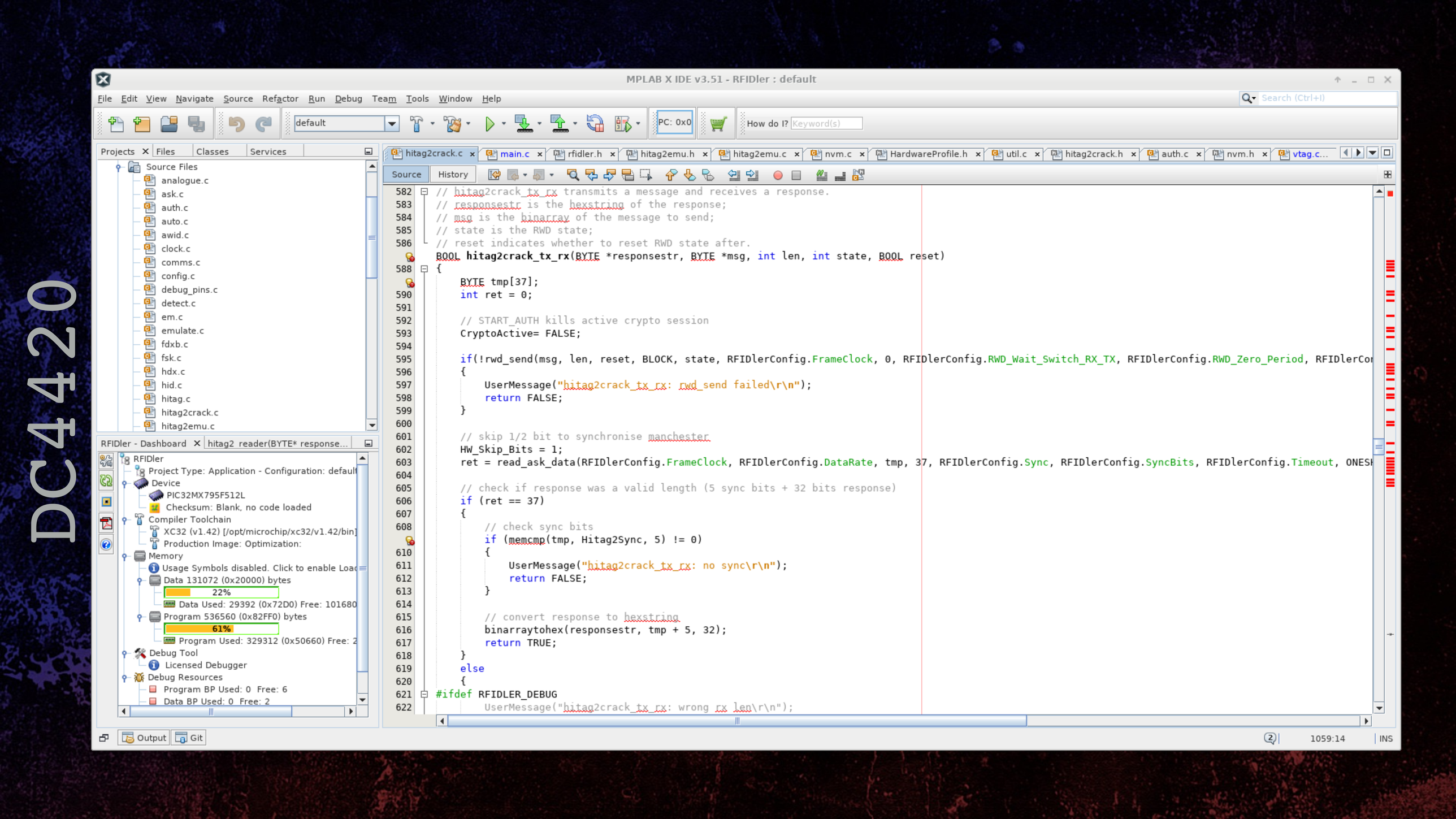
- [Linux] (http://www.microchip.com/mplabx-ide-linux-installer)
- [Windows] (http://www.microchip.com/mplabx-ide-windows-installer)
- [Mac] (http://www.microchip.com/mplabx-ide-osx-installer)

The compiler used was the free version of Microchip's XC32 V1.40:

- [Linux] (http://www.microchip.com/mplabxc32linux)
- [Windows] (http://www.microchip.com/mplabxc32windows)

DC4420


MPLAB X IDE v3.51 - RFIDler : default
File Edit View Navigate Source Refactor Run Debug Team Tools Window Help
default
PC: 0x0
How do I?
Projects Files Classes Services
Source Files
analogue.c
ask.c
auth.c
auto.c
awid.c
clock.c
comms.c
config.c
debug_pins.c
detect.c
em.c
emulate.c
fdxb.c
fsk.c
hdx.c
hid.c
hitag.c
hitag2crack.c
hitag2emu.c
RFIDler - Dashboard
hitag2 reader(BYTE* response...
RFIDler
Project Type: Application - Configuration: default
Device
PIC32MX795F512L
Checksum: Blank, no code loaded
Compiler Toolchain
XC32 (v1.42) [/opt/microchip/xc32/v1.42/bin]
Production Image: Optimization:
Memory
Usage Symbols disabled. Click to enable Load
Data 131072 (0x20000) bytes
22%
Data Used: 29392 (0x72D0) Free: 101680
Program 536560 (0x82FF0) bytes
61%
Program Used: 329312 (0x50660) Free: 2
Debug Tool
Licensed Debugger
Debug Resources
Program BP Used: 0 Free: 6
Data BP Used: 0 Free: 2
hitag2crack.c
main.c
rfidler.h
hitag2emu.h
hitag2emu.c
nvm.c
HardwareProfile.h
util.c
hitag2crack.h
auth.c
nvm.h
vtag.c...
Source History
193 if (hextobinarray(nrar + 32, spaceptr + 1) != 32)
194 {
195 UserMessage("aR is not 32 bits long\r\n");
196 return FALSE;
197 }
198
199 // find a valid encrypted command
200 if (!hitag2crack_find_valid_e_cmd(e_firstcmd, nrar))
201 {
202 UserMessage("Cannot find a valid encrypted command\r\n");
203 return FALSE;
204 }
205
206 // find the 'read page 0' command and recover key stream
207 if (!hitag2crack_find_e_page0_cmd(keybits, e_firstcmd, nrar, uid))
208 {
209 UserMessage("Cannot find encrypted 'read page0' command\r\n");
210 return FALSE;
211 }
212
213 // empty the response string
214 response[0] = 0x00;
215
216 // read all pages using key stream
217 for (i=0; i<8; i++)
218 {
219 if (hitag2crack_read_page(pagehex, i, nrar, keybits))
220 {
222 sprintf(temp, "%1d: %s\r\n", i, pagehex);
223 }
224 else
226 {
227 sprintf(temp, "%1d:\r\n", i);
229 }
230 // add page string to response
231 strcat(response, temp);
232 }
233
return TRUE;
}
Output Git
1059:14
INS


DC4420
MPLAB X IDE v3.51 - RFIDler : default
File Edit View Navigate Source Refactor Run Debug Team Tools Window Help
Projects Files Classes Services
Source Files
analogue.c
ask.c
auth.c
auto.c
awid.c
clock.c
comms.c
config.c
debug_pins.c
detect.c
em.c
emulate.c
fdxb.c
fsk.c
hdx.c
hid.c
hitag.c
hitag2crack.c
hitag2emu.c
RFIDler - Dashboard
hitag2_reader(BYTE* response...
RFIDler
Project Type: Application - Configuration: default
Device
PIC32MX795F512L
Checksum: Blank, no code loaded
Compiler Toolchain
XC32 (v1.42) [/opt/microchip/xc32/v1.42/bin]
Production Image: Optimization:
Memory
Usage Symbols disabled. Click to enable Load
Data 131072 (0x20000) bytes
22%
Data Used: 29392 (0x72D0) Free: 101680
Program 536560 (0x82FF0) bytes
61%
Program Used: 329312 (0x50660) Free: 2
Debug Tool
Licensed Debugger
Debug Resources
Program BP Used: 0 Free: 6
Data BP Used: 0 Free: 2
hitag2crack.c
main.c
rfidler.h
hitag2emu.h
hitag2emu.c
nvm.c
HardwareProfile.h
util.c
hitag2crack.h
auth.c
nvm.h
vtag.c...
Source History
// hitag2crack_tx_rx transmits a message and receives a response.
// responsestr is the hexstring of the response;
// msg is the binarray of the message to send;
// state is the RWD state;
// reset indicates whether to reset RWD state after.
BOOL hitag2crack_tx_rx(BYTE *responsestr, BYTE *msg, int len, int state, BOOL reset)
{
BYTE tmp[37];
int ret = 0;
// START_AUTH kills active crypto session
CryptoActive= FALSE;
if(!rwd_send(msg, len, reset, BLOCK, state, RFIDlerConfig.FrameClock, 0, RFIDlerConfig.RWD_Wait_Switch_RX_TX, RFIDlerConfig.RWD_Zero_Period, RFIDlerCo
{
UserMessage("hitag2crack_tx_rx: rwd_send failed\r\n");
return FALSE;
}
// skip 1/2 bit to synchronise manchester
HW_Skip_Bits = 1;
ret = read_ask_data(RFIDlerConfig.FrameClock, RFIDlerConfig.DataRate, tmp, 37, RFIDlerConfig.Sync, RFIDlerConfig.SyncBits, RFIDlerConfig.Timeout, ONES
// check if response was a valid length (5 sync bits + 32 bits response)
if (ret == 37)
{
// check sync bits
if (memcmp(tmp, Hitag2Sync, 5) != 0)
{
UserMessage("hitag2crack_tx_rx: no sync\r\n");
return FALSE;
}
// convert response to hexstring
binarraytohex(responsestr, tmp + 5, 32);
return TRUE;
}
else
{
#ifdef RFIDLER_DEBUG
UserMessage("hitag2crack_tx_rx: wrong rx len\r\n");
Output Git
1059:14
INS

New RFIDler commands

- SNIFF-PWM <C|S|L> - HiTag2 clear/store/list {nR},{aR}
- HITAG2-CRACK <{NR}> <{AR}> - Attack 1
- HITAG2-KEYSTREAM <{NR}> <{AR}> - for Attack 2
- HITAG2-READER <KEY> [S] - read HiTag2 tags
- HITAG2-CLEARSTOREDTAGS
- HITAG2-COUNTSTOREDTAGS
- HITAG2-LISTSTOREDTAGS [START] [END]

Nonce replay and length extension

- SET TAG HITAG2
- SNIFF-PWM C - initialises {nR}/{aR} storage
- AUTORUN SNIFF-PWM S - sets autorun function
- SAVE - stores current tag type and autorun command
- Power RFIDler from USB battery
- RFIDler antenna on RWD, present tag (n times)
- SNIFF-PWM L - list stored {nR}/{aR} values
- HITAG2-CRACK <{nR}> <{aR}>

Demo

Nonce replay and length extension attack

DC4420

Tag cloning

- With the key it is easy to copy a tag
  - Emulate reader
  - Auth to tag with key, dump all 8 pages and copy to VTAG (HITAG2-READER)
  - Write to new tag (CLONE)
- UID (p0) is fixed but this is rarely a problem
- Can be done at distance with right equipment

However

DC4420

- Pages 1 & 2 can be read-protected
- Stops reading these pages even with the key stream
- Need other attacks to recover key (pages 1 & 2)
- Can then authenticate to tag
- Tag cloning then possible again

DC4420

Intermission

DC4420

Time/memory trade off key recovery

Key space

DC4420

- Key is 48 bits long
- Key space is 2^48
- Protocol is far too slow for on-line brute force
- Key space ~~is~~ was far too big to build a table or conduct off-line brute force
- Time/memory trade off ~~is~~ was required.

