Formal Representation Language for PUF Constructions and Compositions and Learnability Analysis

Durba Chatterjee, Debdeep Mukhopadhyay, and Aritra Hazra

Indian Institute of Technology Kharagpur, India
durba@iitkgp.ac.in, debdeep@iitkgp.ac.in, aritrah@cse.iitkgp.ac.in

We present a syntactical representation (grammar) to formally describe any PUF construction. We then use this grammar to represent several PUF designs.

1 Grammar for Formal PUF Representation

The grammar to represent any PUF design or composition of PUFs is given as follows:

```
TOP::
    MODULE TOP
  | MODULE
MODULE::
    begin PRIMITIVE ( INPUT_DEF )
        STATEMENTS
        OUTPUT_DEF
    end PRIMITIVE

PRIMITIVE::
    PUF_PRIMITIVE
  | BASIC_PRIMITIVE

PUF_PRIMITIVE::
    APUF
  | XORAPUF

BASIC_PRIMITIVE::
    D_FLIPFLOP
  | ARBITER
  | MUX_2x1
  | SWITCH_2x2
  | DELAY-CHAIN
  | NAND_LATCH

INPUT_DEF::
    DATA_TYPE TUPLE DELIMITER INPUT_DEF
  | DATA_TYPE TUPLE
  | //no-input

OUTPUT_DEF::
    return ( INPUT_DEF ) DELIMITER | //no-output
```

```
PRIMITIVE_CALL::
    PRIMITIVE ( VARIABLES )

TUPLE::
    < VARIABLES >
  | STRING
  | NUMBER

VARIABLES::
    TUPLE DELIMITER VARIABLES
  | TUPLE
  | //null

STRING::
    [a-zA-Z_][a-zA-Z0-9_]*

NUMBER::
    [1-9][0-9]*

DELIMITER::
    ; | ,  //semi-colon or comma

STATEMENTS::
    STATEMENT STATEMENTS
  | STATEMENT

STATEMENT::
    ASSIGNMENT
  | IFELSE_STATEMENT
  | SERIAL_STATEMENT
  | PARALLEL_STATEMENT

ASSIGNMENT::
    STRING = EXPRESSION DELIMITER
  | TUPLE = PRIMITIVE_CALL DELIMITER
  | DELIMITER    //null-statement

EXPRESSION::
    EXPRESSION ARITHMETIC_OPERATOR EXPRESSION
  | EXPRESSION LOGICAL_OPERATOR EXPRESSION
  | ( EXPRESSION )
  | not EXPRESSION
  | STRING
  | NUMBER

ARITHMETIC_OPERATOR::
    / | * | + | - | %

LOGICAL_OPERATOR::
    and | or | xor | == | != | <= | >= | < | >

IFELSE_STATEMENT::
    if EXPRESSION then
```

```
        STATEMENTS
    end if
  | if EXPRESSION then
        STATEMENTS
    end if
    else
        STATEMENTS
    end if
  | if EXPRESSION then
        STATEMENTS
    end if
    ELSEIF_STATEMENTS
    else
        STATEMENTS
    end if

ELSEIF_STATEMENTS::
    else if EXPRESSION then
        STATEMENTS
    end if
    ELSEIF_STATEMENTS
  | else if EXPRESSION then
        STATEMENTS
    end if

SERIAL_STATEMENT::
    serial ASSIGNMENT to EXPRESSION do
        STATEMENTS
    end serial

PARALLEL_STATEMENT::
    parallel ASSIGNMENT to EXPRESSION do
        STATEMENTS
    end parallel
```

2 Structural Design of PUF Constructions

In this section, we present the structural design of PUFs using the grammar given in Section 1. We begin by enlisting the primitive components required in each construction and then represent it using the formal representation language generated by the grammar.

2.1 APUF [1]

Primitive components:

1. D Flip-flop (`D_FLIPFLOP`) and Arbiter (`ARBITER`)
2. 2×1-Multiplexer (`MUX_2x1`) and 2×2-Switch (`SWITCH_2x2`)
3. Delay chain of Arbiter PUF (`DELAY-CHAIN`)

Representation:

Algorithm 1: Structural Representation of APUF

```
Input parameters:
    – Number of stages/switches (n)
    – Challenge bits ($c = \langle c_1, \ldots, c_n \rangle$)
    – Enable bit (en)
Output parameters:
    – Response bit (apuf_out)
Internal variables:
    – Top signal lines at the input of each stage (t)
    – Bottom signal lines at the input of each stage (b)

Structural Design:
begin APUF ( num n, vec c,
                 bit en )
$\langle t, b \rangle$ = DELAY-CHAIN (n, en, c);
apuf_out = ARBITER (t, b);
return ( bit apuf_out );
end APUF

begin DELAY-CHAIN ( num n,
             vec c, bit en )
t = en; b = en;
serial $i = 1$ to $n - 1$ do
  | $\langle t, b \rangle$ = SWITCH_2x2 (t, b, $c_i$);
end serial
return ( vec $\langle t, b \rangle$ );
end DELAY-CHAIN

begin SWITCH_2x2 ( bit t_in,
           bit b_in, bit c_in )
top_out = MUX_2x1 (t_in, b_in, c_in);
bot_out = MUX_2x1 (b_in, t_in, c_in);
return ( vec $\langle$top_out, bot_out$\rangle$ );
end SWITCH_2x2

begin ARBITER (bit in, bit clk)
out = D_FLIPFLOP (in, clk);
return ( bit out );
end ARBITER
```

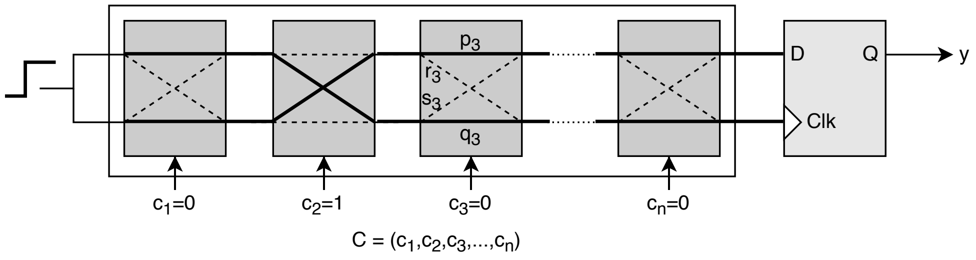


Figure 1: Arbiter PUF

2.2 XOR-APUF [2]

Primitive components:

1. Arbiter PUF (APUF)
2. XOR gate ($\oplus$)

Representation:

Algorithm 2: Structural Representation of XORPUF

```
*Input parameters:*
    – Number of stages/switches (n)
    – Number of delay chains (k)
    – Challenge bits ($c = \langle c_1, \dots, c_n \rangle$)
    – Enable bit (en)
*Output parameters:*
    – Response bit (xorpuf_out)
*Internal variables:*
    – Response from each APUF (a)

*Structural Design:*
begin XORPUF ( num n, num k, vec c, bit en )
parallel $i = 1$ to $k$ do
  |  $a_i$ = APUF (n, c, en);
end parallel
xorpuf_out = $a_1 \oplus \cdots \oplus a_k$;
return ( bit xorpuf_out );
end XORPUF
```