Time/memory trade off

- Build table of 2^37 entries
  - 2^37 = 2^48 / 2^11 (2048)
  - PRNG state, 48 bits of PRNG output
- Recover 2048 bits of key stream (PRNG output)
- Search for key stream matches in table
- Recover PRNG state
- Roll back to recover key

Time/memory trade off

- Build table (once only):
  - Start with random 48 bit PRNG state
  - For 2^37 times:
    - Store 48 bit PRNG state
    - Generate 48 bits of key stream and store it
    - Advance from stored PRNG state by 2048 states
  - Sort table on 48 bit key stream entries

Efficient LFSR n-step jumps

DC4420

and the transponder. The next proposition introduces a small trick that makes it possible to quickly perform n cipher steps at once. Intuitively, this proposition states that the linear difference between a state s and its n-th successor is a combination of the linear differences generated by each bit. This will be later used in the attack.

Proposition 5.1. *Let s be an LFSR state and $n \in \mathbb{N}$. Furthermore, let $d_i = \mathrm{suc}^n(2^i)$ i.e., the LFSR state that results from running the cipher n steps from the state 2^i. Then*

$$\mathrm{suc}^n(s) = \bigoplus_{i=0}^{47} (d_i \cdot s_i) .$$

Efficient LFSR n-step jumps

cipher steps at once. Intuitively, this proposition states that the linear difference between a state s and its n-th successor is a combination of the linear differences generated by each bit. This will be later used in the attack.

- Find nth-successor of each of the 48 states where only one bit is a 1
- Use XOR to combine the nth-successors of those that represent the bit mask of the starting state, s

Efficient LFSR n-step jumps

- d_0 = nth-succ(000 … 0001)
- d_1 = nth-succ(000 … 0010)
- d_2 = nth-succ(000 … 0100)

- If s = 011 … 1110:
  - nth-succ(s) = d_{46} XOR d_{45} XOR … XOR d_3 XOR d_2 XOR d_1

Efficient LFSR n-step jumps

Proposition 5.1. *Let s be an LFSR state and $n \in \mathbb{N}$. Furthermore, let $d_i = \mathrm{suc}^n(2^i)$ i.e., the LFSR state that results from running the cipher n steps from the state 2^i. Then*

$$\mathrm{suc}^n(s) = \bigoplus_{i=0}^{47} (d_i \cdot s_i) .$$

- d_i is the nth-successor from the state where bit i is a 1 and all other bits are 0
  - d_i is a 48 bit state
- s_i is the ith bit of starting state, s
  - s_i is a single bit

DC4420

Efficient LFSR n-step jumps

- Build table d once:
  - statemask = 1;
  - for (i=0; i<48; i++) {
    - d[i] = lfsr_nstep(statemask, nsteps);
    - statemask = statemask << 1;
  - }

Efficient LFSR n-step jumps

- Jump n-steps using table d:
  - bitmask = 1; output = 0;
  - for (i=0; i<48; i++) {
    - if (LFSR & bitmask)
      - output = output XOR d[i];
    - bitmask = bitmask << 1;
  - }

Efficient LFSR n-step jumps

- Building table takes (48 * nsteps) cipher ticks
  - Once only
  - <100,000 cipher ticks (for nsteps = 2048)
- Jumping nsteps using table d takes <= 48 XORs
- Jumping nsteps without table d takes nsteps cipher ticks
  - Done lots and lots of times - 137 billion times in this case!

Building time/memory table

- Divide into multiple threads, one for each core
- Interleave threads to save effort
- Start each thread 2048 states apart
- Jump (2048 * #threads) states each entry
- 2 n-step jump tables:
  - n=2048 for starting points
  - n=(2048 * #threads) for each entry

Building time/memory table

- Need an efficient way to generate table and sort it
- Table is sparse so cannot sort while generating
- My (non-optimised) solution:
  - Generate table
  - Sort on first two bytes of key stream
  - Sort sub-tables
- Recommend directly connected SATA disks

Time/memory trade off

- Using nonce replay and length extension, recover 2048 bits of key stream from tag
- Using sliding window, search for all 48 bit windows of key stream in table
- Test matches by seeding PRNG with recovered state and generating the following (or preceding) key stream
- When PRNG state matches, roll back to setup state

Recover 2048 bits of key stream

DC4420

- Use attack 1, extended to 2048 bits
- On RFIDler, buffer size limits command to 512 bits
- Iterative two-stage process:
  - setup(); [START_AUTH, UID, {nR}, {aR}, {p3}]
  - Consume existing key stream (leaving >=10 bits)
  - Extend key stream by 32 bits

Recover 2048 bits of key stream

- e.g. With 640 bits of key stream:
  - Send 510 bit command
    - Receive 32 bit response
  - Send 50 bit command
    - Receive 32 bit response
  - Send 10 bit command
    - Extend key stream with 32 bit response

Sliding window

2048 bits of recovered key stream

48 bit slice

48 bit slice

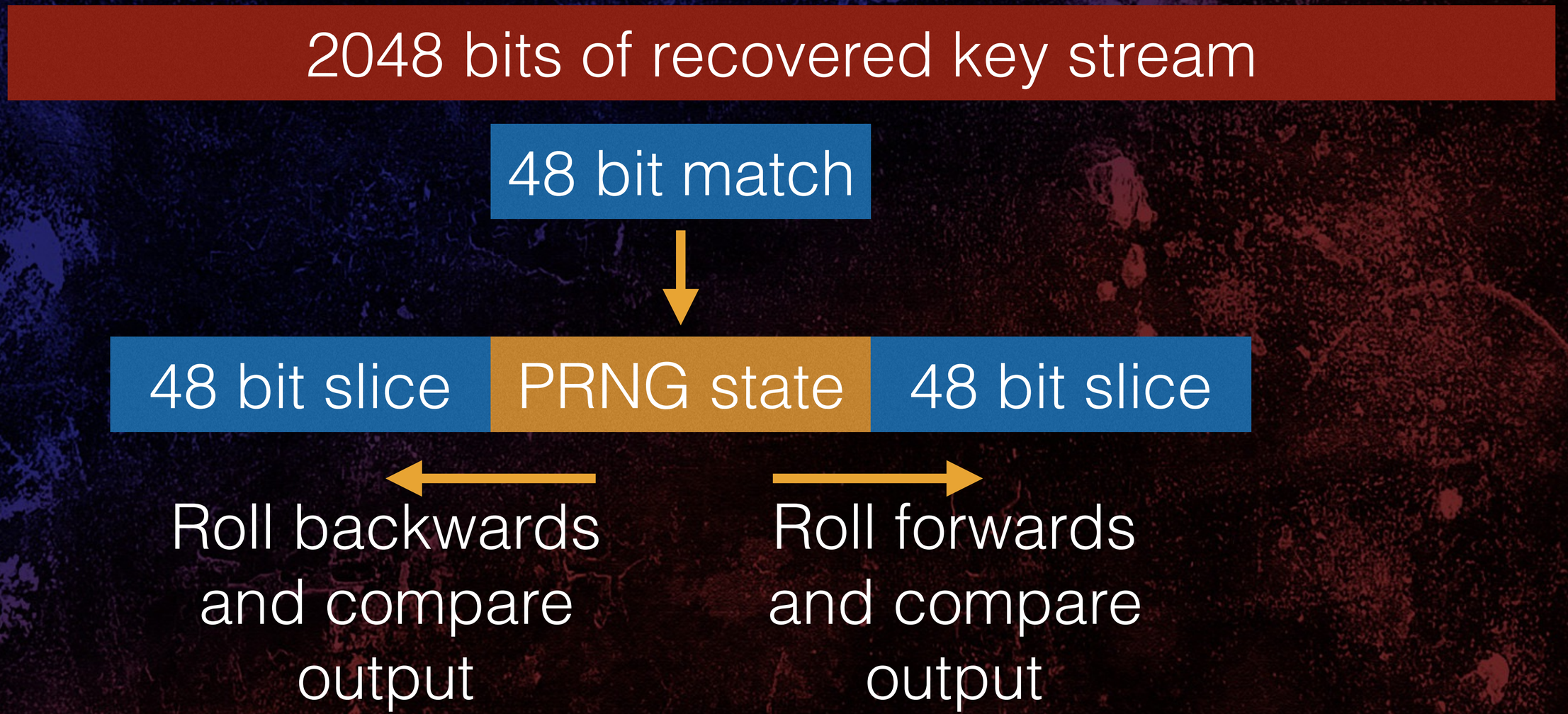
48 bit slice

...

48 bit slice

48 bit slice

Test match



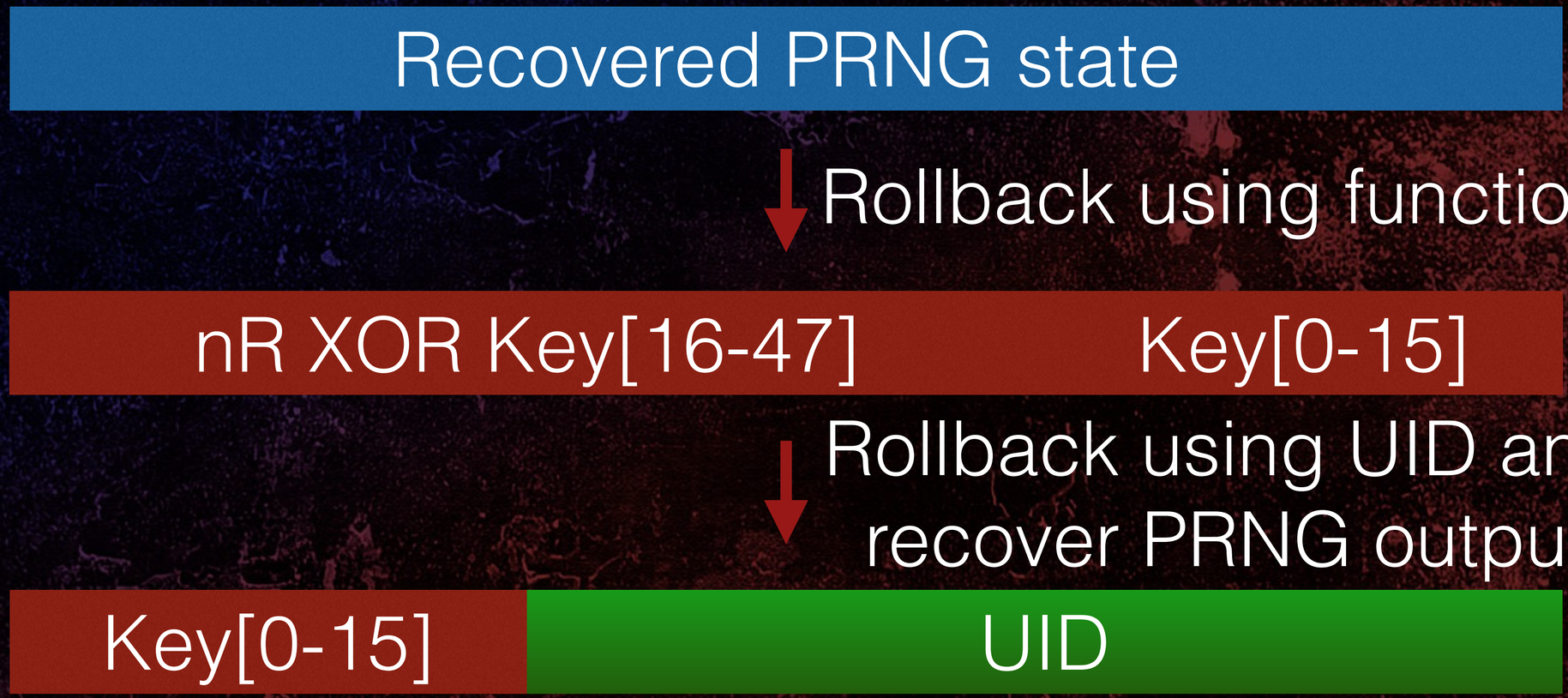
Rollback function, R()

DC4420

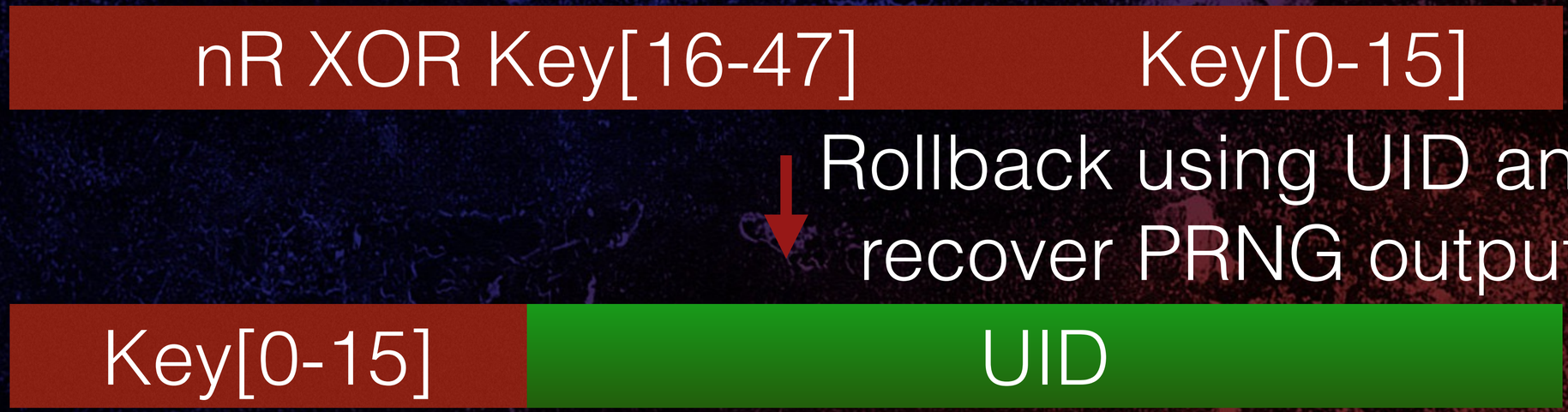
- Calculates the bit that is shifted out of LFSR.
- Treating the LFSR state as an array of bits…
- R(x) = XOR(x[1], x[2], x[5], x[6], x[7], x[15], x[21], x[22], x[25], x[29], x[40], x[41], x[42], x[45], x[46], x[47]);
- Allows us to go backwards in time.

PRNG rollback

- Recovered PRNG state at known key stream offset:



Key recovery



- nR = {nR} XOR (PRNG output)
- Key[16-47] = (nR XOR Key[16-47]) XOR nR
- Key = Key[0-15] || Key[16-47]

Time/memory trade off challenges

- Resultant table is 1.5TB
- Takes 1 day to build and >1 day to sort
- Understanding the ‘little trick’ to jump n cipher steps took effort
- Implementing rollback required understanding the PRNG intimately (and building my own reference)
- Takes seconds to recover a key

New RFIDler commands

- SNIFF-PWM <C|S|L> - HiTag2 clear/store/list {nR},{aR}
- HITAG2-CRACK <{NR}> <{AR}> - Attack 1
- HITAG2-KEYSTREAM <{NR}> <{AR}> - for Attack 2
- HITAG2-READER <KEY> [S] - read HiTag2 tags
- HITAG2-CLEARSTOREDTAGS
- HITAG2-COUNTSTOREDTAGS
- HITAG2-LISTSTOREDTAGS [START] [END]

Time/memory trade off key recovery

DC4420

- ./ht2crack2buildtable - once only, takes a while
- RFIDler: UID
- RFIDler: SNIFF-PWM L - list stored {nR}/{aR} values
- RFIDler: HITAG2-KEYSTREAM <{nR}> <{aR}>
- Copy/paste key stream into file
- ./ht2crack2search <key stream file> <UID> <{nR}>

Demo

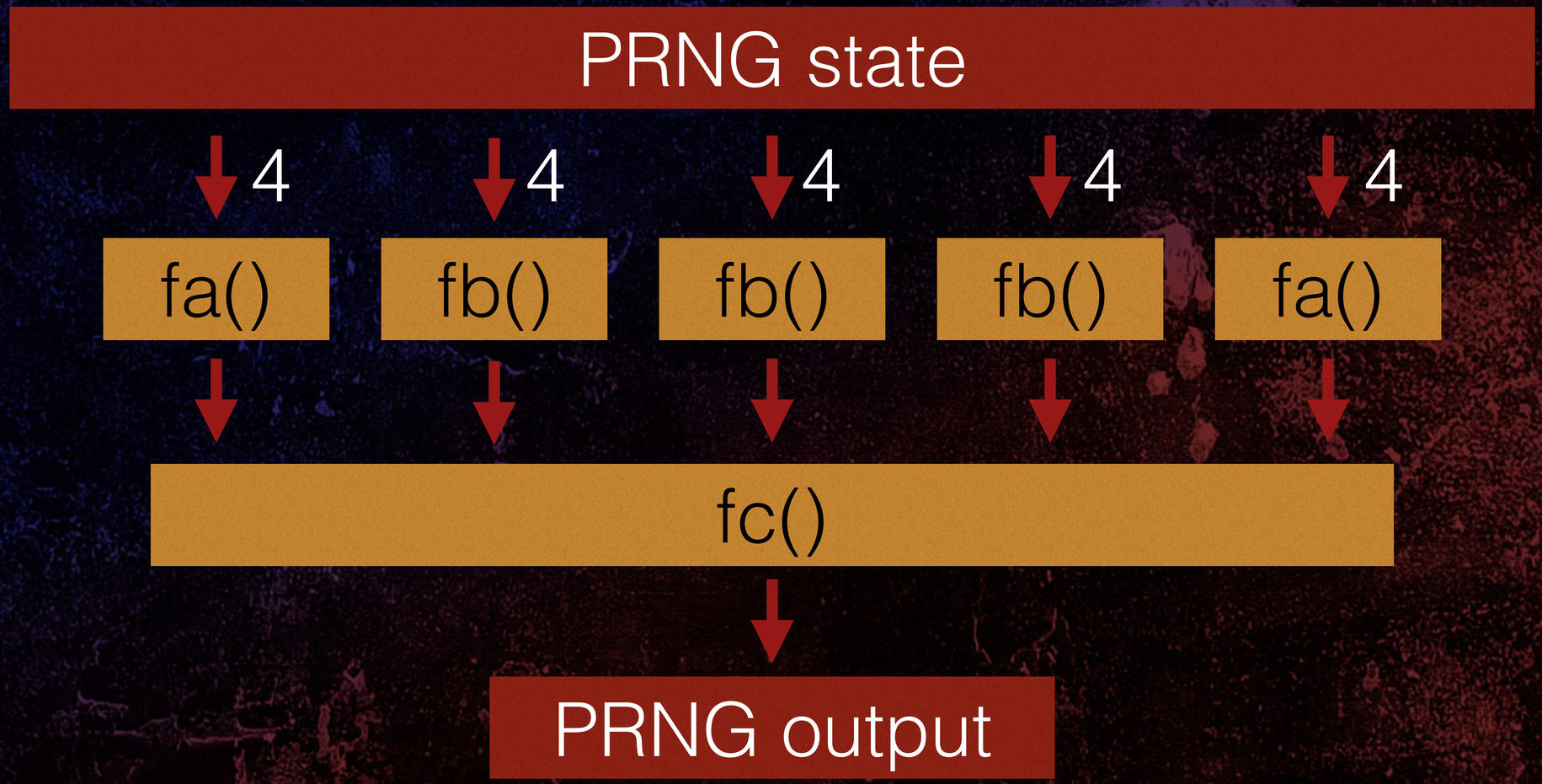
DC4420

Time/memory trade off key recovery attack

DC4420

Cryptanalytic attack for key recovery

Filter function, f()



Filter function, f()

- Produces the output bit from the PRNG
- f(x) = **fc**(

  fa(x[45], x[43], x[42], x[33]),

  fb(x[32], x[30], x[28], x[27]),

  fb(x[25], x[22], x[20], x[16]),

  fb(x[14], x[13], x[11], x[7]),

  fa(x[5], x[4], x[2], x[1]));

Filter function, f(), pre-shift

- Produces the output bit from the PRNG
- f(x) = **fc**(

  fa(x[46], x[44], x[43], x[34]),

  fb(x[33], x[31], x[29], x[28]),

  fb(x[26], x[23], x[21], x[17]),

  fb(x[15], x[14], x[12], x[8]),

  fa(x[6], x[5], x[3], x[2]));

Filter functions, fa(), fb(), fc()

- fa(), fb() and fc() use boolean tables
- fa() and fb() return 1 bit from a 16 bit number:
- fa(a, b, c, d) = bitn(**0x2C79,** abcd);
- fb(a, b, c, d) = bitn(**0x6671**, abcd);