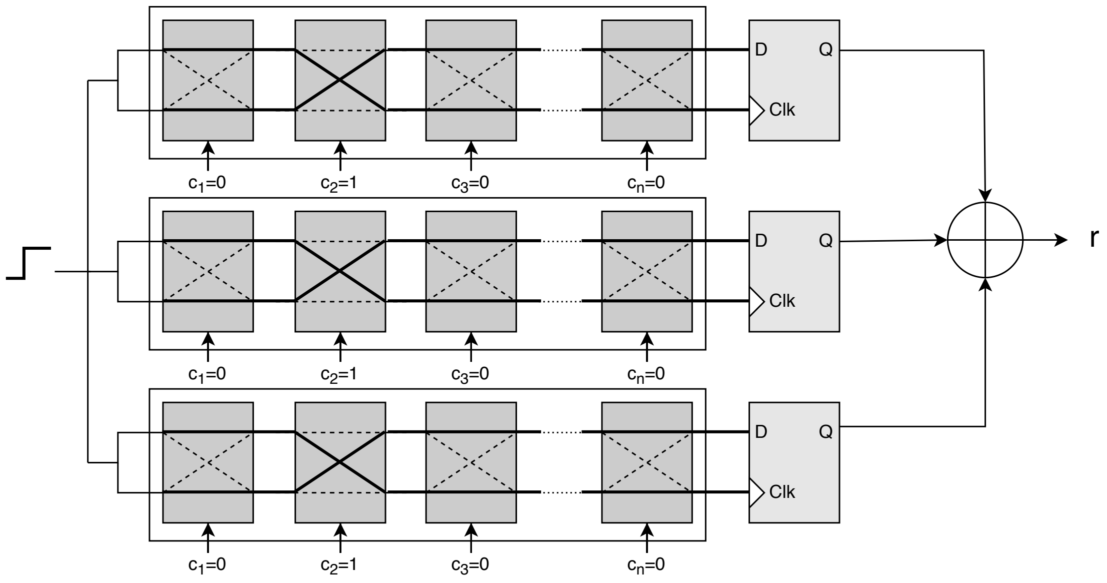


Figure 2: 3-XOR APUF

2.3 FF-APUF [3]

Primitive components:

1. Arbiter with D flip-flop (`ARBITER`)
2. 2×2-switch (`SWITCH_2x2`)

Representation:

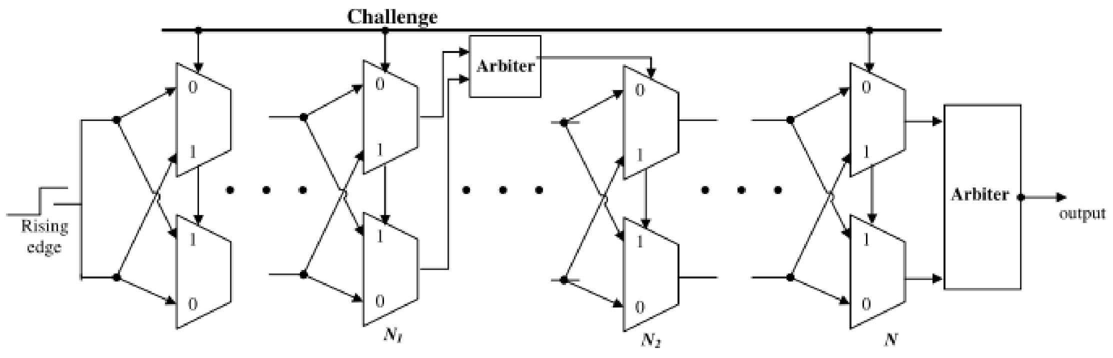


Figure 3: Feed-forward arbiter PUFs [3]

Algorithm 3: Structural Representation of FF-APUF with Single FF Input and Single FF Output

```
Input parameters:
    – Number of stages/switches (n)
    – Challenge bits ($c = \langle c_1, \ldots, c_n \rangle$)
    – Enable bit (en)
    – Feed-forward input stage to arbiter (ff_in)
    – Feed forward output stage from arbiter (ff_out)
Output parameters:
    – Response bit (ffapuf_out)
Internal variables:
    – Top signal lines at the input of each stage (t)
    – Bottom signal lines at the input of each stage (b)
    – Response from feed-forward arbiter (arb_int)

Structural Design:
begin FFAPUF ( num n, vec c, bit en,
                       num ff_in, num ff_out )
$\langle t, b \rangle$ = FF-DELAY-CHAIN (n, c, en, ff_in, ff_out);
ffapuf_out = ARBITER (t, b);
return ( bit ffapuf_out );
end FFAPUF

begin FF-DELAY-CHAIN ( num n, vec c, bit en,
                       num ff_in, num ff_out )
t = en; b = en;
serial $i = 1$ to $n - 1$ do
    $\langle t, b \rangle$ = SWITCH_2x2 (t, b, $c_i$);
    if $i$ == ff_in then
        arb_int = ARBITER (t, b);
    end if
    if $i + 1$ == ff_out then
        $c_{i+1}$ = arb_int;
    end if
end serial
$\langle t, b \rangle$ = SWITCH_2x2 (t, b, $c_n$);
return ( vec $\langle t, b \rangle$ );
end FF-DELAY-CHAIN
```

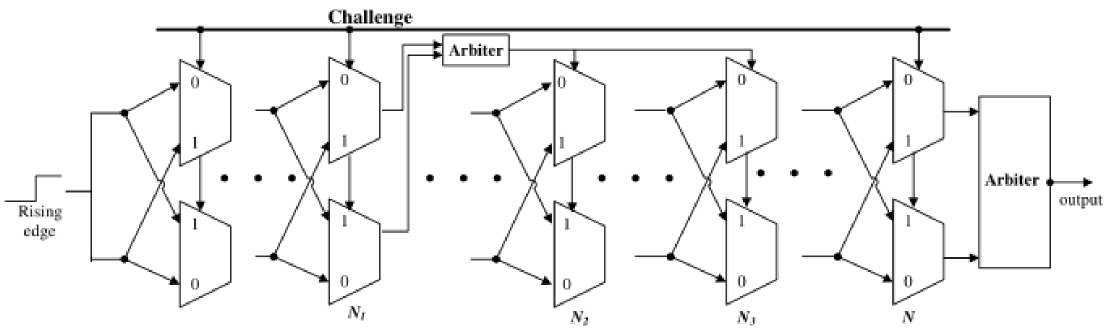


Figure 4: Feed-forward arbiter PUFs with two feed-forward outputs [3]

Other Variants of Feed Forward APUF

1. FF-APUF with single feed-forward input and multiple feed-forward outputs

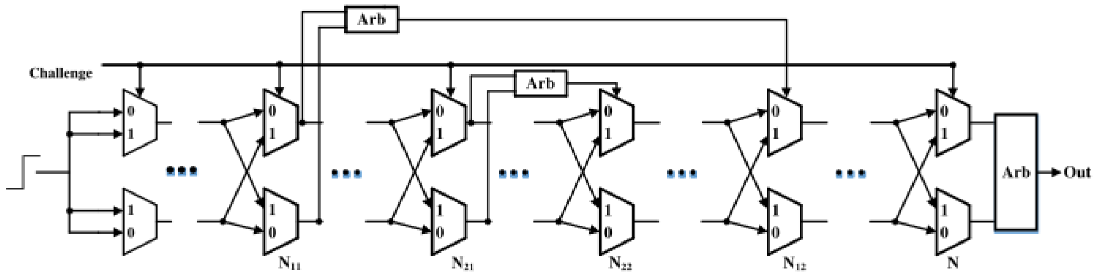
Algorithm 4: Structural Representation of FF-APUF with Single FF Input and Multiple FF Output

```
Input parameters:
    – Number of stages/switches (n)
    – Challenge bits ($c = \langle c_1, \dots, c_n \rangle$)
    – Enable bit (en)
    – Feed-forward input stage to arbiter (ff_in)
    – Feed forward output stages from arbiter (ff_out = $\langle q_1, \dots, q_m \rangle$)
Output parameters:
    – Response bit (ffapuf_out)
Internal variables:
    – Top signal lines at the input of each stage (t)
    – Bottom signal lines at the input of each stage (b)
    – Response from feed-forward arbiter (arb_int)

Structural Design:
begin FFAPUF_SIMO ( num n, vec c, bit en,
                    num ff_in, vec ff_out )
t = en; b = en;
serial $i = 1$ to $n - 1$ do
    $\langle t, b \rangle$ = SWITCH_2x2 ($t, b, c_i$);
    if $i$ == ff_in then
        arb_int = ARBITER ($t, b$);
    end if
    if $i + 1 == q_1$ or $\cdots$ or $i + 1 == q_m$ then
        $c_{i+1}$ = arb_int;
    end if
end serial
$\langle t, b \rangle$ = SWITCH_2x2 ($t, b, c_n$);
ffapuf_out = ARBITER ($t, b$);
return ( bit ffapuf_out );
end FFAPUF_SIMO
```

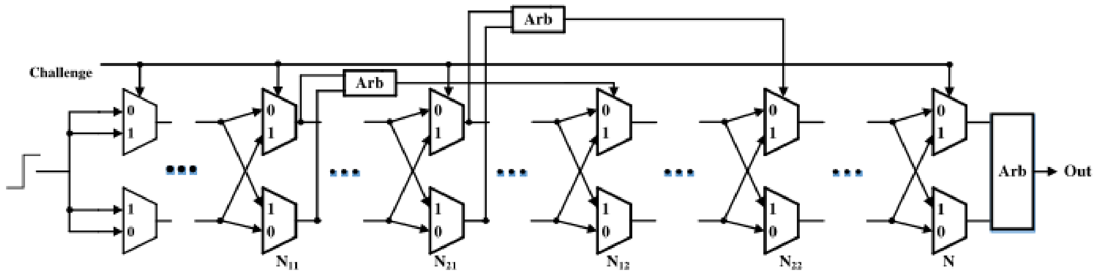
2. FF-APUF with multiple feed-forward inputs and multiple feed-forward outputs
   Depending on the relative position of the input and output stages of the feed forward loops, the FF-APUFs architectures can be categorized as – (a) Nested, (b) Overlap, (c) Cascade, and (d) Separate. Figure 5 depicts all the four configurations assuming only two feed forward loops.

Algorithm 5: Structural Representation of FF-APUF with Multiple FF Input and Multiple FF Output

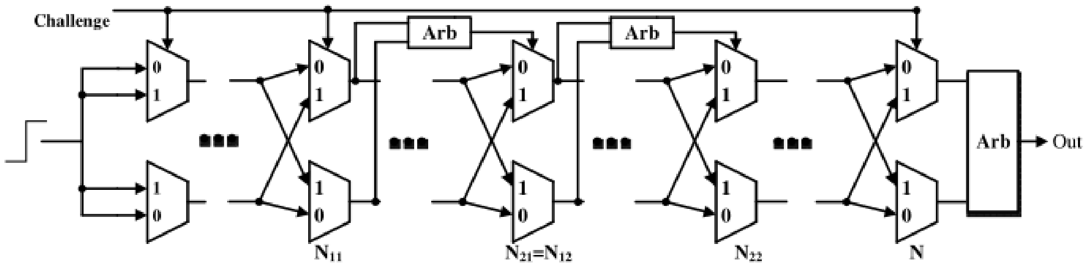
```
Input parameters:
    – Number of stages/switches (n)
    – Challenge bits ($c = \langle c_1, \ldots, c_n \rangle$)
    – Enable bit (en)
    – Feed-forward input stages to arbiters ($ff\_in = \langle p_1, \ldots, p_m \rangle$)
    – Feed forward output stages from arbiters ($ff\_out = \langle q_1, \ldots, q_m \rangle$)
Output parameters:
    – Response bit (ffapuf_out)
Internal variables:
    – Top signal lines at the input of each stage (t)
    – Bottom signal lines at the input of each stage (b)
    – Response from feed-forward arbiter (arb_int)

Structural Design:
begin FFAPUF_MIMO ( num n, vec c, bit en,
                    vec ff_in, vec ff_out )
t = en; b = en;
serial $i = 1$ to $n - 1$ do
    $\langle t, b \rangle$ = SWITCH_2x2 ($t, b, c_i$);
    serial $k = 1$ to $m$ do
        if $i == p_k$ then
            $arb\_int_k$ = ARBITER ($t, b$);
        end if
    end serial
    serial $k = 1$ to $m$ do
        if $i + 1 == q_k$ then
            $c_{i+1} = arb\_int_k$;
        end if
    end serial
end serial
$\langle t, b \rangle$ = SWITCH_2x2 ($t, b, c_n$);
ffapuf_out = ARBITER ($t, b$);
return ( bit ffapuf_out );
end FFAPUF_MIMO
```



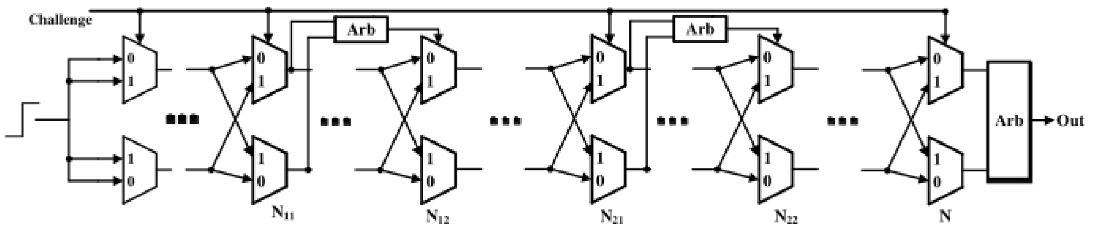
(a) Nested configuration.



(b) Overlap configuration.



(c) Cascade configuration.



(d) Separate configuration.

Figure 5: Different variants of FF-APUF with multiple feed-forward input and multiple feed-forward output

2.4 FF-XOR-PUF [4]

Primitive components:

1. FF-APUF (FFAPUF)
2. XOR gate ($\oplus$)

Representation:

Algorithm 6: Structural Representation of FF-XORPUF

```
Input parameters:
    – Number of stages/switches (n)
    – Number of delay chains (k)
    – Challenge bits ($c = \langle c_1, \dots, c_n \rangle$)
    – Enable bit (en)
    – Feed-forward input stages to arbiters ($ff\_in = \langle p_1, \dots, p_k \rangle$)
    – Feed forward output stages from arbiters ($ff\_out = \langle q_1, \dots, q_k \rangle$)
Output parameters:
    – Response bit (ffxorpuf_out)
Internal variables:
    – Response from each FF-APUF (ffa)

Structural Design:
begin FFXORPUF ( num n, vec c, num k, bit en,
                 num ff_in, num ff_out )
parallel $i = 1$ to $k$ do
  | $ffa_i$ = FFAPUF ($n, c, en, p_i, q_i$);
end parallel
$ffxorpuf\_out = ffa_1 \oplus \cdots \oplus ffa_k$;
return ( bit ffxorpuf_out );
end FFXORPUF
```

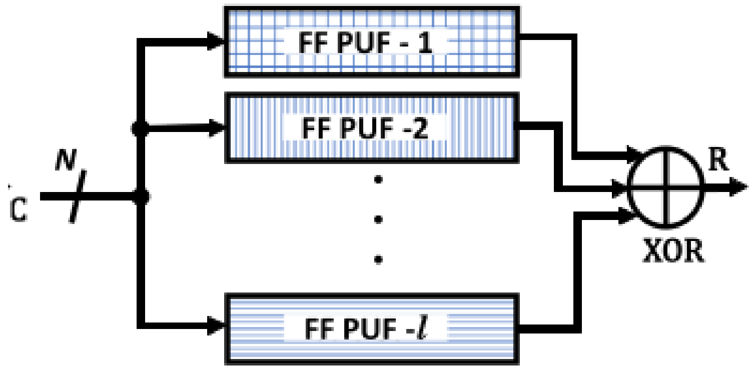


Figure 6: Feed-forward XOR APUFs [4]