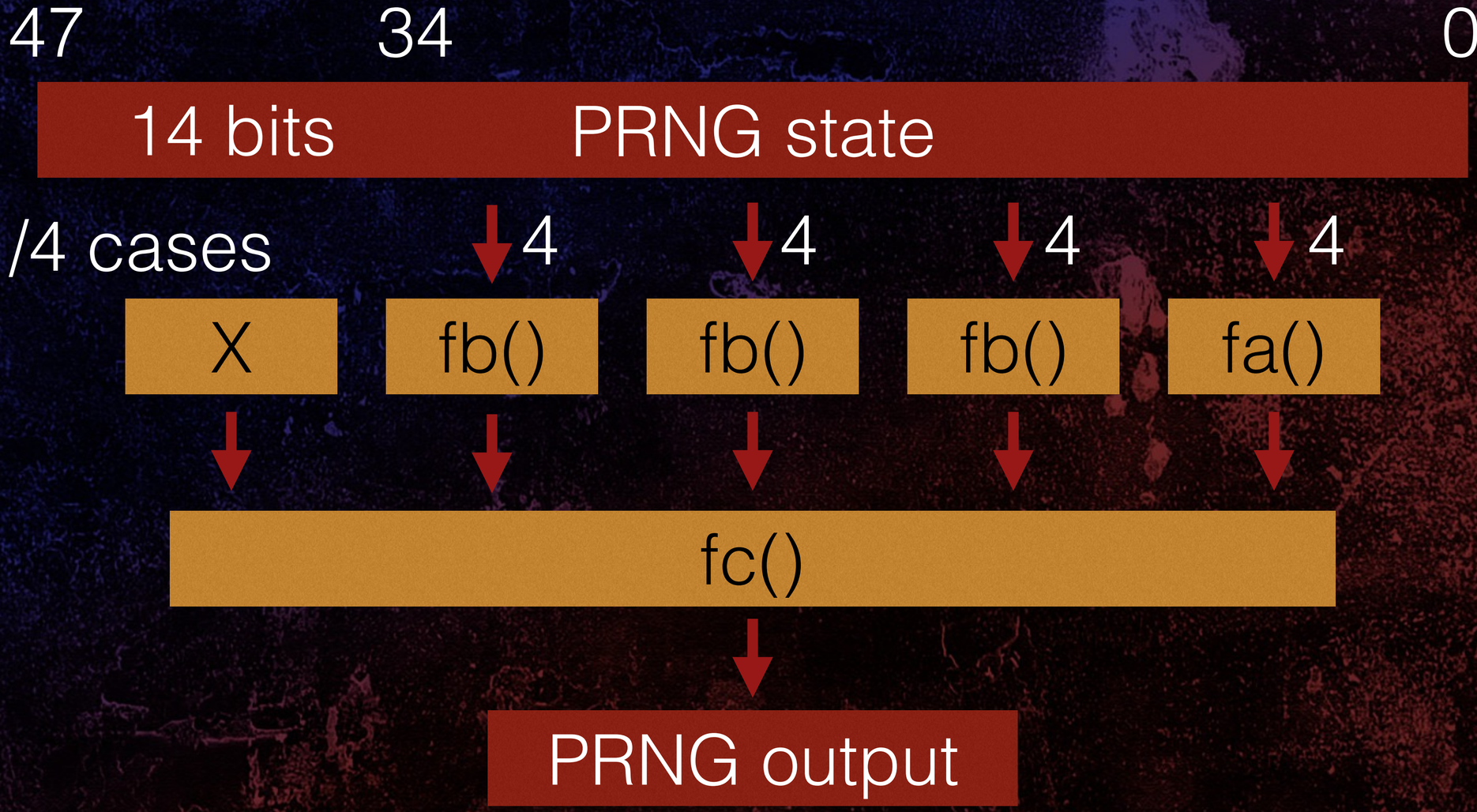
- fc() returns 1 bit from a 32 bit number
- fc(a, b, c, d, e) = bitn(**0x7907287B**, abcde);

Filter function, fc()

- fc(a, b, c, d, e) = bitn(**0x7907287B**, abcde);
- In 1/4 cases, most sig. input bit to fc() is irrelevant
  - **0**111 **100**1 **0**000 01**11**
  - **0**01**0** **100**0 **0**111 10**11**
- Most sig. input bit is generated by

  fa(x[46], x[44], x[43], x[34])
- Therefore, in 1/4 cases, bits 34-47 of LFSR are irrelevant to PRNG output

Filter function, f()



Function P()

- Tests if most sig. input bit to fc() is irrelevant
- P(x) = **fp**(

  fb(x[33], x[31], x[29], x[28]),

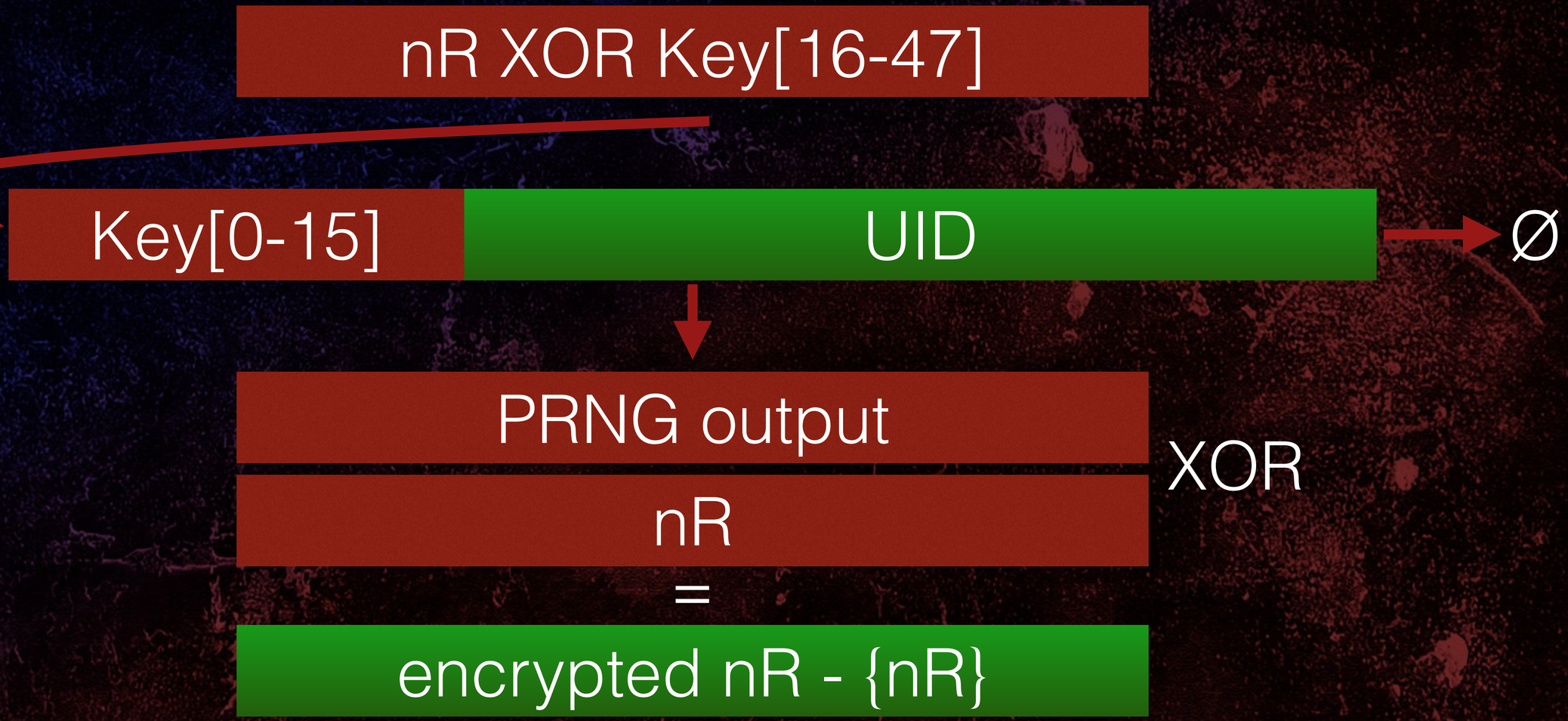
  fb(x[26], x[23], x[21], x[17]),

  fb(x[15], x[14], x[12], x[8]),

  fa(x[6], x[5], x[3], x[2]));
- fp(a, b, c, d) = bitn(**0xAE83**, abcd);

HiTag2 encryption

- Encrypt nR by XORing it with PRNG output



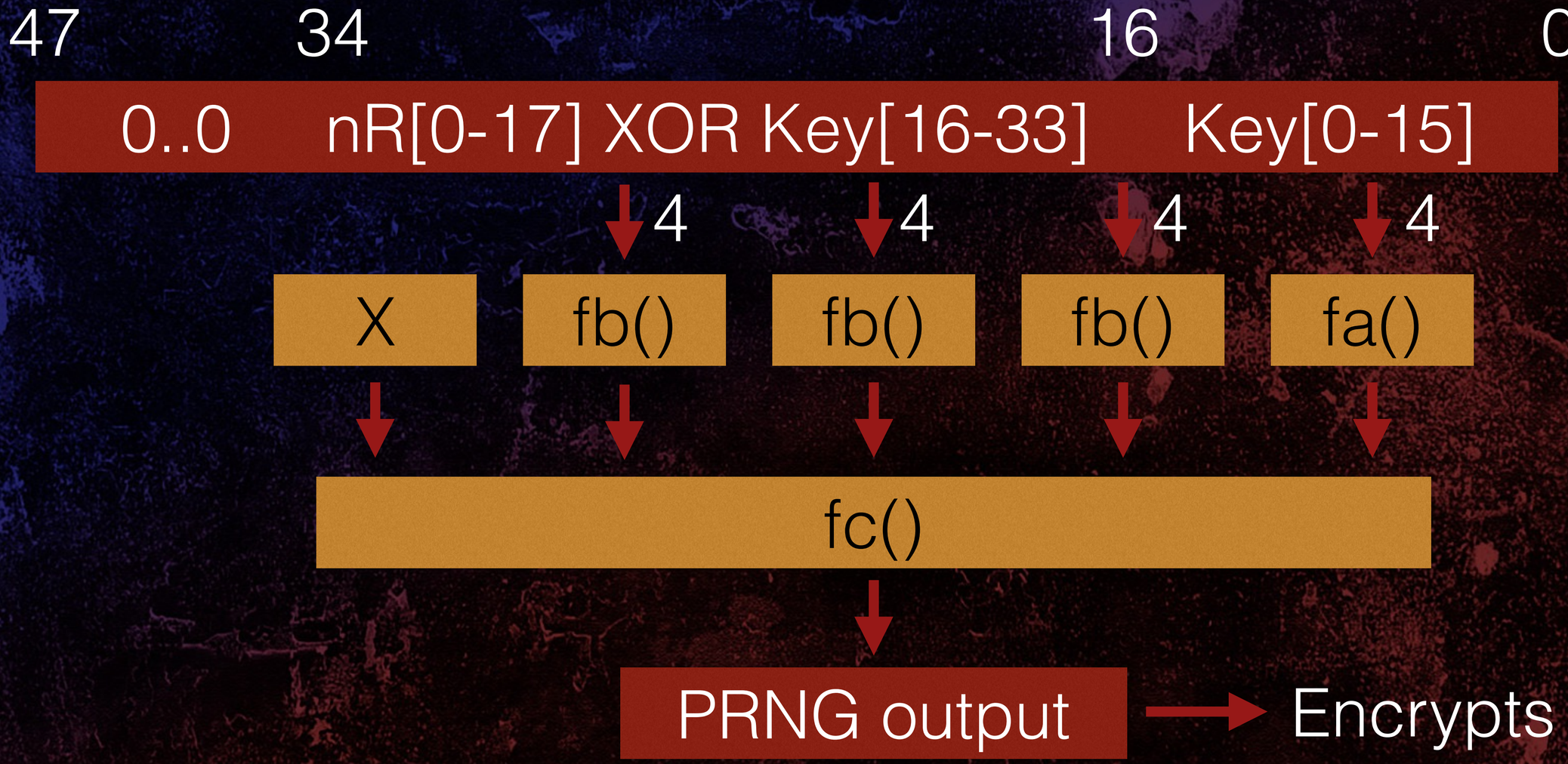
Approach to attack

- nR is encrypted as k[16-47] XOR nR is pushed into the PRNG
- aR is encrypted next, using feedback function L()
- First bit of aR is encrypted when PRNG state contains k[16-47] XOR nR || k[0-15]
- If P(this state) succeeds then upper 14 bits of k[16-47] XOR nR doesn’t affect encryption of first bit of aR => only lower 18 bits are relevant

Approach to attack

- Given a k[0-15]
- Find values for k[16-47] XOR nR that satisfy P(x)
- Implies k[34-47] XOR nR[18-31] does not affect encryption of first bit of aR
- Can push in first 18 bits of k[16-47] XOR nR and extract key stream to encrypt first 18 bits of nR,
- Then push in 14 0s and encrypt first bit of aR

HiTag2 encryption



Approach to attack

DC4420

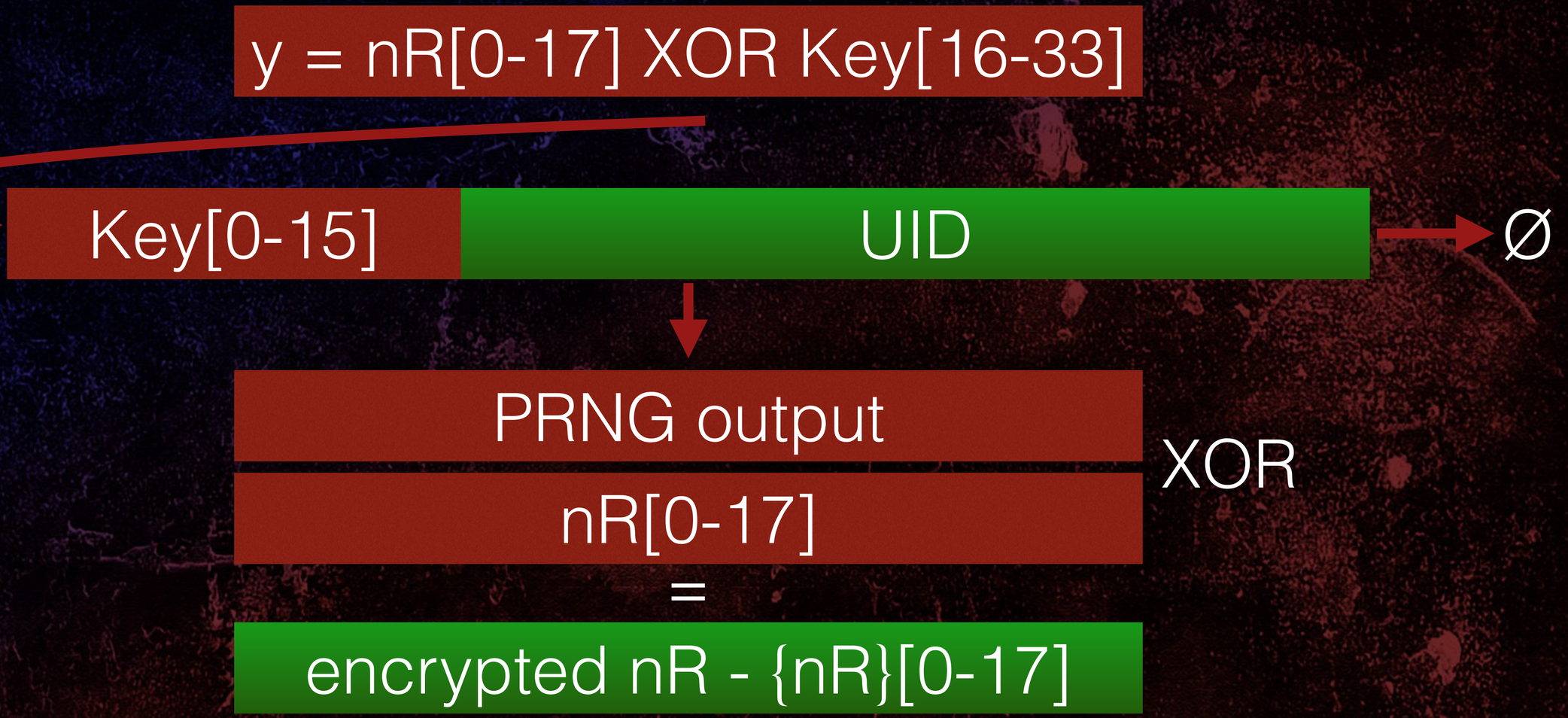
- For a given k[0-15], we can therefore:
  - Find candidates for k[16-33] XOR nR[0-17]
  - Test whether they encrypt first bit of aR correctly
  - Reduce the set of potential k[16-33]
  - Test surviving candidates by brute forcing remaining k[34-47]

Approach to attack

- Capture lots of {nR} {aR} pairs (136 total)
- Divide key into k[0-15], k[16-33] and k[34-47]
- Loop over all k[0-15]
  - Find all y (18 bits) (= k[16-33] XOR nR[0-17])
    - where P(y || k[0-15]) = 1
  - Test all {nR} {aR} pairs to find potential k[16-33]
  - Brute force k[34-47]

HiTag2 encryption

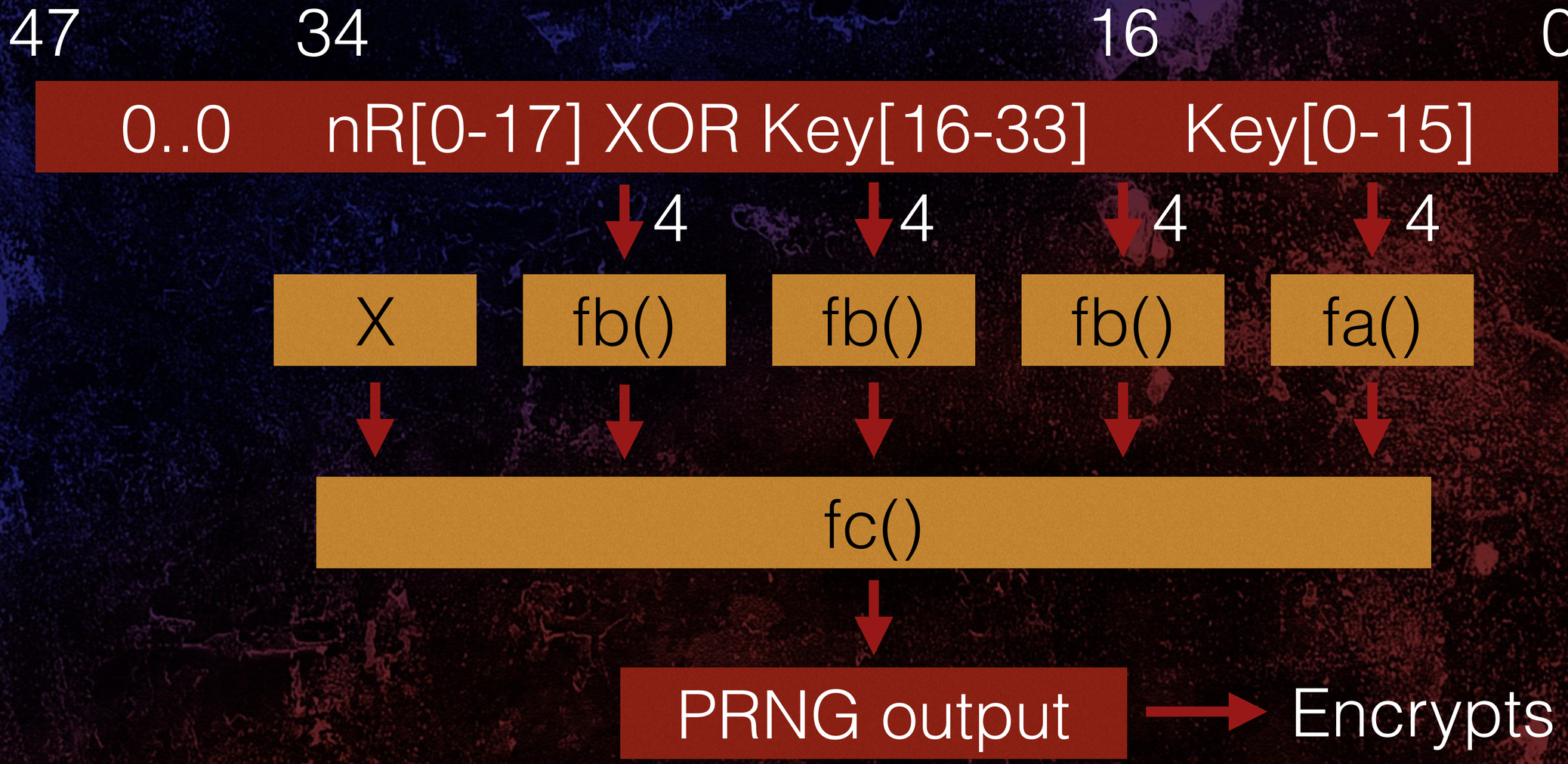
- Encrypt 18 bits of nR by XORing it with PRNG output



Attack details

- Working with first 18 bits of this process
- y = k[16-33] XOR nR[0-17]
- {nR}[0-17] = nR[0-17] XOR output[0-17]
- Therefore:
- nR[0-17] = {nR}[0-17] XOR output[0-17]
- y = k[16-33] XOR {nR}[0-17] XOR output[0-17]
- y XOR output[0-17] = k[16-33] XOR {nR}[0-17]

HiTag2 encryption



Attack details

- aR = 0b11111111111111111111111111111111
- {aR} = aR XOR output[32-63]
- {aR} = ~output[32-63]
- {aR}[0] = ~output[32]

- Because P(y || k[0-15]) = 1, shift in 0s => output[32]
- Build table of (y XOR output[0-17]), ~output[32]

Attack details

- Recall:
  - y XOR output[0-17] = k[16-33] XOR {nR}[0-17]
  - ~output[32] = {aR}[0]
- Search for k[16-33], testing all {nR} {aR} pairs:
  - z = k[16-33] XOR {nR}[0-17]
  - If (z == (y XOR output[0-17])) (z is in table)
    - If (~output[32] != {aR}[0]) => k[16-33] is BAD

Attack details

DC4420

- For each k[0-15]
  - Build table of (y XOR output[0-17], ~output[32])
  - Find small set of potential k[16-33]
    - Brute force k[34-47] (14 bits)
      - Test against 2 x {nR} {aR}

Complexity

- For 2^16 tables:
  - 2^18 encryptions + 2^18 table lookups
  - 2^14 brute force (negligible)
- 2^16 X (2^18 + 2^18) = 2^35 operations
- Much smaller than 2^48 operations
- On quad-core Mac takes 16 mins on average
- Could improve with optimisations

Cryptanalysis challenges

DC4420

- By far the most difficult to understand
- Can’t code it unless it is fully understood
- Had to contact one of the paper authors over Prob(P(x) == 1) = 1/4 or 1/2
- Issue over endianness

New RFIDler commands

- SNIFF-PWM <C|S|L> - HiTag2 clear/store/list {nR},{aR}
- HITAG2-CRACK <{NR}> <{AR}> - Attack 1
- HITAG2-KEYSTREAM <{NR}> <{AR}> - for Attack 2
- HITAG2-READER <KEY> [S] - read HiTag2 tags
- HITAG2-CLEARSTOREDTAGS
- HITAG2-COUNTSTOREDTAGS
- HITAG2-LISTSTOREDTAGS [START] [END]

Cryptanalysis key recovery attack

- SET TAG HITAG2
- SNIFF-PWM C - initialises {nR}/{aR} storage
- AUTORUN SNIFF-PWM S - sets autorun function
- SAVE - stores current tag type and autorun command
- Power RFIDler from USB battery
- RFIDler antenna on RWD, present tag (>=136 times)

Cryptanalysis key recovery attack

DC4420

- SNIFF-PWM L - list stored {nR}/{aR} values
- UID
- Copy/paste {nR}/{aR} values to file
- ./ht2crack3 <UID> <{nR} {aR} file>