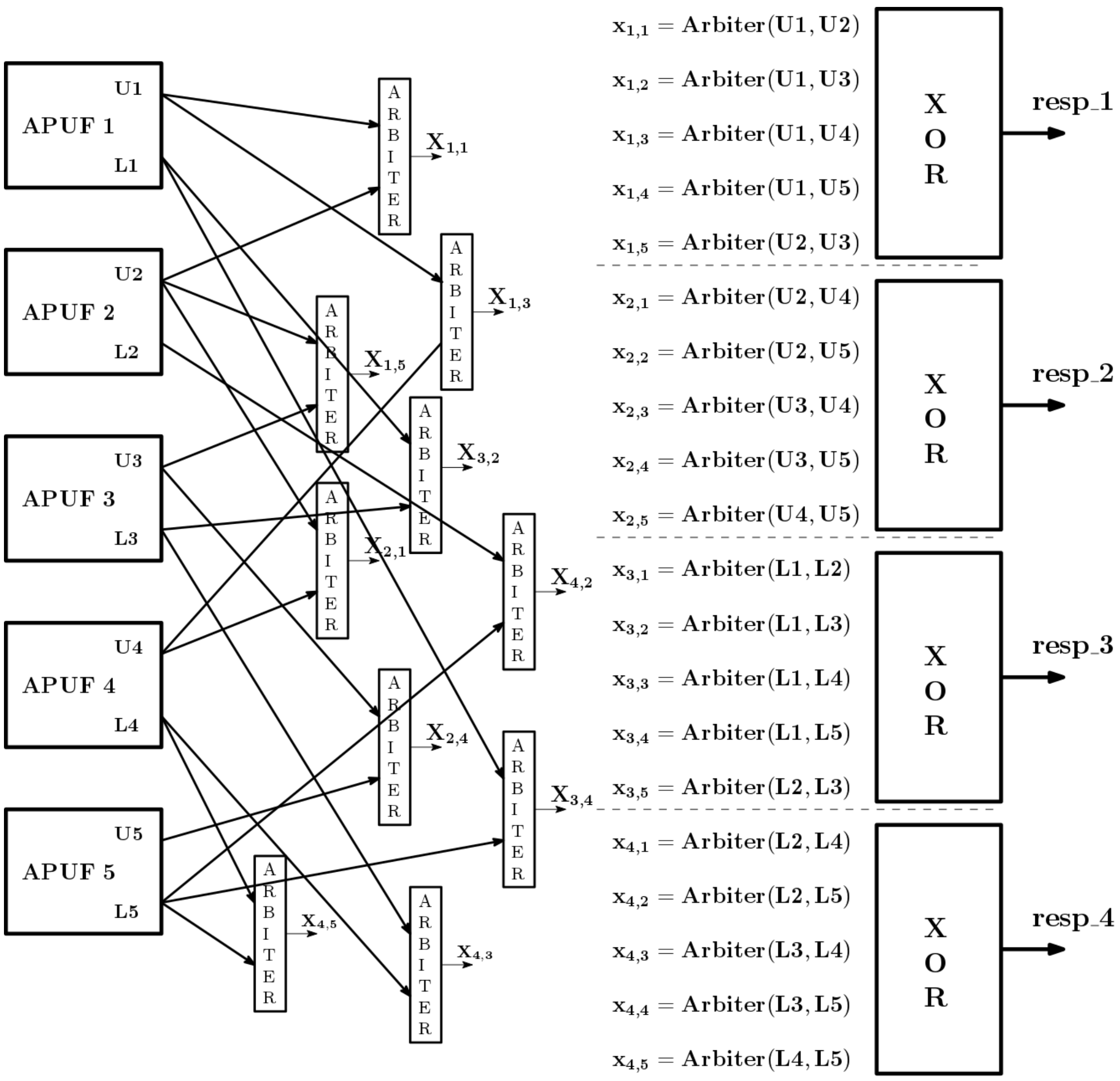

APUF 1
U1
L1
APUF 2
U2
L2
APUF 3
U3
L3
APUF 4
U4
L4
APUF 5
U5
L5
ARBITER
$X_{1,1}$
$X_{1,3}$
$X_{1,5}$
$X_{3,2}$
$X_{2,1}$
$X_{4,2}$
$X_{2,4}$
$X_{3,4}$
$X_{4,5}$
$X_{4,3}$
$x_{1,1}$ = Arbiter(U1, U2)
$x_{1,2}$ = Arbiter(U1, U3)
$x_{1,3}$ = Arbiter(U1, U4)
$x_{1,4}$ = Arbiter(U1, U5)
$x_{1,5}$ = Arbiter(U2, U3)
$x_{2,1}$ = Arbiter(U2, U4)
$x_{2,2}$ = Arbiter(U2, U5)
$x_{2,3}$ = Arbiter(U3, U4)
$x_{2,4}$ = Arbiter(U3, U5)
$x_{2,5}$ = Arbiter(U4, U5)
$x_{3,1}$ = Arbiter(L1, L2)
$x_{3,2}$ = Arbiter(L1, L3)
$x_{3,3}$ = Arbiter(L1, L4)
$x_{3,4}$ = Arbiter(L1, L5)
$x_{3,5}$ = Arbiter(L2, L3)
$x_{4,1}$ = Arbiter(L2, L4)
$x_{4,2}$ = Arbiter(L2, L5)
$x_{4,3}$ = Arbiter(L3, L4)
$x_{4,4}$ = Arbiter(L3, L5)
$x_{4,5}$ = Arbiter(L4, L5)
XOR
resp_1
resp_2
resp_3
resp_4

Figure 7: 5-4 DAPUF

2.5 DAPUF [5]

Primitive components:

1. Delay chain of Arbiter PUF (`DELAY-CHAIN`)
2. Arbiter with D flip-flop (`ARBITER`)
3. XOR gate ($\oplus$)

Figure 7 depicts a specific DAPUF construction consisting of five delay chains, which takes an n-bit challenge and gives a 4-bit response.

Representation:

Algorithm 7: Structural Representation of DAPUF

Input parameters:

- Number of delay chains ($\mathtt{k}$)
- Number of stages/switches in each delay-chain ($\mathtt{n}$)
- Challenge bits ($\mathtt{c} = \langle \mathtt{c_1}, \ldots, \mathtt{c_n} \rangle$)
- Enable bit ($\mathtt{en}$)

Output parameters:

- m Response bits ($\mathtt{dapuf\_out} = \langle \mathtt{r_1}, \ldots, \mathtt{r_m} \rangle$)

Internal variables:

- Top output signal line of each delay-chain ($\mathtt{t} = \langle \mathtt{t_1}, \ldots, \mathtt{t_k} \rangle$)
- Bottom output signal line of each delay-chain ($\mathtt{b} = \langle \mathtt{b_1}, \ldots, \mathtt{b_k} \rangle$)
- Maximum number of XOR inputs from arbiters ($\mathtt{xor\_cnt}$)
- Response from arbiters with top signal lines ($\mathtt{a_g^{top}}$)
- Response from arbiters with bottom signal lines ($\mathtt{a_g^{bot}}$)

Structural Design:

```
begin DAPUF ( num k, num n,
              vec c, bit en )
parallel $i = 1$ to $k$ do
    $\langle t_i, b_i \rangle$ = DELAY-CHAIN $(n, en, c)$;
end parallel
xor_cnt = MATH-CEIL $(k * (k-1)/m)$;
$g = 1$; $h = 1$; $r_h = 0$;
serial $i = 1$ to $k-1$ do
    serial $j = i+1$ to $k$ do
        $a^{top}$ = ARBITER $(t_i, t_j)$;
        $r_h = r_h \oplus a^{top}$;
        $g = g + 1$;
        if $g >$ xor_cnt then
            $h = h + 1$; $r_h = 0$; $g = 1$;
        end if
    end serial
end serial
serial $i = 1$ to $k-1$ do
    serial $j = i+1$ to $k$ do
        $a^{bot}$ = ARBITER $(b_i, b_j)$;
        $r_h = r_h \oplus a^{bot}$;
        $g = g + 1$;
        if $g >$ xor_cnt then
            $h = h + 1$; $r_h = 0$; $g = 1$;
        end if
    end serial
end serial
return ( vec $\langle r_1, \ldots, r_m \rangle$ );
end DAPUF
```

OR

```
begin DAPUF ( num k, num n,
              vec c, bit en )
parallel $i = 1$ to $k$ do
    $\langle t_i, b_i \rangle$ = DELAY-CHAIN $(n, en, c)$;
end parallel
xor_cnt = MATH-CEIL $(k * (k-1)/m)$;
$g = 1$;
serial $i = 1$ to $k-1$ do
    parallel $j = i+1$ to $k$ do
        $a_g^{top}$ = ARBITER $(t_i, t_j)$;
        $g = g + 1$;
    end parallel
end serial
serial $i = 1$ to $k-1$ do
    parallel $j = i+1$ to $k$ do
        $a_g^{bot}$ = ARBITER $(b_i, b_j)$;
        $g = g + 1$;
    end parallel
end serial
$h = 1$; $r_h = 0$;
parallel $g = 1$ to $k * (k-1)/2$ do
    if $g > (h *$ xor_cnt$)$ then
        $h = h + 1$; $r_h = 0$;
    end if
    $r_h = r_h \oplus a_g^{top}$;
end parallel
parallel $g = k * (k-1)/2$ to $k * (k-1)$ do
    if $g > (h *$ xor_cnt$)$ then
        $h = h + 1$; $r_h = 0$;
    end if
    $r_h = r_h \oplus a_g^{bot}$;
end parallel
return ( vec $\langle r_1, \ldots, r_m \rangle$ );
end DAPUF
```