Demo

Cryptanalysis key recovery attack

DC4420

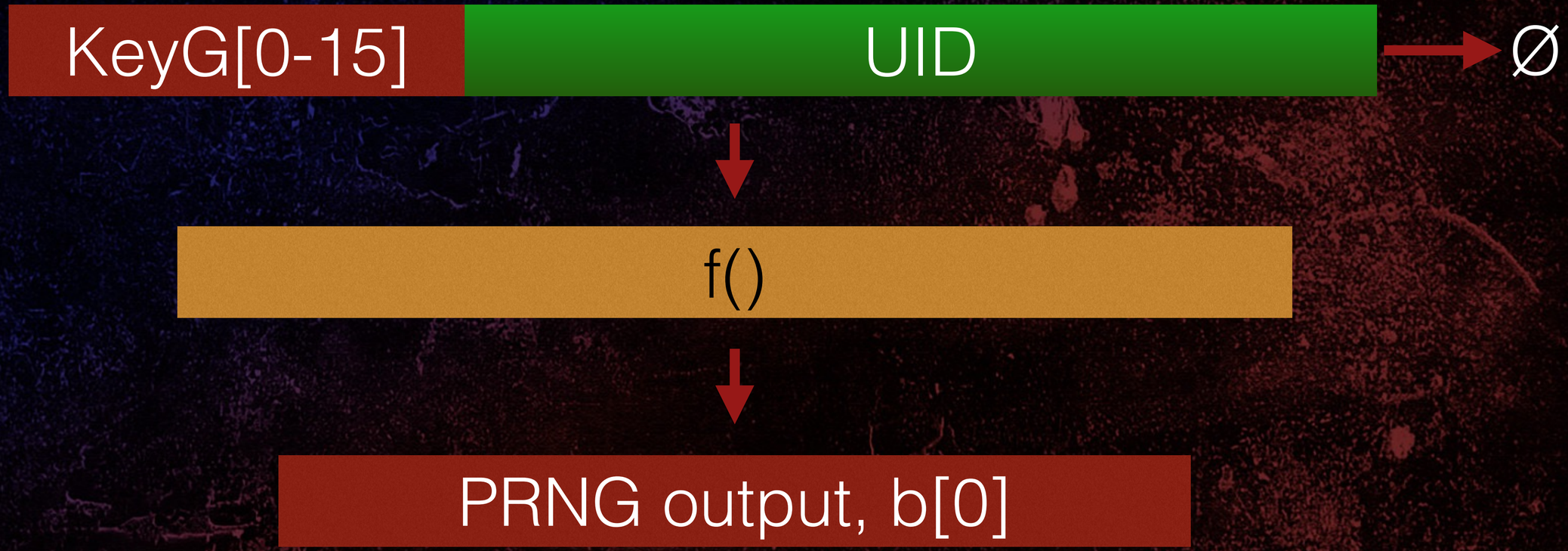
Fast correlation attack

Fast correlation attack

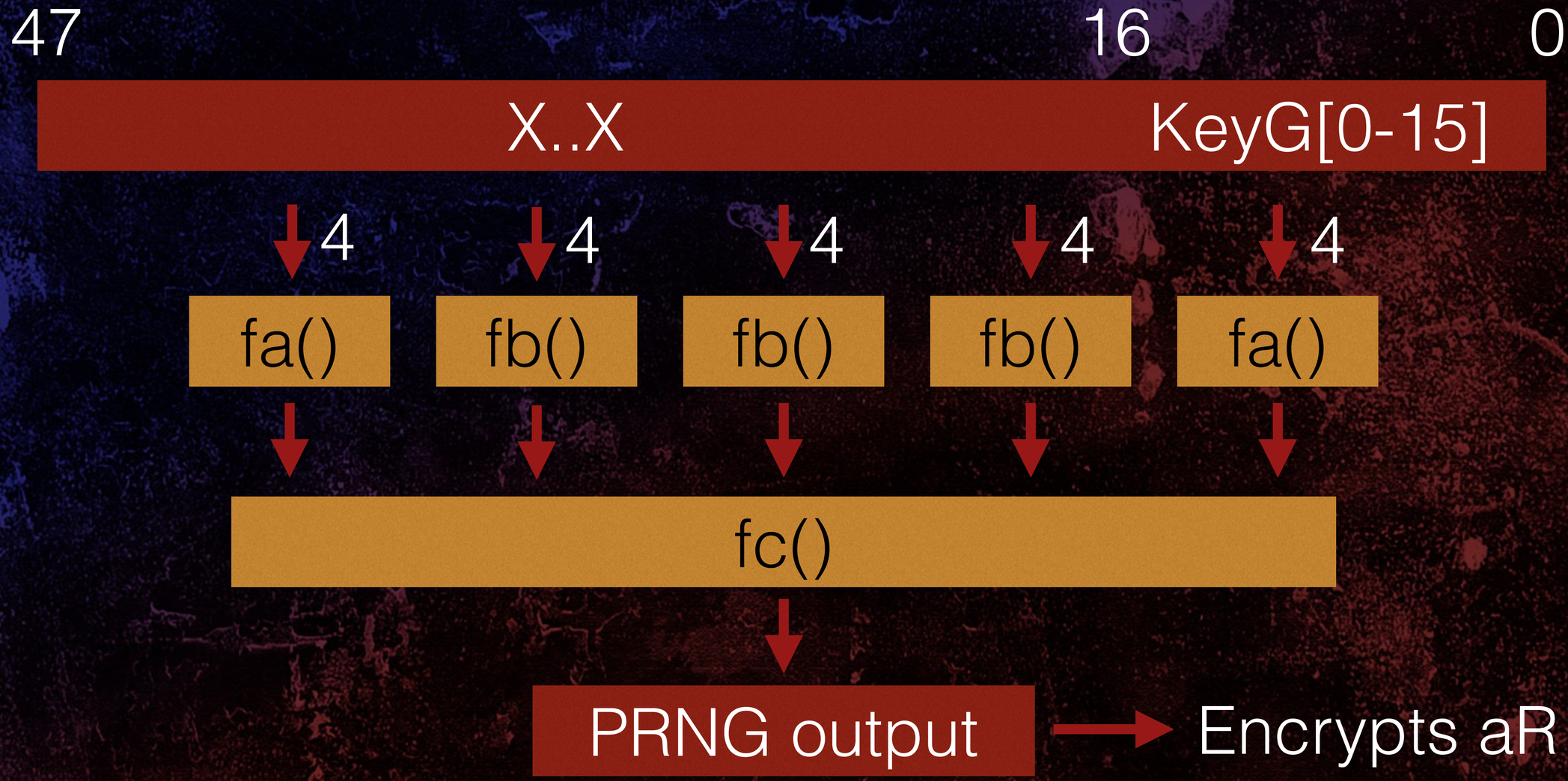
- Lock It And Still Lose It - Garcia et al
- Uses fewer {nR}, {aR} values - 4 to 32
- Much faster than attack 3
- Starts with guesses for first 16 bits of key
- Scores and orders them based on probability
- Expands best guesses by 1 bit and iterates until full key is recovered

Round 0

- Initial state produces first bit of PRNG output



Round 0



Scoring

- Calc probability of predicting first bit of {aR}
- Shift state by 1 bit:
  - Calc probability on second bit of {aR}
  - Fewer bits => less certainty
- Shift state again (repeatedly until no bits left)
  - Even fewer bits => even less certainty
- Combine probabilities for total score

Scoring 2

DC4420

- Repeat scoring for all {nR}, {aR} pairs
- Average scores for guess
- Sort table on scores
- Expand guesses by 1 bit
  - Each guess becomes 2 new guesses

Scoring 3

DC4420

- Each round doubles the number of guesses
- Total number of guesses constrained by table size
- Only the best guesses survive the cut and get expanded into the next round

Scoring 4

DC4420

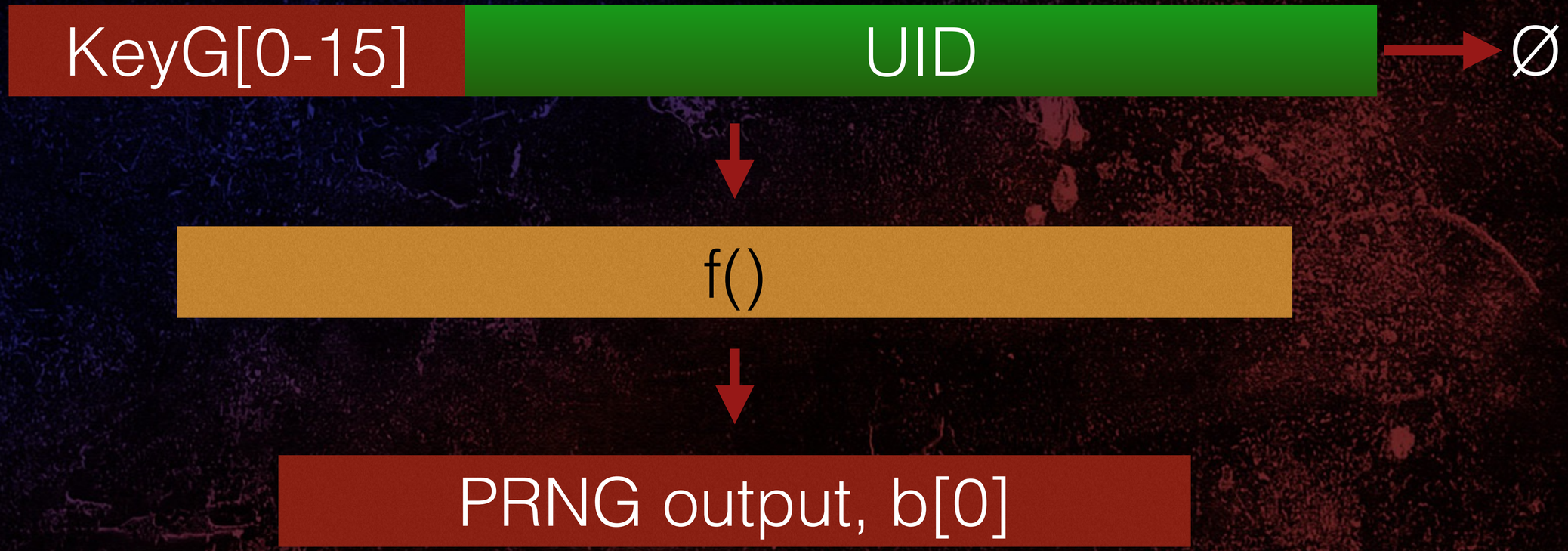
- bit_score = probability of the relevant bits producing the correct output bit
- Pre-compute tables of probabilities for speed and ease
- Use maths to combine probabilities based on known and unknown bits