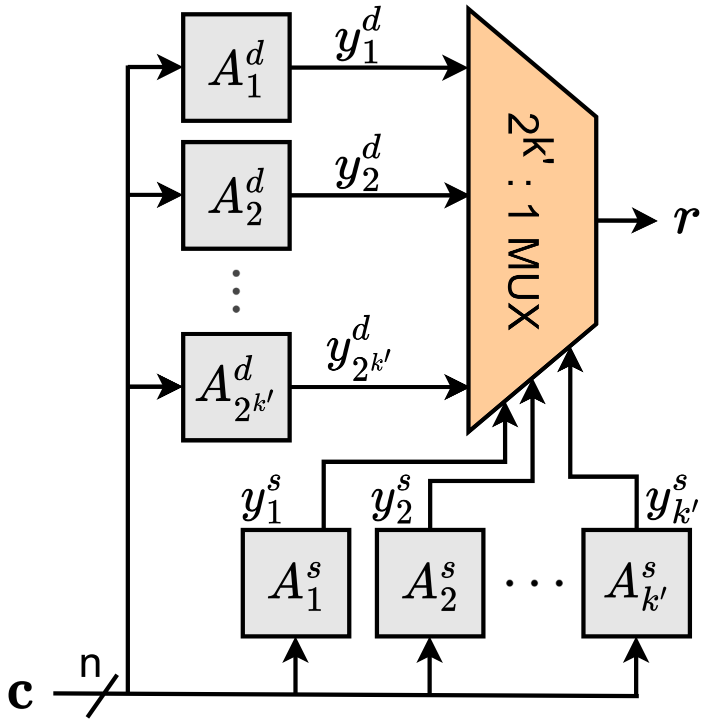


Figure 8: Block diagram of a MUX-PUF

2.6 MUX-PUF [6]

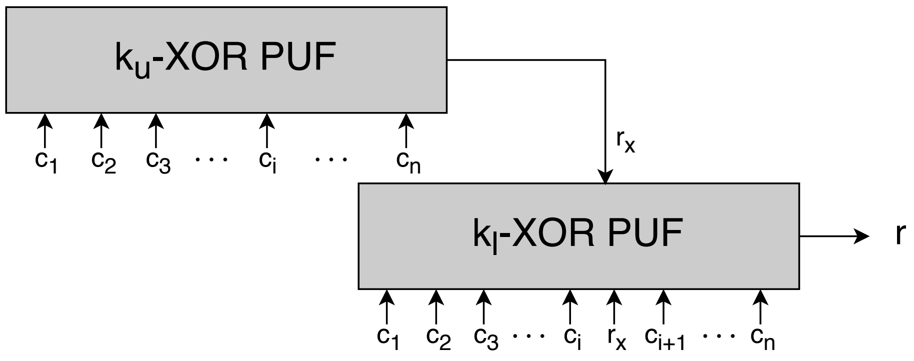
Primitive components:

1. Arbiter PUF (`APUF`)
2. 2×1-Multiplexer (`MUX_2x1`)

Representation:

Algorithm 8: Structural Representation of MUXPUF

```
*Input parameters:*
    – Number of delay chains in selection input (k)
    – Number of stages/switches in each delay-chain (n)
    – Challenge bits ($c = \langle c_1, \ldots, c_n \rangle$)
    – Enable bit (en)
*Output parameters:*
    – $m$ Response bit (muxpuf_out)
*Internal variables:*
    – Response from each APUF connected to selector input ($s_i$)
    – Response from each APUF connected to data input ($d_i$)
    – Input to MUX ($y_{i,j}$)

*Structural Design:*
**begin** MUXPUF ( **num** k, **num** n, **vec** c, **bit** en )
muxpuf_out = 0;
**parallel** $i = 1$ **to** $k$ **do**
    $s_i$ = APUF (n, c, en);
**end parallel**
**parallel** $i = 1$ **to** $2^k$ **do**
    $d_i$ = APUF (n, c, en);
    $y_{1,i} = d_i$;
**end parallel**
**parallel** $i = 1$ **to** $k$ **do**
    **serial** $j = 1$ **to** $2^{k-i}$ **do**
        $y_{i+1,j}$ = MUX_2x1($y_{i,2j-1}, y_{i,2j}, s_i$);
    **end serial**
**end parallel**
muxpuf_out = $y_{k+1,1}$;
**return** ( **bit** muxpuf_out );
**end** MUXPUF
```



k_u-XOR PUF
c_1 c_2 c_3 ⋯ c_i ⋯ c_n
r_x
k_l-XOR PUF
r
c_1 c_2 c_3 ⋯ c_i r_x c_{i+1} ⋯ c_n

Figure 9: (k_u, k_l)- Interpose PUF

2.7 Interpose PUF [7]

Primitive components: XOR-Arbiter PUF (XORPUF)
Representation:

Algorithm 9: Structural Representation of (k_u, k_d)-Interpose PUF

```
Input parameters:
    – Number of stages/switches (n)
    – Number of delay chains in lower XOR PUF($k_l$)
    – Number of delay chains in upper XOR PUF($k_u$)
    – Challenge bits ($c = \langle c_1, \dots, c_n \rangle$)
    – Interpose bit position (t)
    – Enable bit (en)
Output parameters:
    – Response bit (ipuf_out)
Internal variables:
    – Response from upper XORPUF ($y_u$)
    – Input challenge set to lower XORPUF ($x = \langle x_1, \dots, r_{n+1} \rangle$)

Structural Design:
begin IPUF ( num n, num $k_u$, num $k_l$, vec c,
                            num t, bit en)
$y_u$ = XORPUF ($n, k_u, c, en$);
parallel $i = 1$ to $t - 1$ do
  | $x_i = c_i$;
end parallel
$x_t = y_u$;
parallel $i = t + 1$ to $n + 1$ do
  | $x_i = c_{i-1}$;
end parallel
ipuf_out = XORPUF ($n + 1, k_l, x, en$);
return ( bit ipuf_out );
end IPUF
```

2.8 ROPUF [2]

Primitive components:

1. Ring Oscillators with NOT gate (RING_OSC)
2. 2×1-Multiplexer (MUX_2x1)
3. Counter (COUNTER)

Representation:

Algorithm 10: Structural Representation of ROPUF

```
Input parameters:
    – Number of challenge bits (n)
    – Number of inverters in a RO (m)
    – Challenge bits ($c = \langle c_1, \ldots, c_n \rangle$)
    – Enable bit (en)
Output parameters:
    – Response bit (ropuf_out)
Internal variables:
    – Signal line at the input of each RO (t)
    – Signal line at the input of upper MUX (y)
    – Signal line at the input of lower MUX (z)
    – Output of first counter ($count_1$)
    – Output of second counter ($count_2$)

Structural Design:
begin ROPUF ( num n, num m,
              vec c, bit en )
parallel $i = 1$ to $2^n$ do
    $t_i$ = RING_OSC(m, en);
    $y_{1,i} = t_i$;
end parallel
parallel $i = 2^n + 1$ to $2^{n+1}$ do
    $t_i$ = RING_OSC(m, en);
    $z_{1,i-2^n} = t_i$;
end parallel
serial $i = 1$ to n do
    parallel $j = 1$ to $2^n$ do
        $y_{i+1,j}$ = MUX_2x1 ($y_{i,2j-1}, y_{i,2j}, c_i$);
        $z_{i+1,j}$ = MUX_2x1 ($z_{i,2j-1}, z_{i,2j}, c_i$);
    end parallel
end serial
$count_1$ = COUNTER($y_{n+1,1}$);
$count_2$ = COUNTER($z_{n+1,1}$);
if $count_1 > count_2$ then
    ropuf_out = 1;
end if
else
    ropuf_out = 0;
end if
return ( bit ropuf_out );
end ROPUF

begin RING_OSC (num m, bit en)
t = en and t;
serial $i = 1$ to $m - 1$ do
    t = not t;
end serial
return ( vec t );
end RING_OSC

begin COUNTER ( bit sig_in )
count = 0;
if sig_in == 1 then
    count = count + 1;
end if
return ( num count );
end COUNTER
```

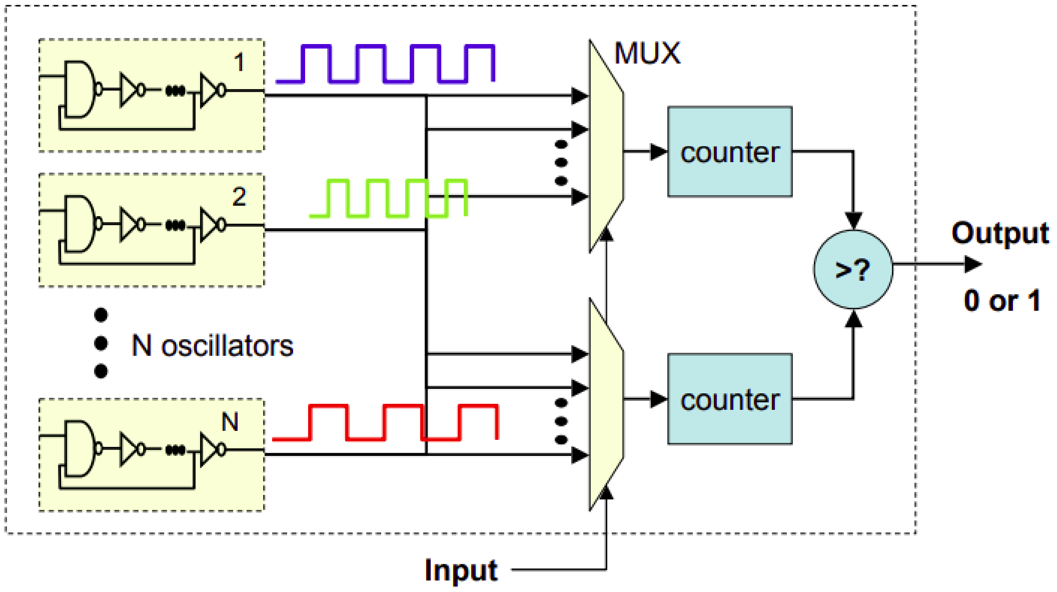


Figure 10: Block diagram of Ring Oscillator PUF [2]

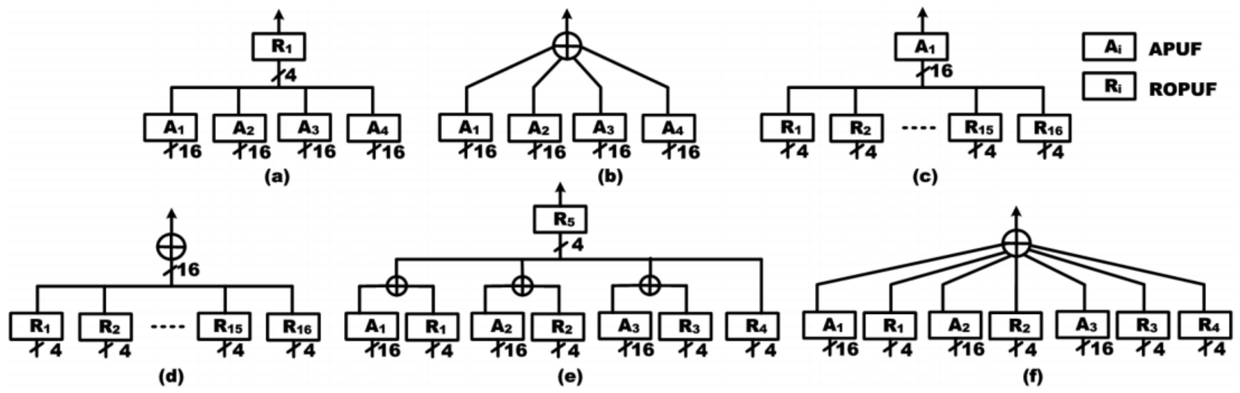


Figure 11: Different configurations of composite PUFs (64-bit challenge and 1-bit response) [8]

2.9 Composite PUF [8]

Here, we consider compositions having at most 2 layers.
Primitive components:

1. Arbiter PUF (`APUF`)
2. Ring Oscillator PUF (`ROPUF`)
3. XOR gate ($\oplus$)
4. Mapping functions

Representation:

Algorithm 11: Structural Representation of Composite PUF – (A) APUFs + ROPUF

```
Input parameters:
    – Challenge bits ($c = \langle c_1, \ldots, c_n \rangle$)
    – Total number of Challenge bits (n)
    – Challenge size of APUFs (na)
    – Number of inverters in RO (m)
    – Enable bit (en)
Output parameters:
    – Response bit (cpuf_out)
Internal variables:
    – Number of PUFs in lower layer (n_arb)
    – Input challenge to lower layer PUF ($x = \langle x_1, \ldots, x_{na} \rangle$)
    – Input challenge to upper layer PUF ($y = \langle y_1, \ldots, y_{n\_arb} \rangle$)

Structural Design:
begin CPUF_AR ( num n, vec c, num na, num m, bit en )
n_arb = n/na;
parallel $i = 1$ to n_arb do
    $x_i = \langle c_{(i-1)*na+1}, \cdots, c_{i*na} \rangle$;
    $y_i$ = APUF (na, $x_i$, en);
end parallel
cpuf_out = ROPUF (n_arb, m, y, en);
return ( bit cpuf_out );
end CPUF_AR
```

Algorithm 12: Structural Representation of Composite PUF – (B) APUFs + XOR

```
Input parameters:
    – Challenge bits ($c = \langle c_1, \ldots, c_n \rangle$)
    – Challenge size of APUFs (na)
    – Enable bit (en)
Output parameters:
    – Response bit (cpuf_out)
Internal variables:
    – Number of PUFs in lower layer (n_arb)
    – Input challenge to lower layer PUF ($x = \langle x_1, \ldots, x_{na} \rangle$)

Structural Design:
begin CPUF_AX ( num n, num na, vec c, bit en )
n_arb = n/na;
parallel $i = 1$ to n_arb do
    $x_i = \langle c_{(i-1)*na+1}, \cdots, c_{i*na} \rangle$;
    $y_i$ = APUF (na, $x_i$, en);
end parallel
cpuf_out = $y_1 \oplus \cdots \oplus y_{n\_arb}$;
return ( bit cpuf_out );
end CPUF_AX
```

Algorithm 13: Structural Representation of Composite PUF – (C) ROPUFs + APUF

```
Input parameters:
    – Challenge bits ($c = \langle c_1, \ldots, c_n \rangle$)
    – Challenge size of ROPUFs (nr)
    – Number of inverters in RO (m)
    – Enable bit (en)
Output parameters:
    – Response bit (cpuf_out)
Internal variables:
    – Number of PUFs in lower layer (n_ro)
    – Input challenge to lower layer PUF ($x = \langle x_1, \ldots, x_{nr} \rangle$)
    – Input challenge to upper layer PUF ($y = \langle y_1, \ldots, y_{n\_ro} \rangle$)

Structural Design:
begin CPUF_RA ( num n, num m, num nr, vec c, bit en )
cpuf_out = 0;
n_ro = n/nr;
parallel $i = 1$ to n_ro do
    $x_i = \langle c_{(i-1)*nr+1}, \cdots, c_{i*nr} \rangle$;
    $y_i$ = ROPUF ($nr, m, x_i, en$);
end parallel
cpuf_out = APUF (n_ro, y, en);
return ( bit cpuf_out );
end CPUF_RA
```