Scoring 5

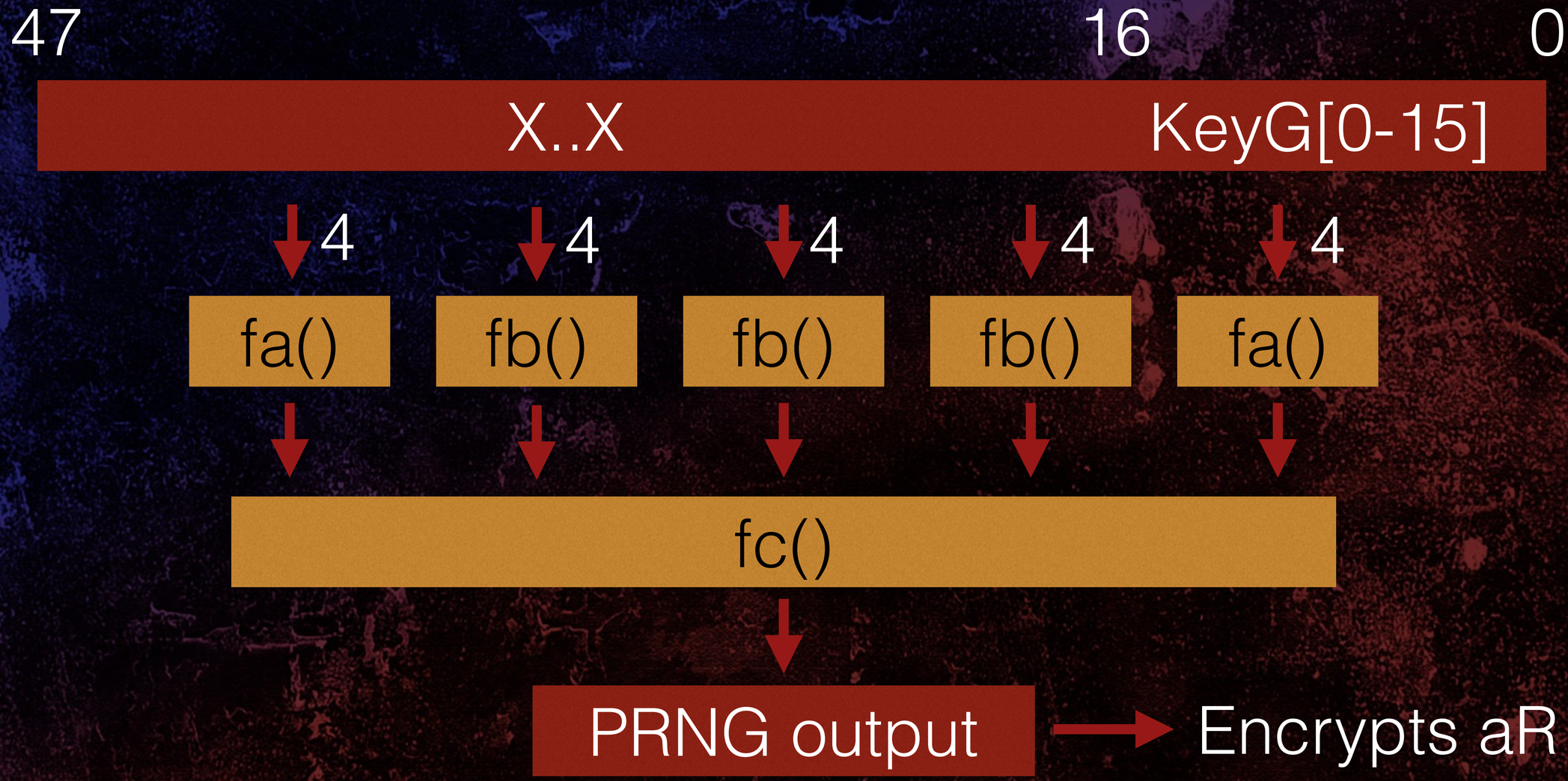
- Any probabilities of 0.0 for any bit of {aR} for a given {nR} indicates the guess is wrong:
  - Short cut by scoring guess as 0.0 and move on to next guess
- Weight and combine scores for bits of {aR}:
  - score = ∑ (bit_score * (#relevant_bits + 1))
  - Note: sum works better than product (for some reason, determined through experimentation)

Round 0

- Initial state produces first bit of PRNG output

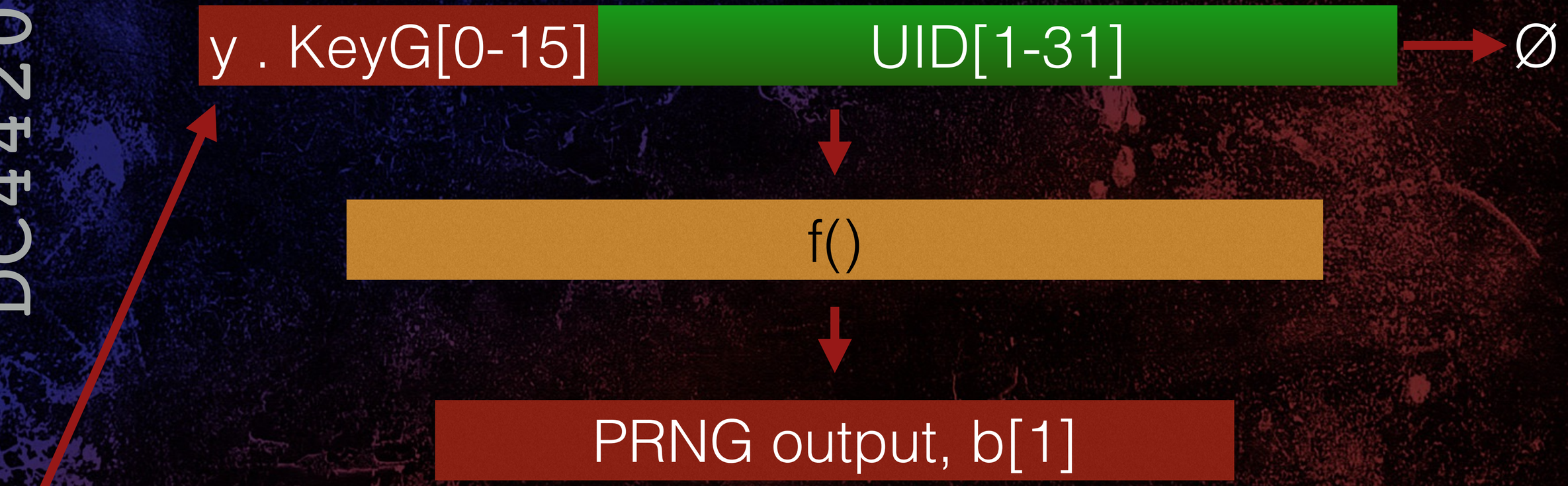


Round 0

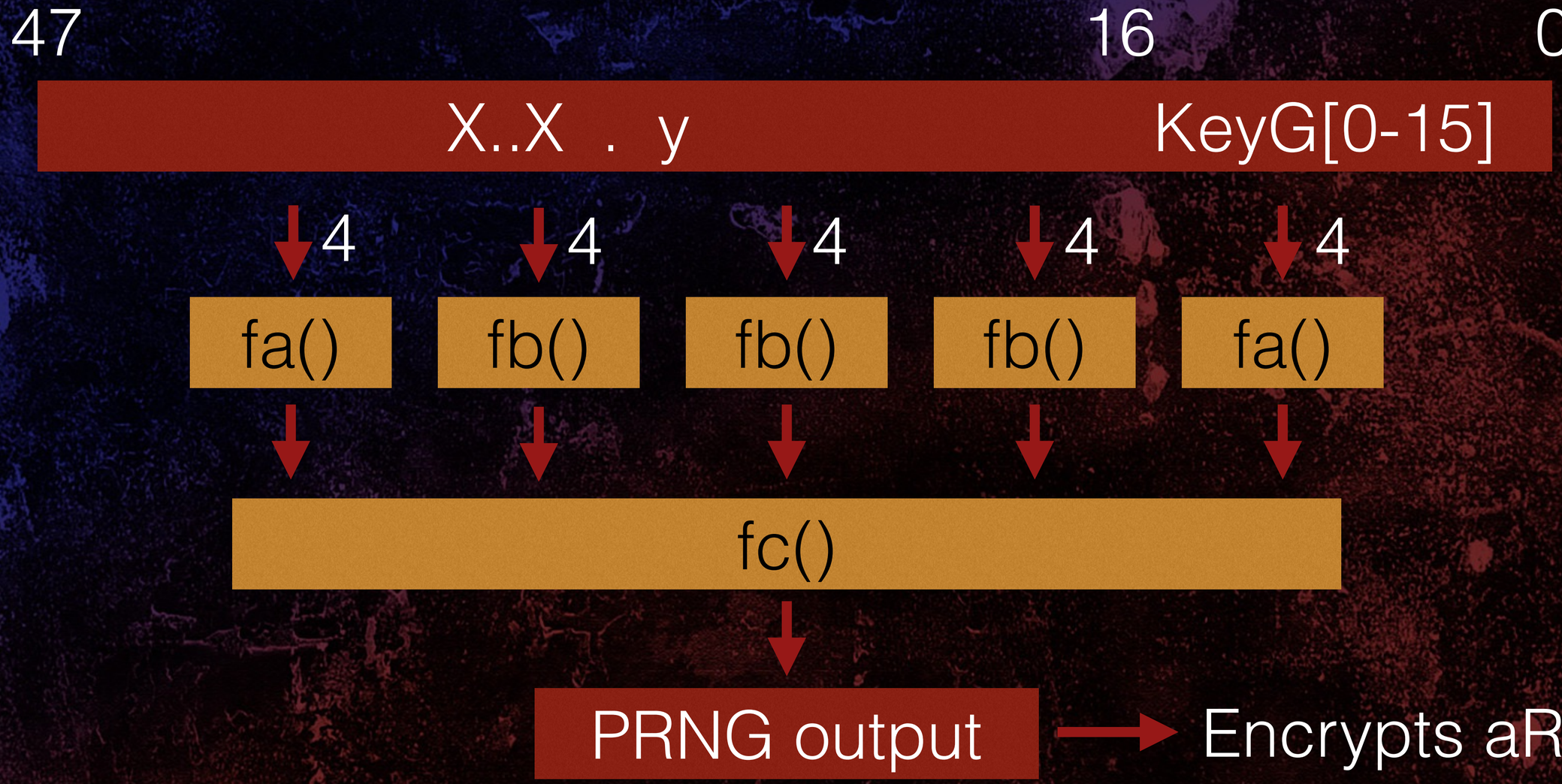


Round 1

- Next state produces next bit of PRNG output

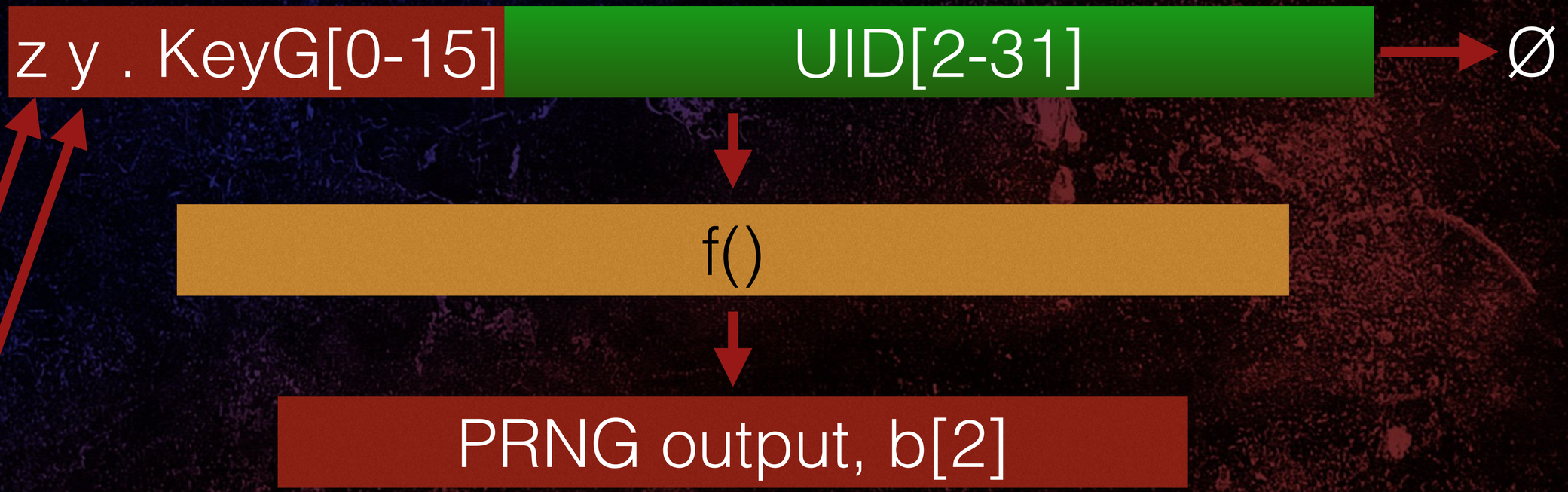


Round 1


47
16
0
X..X . y
KeyG[0-15]
4
4
4
4
4
fa()
fb()
fb()
fb()
fa()
fc()
PRNG output
Encrypts aR

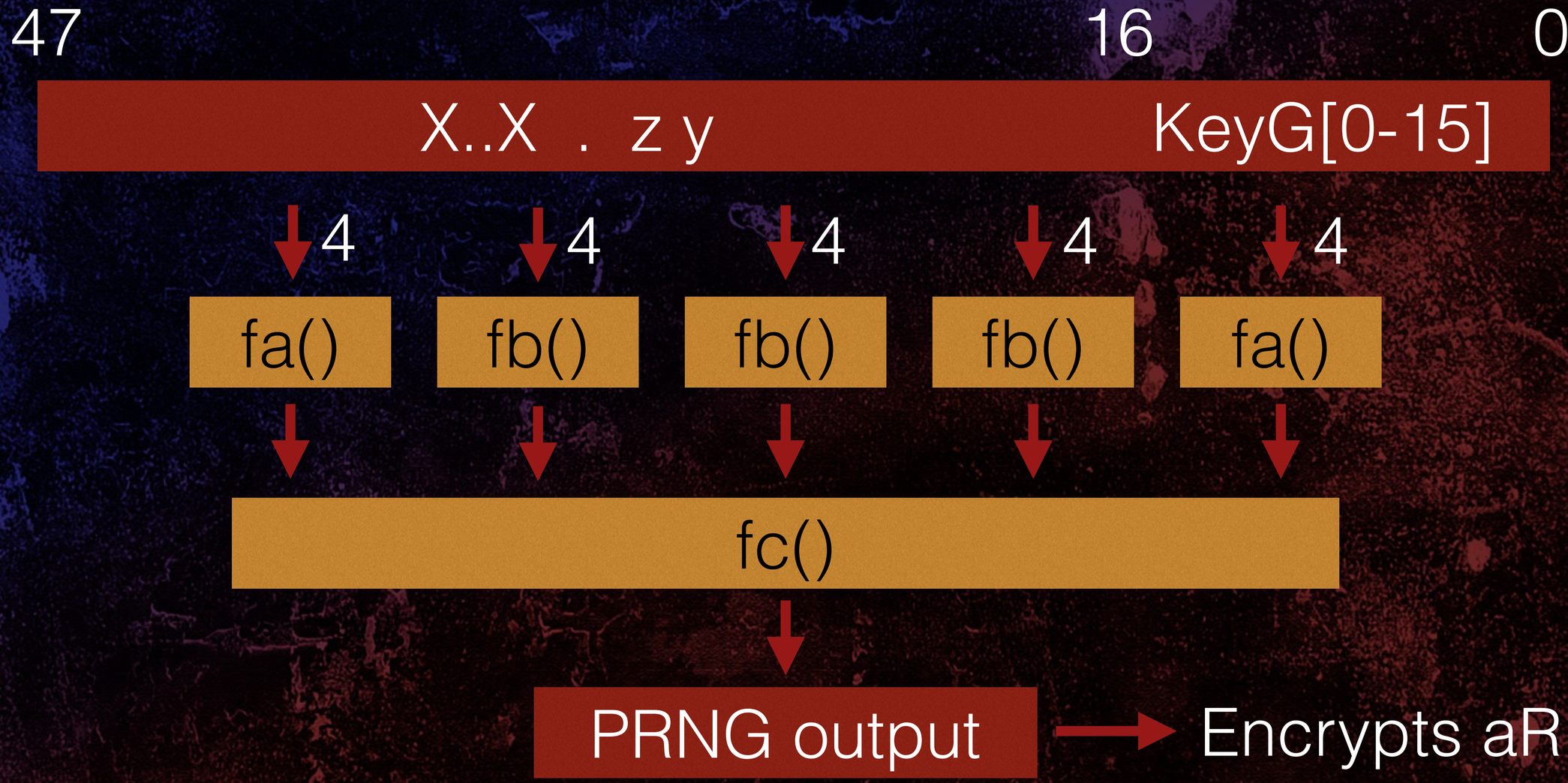
Round 2

- Next state produces next bit of PRNG output

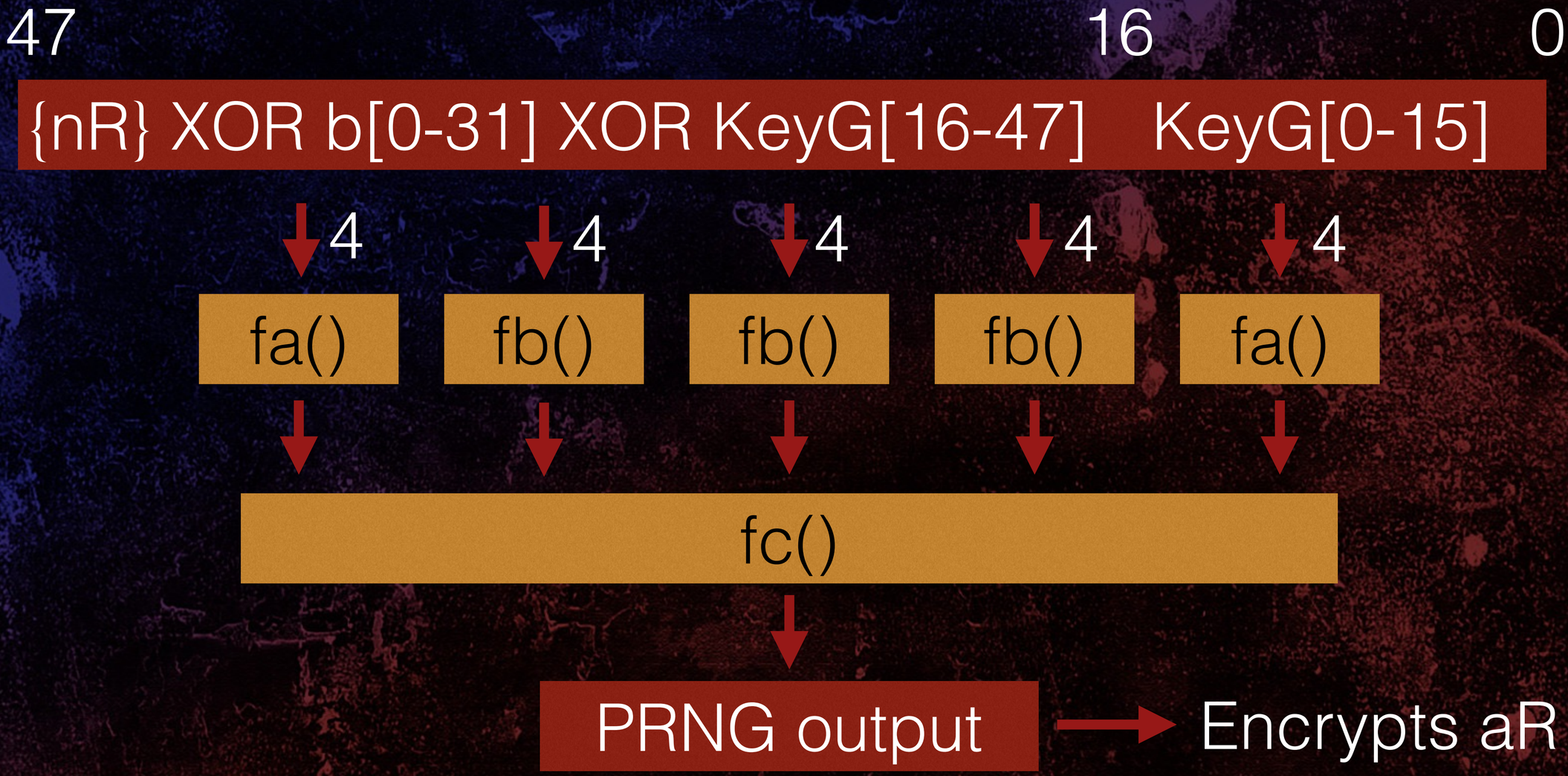


y = {nR}[0] XOR b[0] XOR KeyG[16]
z = {nR}[1] XOR b[1] XOR KeyG[17]

Round 2



Round 32


47
16
0
{nR} XOR b[0-31] XOR KeyG[16-47]
KeyG[0-15]
4
4
4
4
4
fa()
fb()
fb()
fb()
fa()
fc()
PRNG output
Encrypts aR

Fast correlation challenges

- Thinking required for implementing probability calculations
- Lots of testing against known keys to find best scoring and averaging functions
- 16 pairs usually works, 32 gives high success
- Increasing table size helps, but slows it down
- Averages 45s to 3 minutes to find the key

New RFIDler commands

- SNIFF-PWM <C|S|L> - HiTag2 clear/store/list {nR},{aR}
- HITAG2-CRACK <{NR}> <{AR}> - Attack 1
- HITAG2-KEYSTREAM <{NR}> <{AR}> - for Attack 2
- HITAG2-READER <KEY> [S] - read HiTag2 tags
- HITAG2-CLEARSTOREDTAGS
- HITAG2-COUNTSTOREDTAGS
- HITAG2-LISTSTOREDTAGS [START] [END]

Fast correlation key recovery attack

- SET TAG HITAG2
- SNIFF-PWM C - initialises {nR}/{aR} storage
- AUTORUN SNIFF-PWM S - sets autorun function
- SAVE - stores current tag type and autorun command
- Power RFIDler from USB battery
- RFIDler antenna on RWD, present tag (>=16 times)

Fast correlation key recovery attack

DC4420

- SNIFF-PWM L - list stored {nR}/{aR} values
- UID
- Copy/paste {nR}/{aR} values to file
- ./ht2crack4 -u <UID> -n <{nR} {aR} file>
  [-N number of nRaR pairs to use]
  [-T table size, defaults to 800000]

Demo

Fast correlation key recovery attack

DC4420

Closing remarks

Closing remarks

- Read-protecting pages defeats attack 1
  - Cannot read key
- Attacks 2, 3 and 4 recover the key
  - Overcomes read-protection countermeasure
- Can crack HiTag2 in minutes
- Cars require extra work beyond basic HiTag2 cracking

Closing remarks

DC4420

- We can learn a lot from academic experts in cryptography
  - The approaches are inspirational
- We can implement their attacks
- There must be more amazing attacks
- We should do more research

Cracking HiTag2 Crypto

Weaponising Academic Attacks for Breaking and Entering

DC4420

The End!

Kev Sheldrake

rtfcode@gmail.com || @kevsheldrake

github/rtfcode rtfc.org.uk