Algorithm 14: Structural Representation of Composite PUF – (D) ROPUFs + XOR

```
Input parameters:
    – Challenge bits ($c = \langle c_1, \ldots, c_n \rangle$)
    – Challenge size of ROPUFs (nr)
    – Number of inverters in RO (m)
    – Enable bit (en)
Output parameters:
    – Response bit (cpuf_out)
Internal variables:
    – Number of PUFs in lower layer (n_ro)
    – Input challenge to lower layer PUF ($x = \langle x_1, \cdots, x_{nr} \rangle$)
    – Input challenge to upper layer PUF ($y = \langle y_1, \cdots, y_{n\_arb} \rangle$)

Structural Design:
begin CPUF_RX ( num n, num m, num nr, vec c, bit en )
cpuf_out = 0;
n_ro = n/nr;
parallel $i = 1$ to n_ro do
    $x_i = \langle c_{(i-1)*nr+1}, \cdots, c_{i*nr} \rangle$;
    $y_i$ = ROPUF($nr, m, x_i, en$);
end parallel
cpuf_out = $y_1 \oplus \cdots \oplus y_{n\_ro}$;
return ( bit cpuf_out );
end CPUF_RX
```

Algorithm 15: Structural Representation of Composite PUF(e)

```
Input parameters:
    – Challenge bits ($\mathtt{c} = \langle \mathtt{c_1}, \ldots, \mathtt{c_n} \rangle$)
    – Number of APUFs in lower layer($\mathtt{a}$)
    – Challenge size of ROPUFs ($\mathtt{m}$)
    – Number of inverters in RO ($\mathtt{inv}$)
    – Enable bit ($\mathtt{en}$)
Output parameters:
    – Response bit ($\mathtt{cpuf\_out}$)
Internal variables:
    – Number of PUFs in lower layer ($\mathtt{n\_arb}$)
    – Input challenge to lower layer PUF ($\mathtt{x} = \langle \mathtt{x_1}, \cdots, \mathtt{x_n r} \rangle$)
    – Input challenge to upper layer PUF ($\mathtt{y} = \langle \mathtt{y_1}, \cdots, \mathtt{y_{n\_arb}} \rangle$)

Structural Design:
begin COMPOSEPUF (num n, vec c, bit en, num a, num m, num inv, bit en)
$\mathtt{n_a} = (\mathtt{n} - \mathtt{m})/\mathtt{a} - \mathtt{m}$;
parallel $i = 1$ to a do
    $\mathtt{loc_a} = (\mathtt{i} - 1) * (\mathtt{n_a} + \mathtt{m})$;
    $\mathtt{c_a} = \langle \mathtt{c_{loc_a+1}}, \cdots, \mathtt{c_{loc_a+n_a}} \rangle$;
    $\mathtt{a_i} = \mathtt{APUF}(\mathtt{n_a}, \mathtt{c_a}, \mathtt{en})$;
    $\mathtt{loc_r} = \mathtt{loc_a} + \mathtt{n_a}$;
    $\mathtt{c_r} = \langle \mathtt{c_{loc_r+1}}, \cdots, \mathtt{c_{loc_r+m}} \rangle$;
    $\mathtt{a_i} = \mathtt{ROPUF}(\mathtt{m}, \mathtt{c_r}, \mathtt{inv}, \mathtt{en})$;
end parallel
parallel $i = 1$ to a do
    $\mathtt{x_i} = \mathtt{a_i} \oplus \mathtt{r_i}$;
end parallel
$\mathtt{c_r} = \langle \mathtt{c_{n-m+1}}, \cdots, \mathtt{c_n} \rangle$;
$\mathtt{x_{a+1}} = \mathtt{ROPUF}(\mathtt{m}, \mathtt{c_{r_{a+1}}}, \mathtt{inv}, \mathtt{en})$;
$\mathtt{cpuf\_out} = \mathtt{ROPUF}(\mathtt{m}, \mathtt{x}, \mathtt{inv}, \mathtt{en})$;
return cpuf_out;
end COMPOSEPUF
```

Algorithm 16: Structural Representation of Composite PUF(f)

```
Input parameters:
    – Challenge bits ($c = \langle c_1, \ldots, c_n \rangle$)
    – Challenge size of APUFs (na)
    – Challenge size of ROPUFs (nr)
    – Number of inverters in RO (m)
    – Order of APUFs and ROPUFs in first layer
            ($o = \langle o_1, \cdots, o_k \rangle; o_i \in \{0, 1\}$)
    – Enable bit (en)
Output parameters:
    – Response bit (cpuf_out)
Internal variables:
    – Number of PUFs in lower layer ($n_1$)
    – Input challenge to lower layer APUF ($x = \langle x_1, \cdots, x_{na} \rangle$)
    – Input challenge to lower layer ROPUF ($x = \langle x_1, \cdots, x_{nr} \rangle$)
    – Input challenge to upper layer PUF ($y = \langle y_1, \cdots, y_{n_1} \rangle$)

Structural Design:
begin COMPOSEPUF_X (num n, num m, num na, num nr, vec c, vec o, bit en)
cpuf_out = 0;
parallel $i = 1$ to k do
    if $o_i == 0$ then
        $x_i = \langle c_{(i-1)*nr+1}, \cdots, c_{i*nr} \rangle$;
    end if
    else
        $x_i = \langle c_{(i-1)*nr+1}, \cdots, c_{i*nr} \rangle$;
    end if
    y = ROPUF(nr, m, $x_i$, en);
    cpuf_out = cpuf_out $\oplus$ y;
end parallel
return cpuf_out;
end COMPOSEPUF_X
```

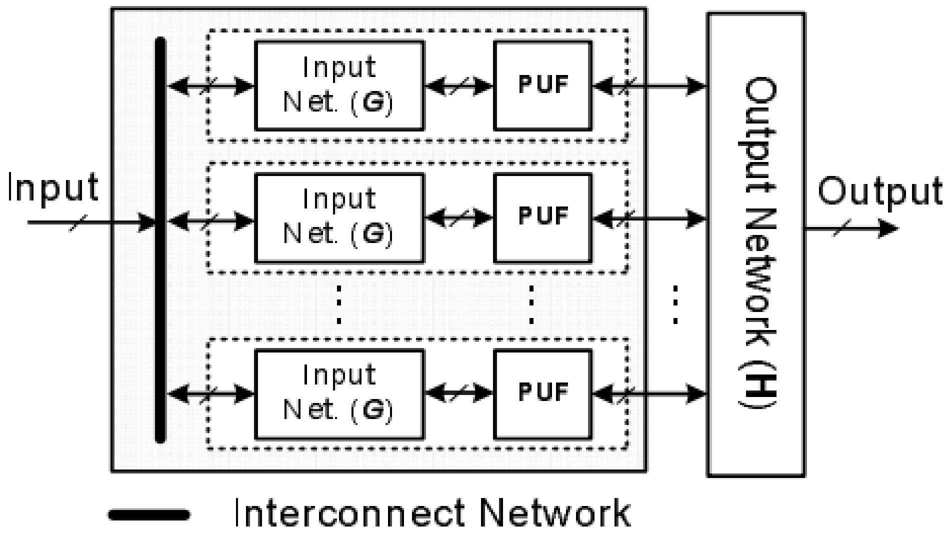


Figure 12: Block diagram of Lightweight Secure PUF [9]

2.10 LS-PUF [9]

Primitive components:

1. Arbiter PUF (`APUF`)
2. XOR gate ($\oplus$) (used in input-output networks as shown in Fig 12)
3. Shift Register (`SHIFTREG`) (used in interconnect network to realize a one-to-one permutation of challenge bits as shown in Fig 12)

Representation:

Algorithm 17: Structural Representation of LS-PUF

```
/*combines the input of the input network of all rows into a single input*/
Input parameters:
    – Number of PUF rows (Q)
    – Number of stages/switches in each PUF (n)
    – Challenge bits ($c = \langle c_1, \ldots, c_n \rangle$)
    – Enable bit (en)
Output parameters:
    – $m$ Response bits ($lspuf\_out = \langle r_1, \ldots, r_m \rangle$)
Internal variables:
    – Output of the interconnect network
          ($x = \langle x_1, \ldots, x_Q \rangle = \langle \langle x_{1,1}, \ldots, x_{1,n} \rangle, \ldots, \langle x_{Q,1}, \ldots, x_{Q,n} \rangle \rangle$)
    – Output of the input network
          ($d = \langle d_1, \ldots, d_Q \rangle = \langle \langle d_{1,1}, \ldots, d_{1,n} \rangle, \ldots, \langle d_{Q,1}, \ldots, d_{Q,n} \rangle \rangle$)
    – Output of each PUF ($y = \langle y_1, \ldots, y_m \rangle$)

Structural Design:

begin LSPUF ( num Q, num n,
              vec c, bit en )
x = INTERCON_NETWORK (Q, c, n);
d = INPUT_NETWORK (Q, n, x);
parallel $i = 1$ to Q do
    $r_i$ = APUF (n, $d_i$, en);
end parallel
lspuf_out = OUTPUT_NETWORK(Q, r);
return ( bit lspuf_out );
end LSPUF;

begin INTERCON_NETWORK ( num Q,
              vec c, num n)
$x_i$ = c;
parallel $i = 1$ to $Q - 1$ do
    $x_{i+1}$ = SHIFTREG (n, $x_i$, $i - 1$);
end parallel
return ( vec x );
end INTERCON_NETWORK

begin SHIFTREG ( num n, vec c,
                 num k )
s = $\langle s_1, \ldots, s_n \rangle$;
parallel $i = 1$ to n do
    $s_{(i+k)\%n} = c_i$;
end parallel
return ( vec s );
end SHIFTREG

begin INPUT_NETWORK ( num Q,
                      num n, vec x )
parallel $i = 1$ to Q do
    parallel $j =$ to $n - 1$ do
        if $j == 1$ then
            $d_{i,(n+2)/2} = x_{i,j}$;
        end if
        else if $j\%2 \neq 0$ then
            $d_{i,(j+1)/2} = x_{i,j} \oplus x_{i,j+1}$;
        end if
        else
            $d_{i,(n+j+2)/2} = x_{i,j} \oplus x_{i,j+1}$;
        end if
    end parallel
end parallel
return ( vec d );
end INPUT_NETWORK

begin OUTPUT_NETWORK ( num Q,
                       vec r)
/*$z$ and $s$ are chosen depending on the security and
 resource trade-off*/
parallel $j = 1$ to m do
    $y_j = 0$;
    parallel $i = 1$ to $z$ do
        $y_j = y_j \oplus r_{(j+s+i)\%Q}$;
    end parallel
end parallel
return ( vec y );
end OUTPUT_NETWORK
```

2.11 CRC-PUF [10]

Primitive components:

1. Arbiter PUF (APUF)
2. LFSR (FIBO_LFSR) with XOR ($\oplus$) and AND (**and**) operations
3. Shift Register (SHIFTREG)

Representation:

Algorithm 18: Structural Representation of CRC-PUF

```
Input parameters:
    – Number of stages/es in each delay-chain ($\mathtt{n}$)
    – Challenge bits ($\mathtt{c} = \langle \mathtt{c_1}, \dots, \mathtt{c_n} \rangle$)
    – Enable bit ($\mathtt{en}$)
Output parameters:
    – $m$ Response bits ($\mathtt{crcpuf\_out} = \langle \mathtt{r_1}, \dots, \mathtt{r_m} \rangle$)
Internal variables:
    – Previous challenge bits ($\mathtt{t} = \langle \mathtt{t_1}, \dots, \mathtt{t_n} \rangle$)
    – Generator polynomial coefficients ($\mathtt{g} = \langle \mathtt{g_1}, \dots, \mathtt{g_n} \rangle$)

Structural Design:
begin CRCPUF ( num n, vec c, bit en )
x = c;
parallel $i = 1$ to $m$ do
    x = FIBO_LFSR (n, x);
    $\mathtt{r_i}$ = APUF (n, x, en);
end parallel
return ( vec $\langle r_1, \dots, r_m \rangle$ );
end CRCPUF

begin FIBO_LFSR ( num n, vec c)
/*$g(x)$ = generator polynomial and
g = $\langle \mathtt{g_1}, \dots, \mathtt{g_n} \rangle$ = coefficient vector*/
xor_poly = 0;
parallel $i = 1$ to $n$ do
    xor_poly = xor_poly $\oplus$ ($\mathtt{g_i}$ and $\mathtt{c_i}$);
end parallel
c = SHIFTREG(n, c, 1);
$\mathtt{c_n}$ = xor_poly;
return ( vec c );
end FIBO_LFSR
```

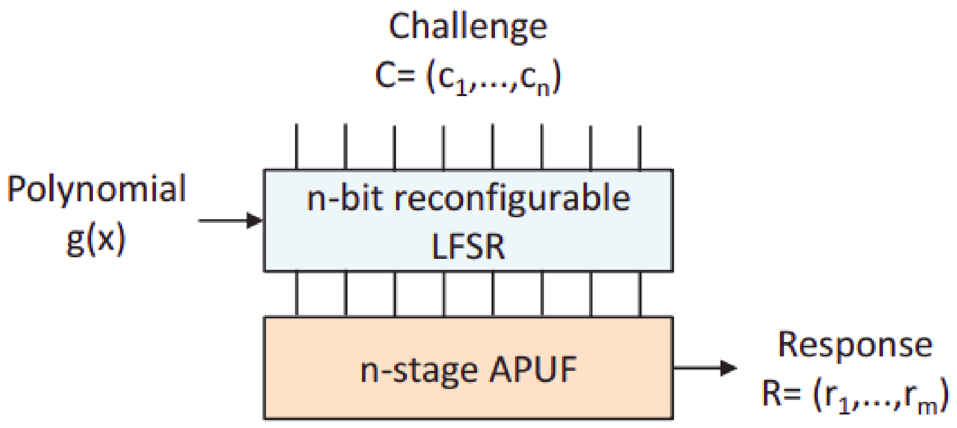


Figure 13: Block diagram of CRC-PUF [10]

2.12 Configurable ROPUF [11]

Primitive components:

1. Ring Oscillators with NOT gate (`RING_OSC`)
2. 2×1-Multiplexer (`MUX_2x1`)
3. Counter (`COUNTER`)

Representation:

Algorithm 19: Structural Representation of Configurable ROPUF

Input parameters:
- Number of Configurable Ring Oscillators (CRO) (`n`)
- Number of inverters in a CRO (`m`)
- Challenge bits ($\mathtt{c} = \langle \mathtt{c_1}, \ldots, \mathtt{c_n} \rangle$)
- Enable bit (`en`)

Output parameters:
- Response bit (`cropuf_out`)

Internal variables:
- Signal line at the input of each CRO (`t`)
- Signal line at the control input of each CRO (`s`)
- Signal line at the input of first MUX (`y`)
- Signal line at the input of second MUX (`z`)
- Input signal to counter (`sig_in`)
- Output of first counter ($\mathtt{count_1}$)
- Output of second counter ($\mathtt{count_2}$)

Structural Design:

```
begin CROPUF ( num n, num m,
               vec c, bit en )
s = $\langle c_1, \cdots, c_m \rangle$;
parallel $i = 1$ to $2^n$ do
    $t_i$ = CONF_RING_OSC $(m, s, en)$;
    $y_{1,i} = t_i$;
end parallel
parallel $i = 2^n + 1$ to $2^{n+1}$ do
    $t_i$ = CONF_RING_OSC $(m, s, en)$;
    $z_{1,i-2^n} = t_i$;
end parallel
serial $i = 1$ to n do
    parallel $j = 1$ to $2^n$ do
        $y_{i+1,j}$ = MUX_2x1 $(y_{i,2j-1}, y_{i,2j}, c_i)$;
        $z_{i+1,j}$ = MUX_2x1 $(z_{i,2j-1}, z_{i,2j}, c_i)$;
    end parallel
end serial
$count_1$ = COUNTER$(y_{n+1,1})$;
$count_2$ = COUNTER$(z_{n+1,1})$;
if $count_1 > count_2$ then
    cropuf_out = 1;
end if
else
    cropuf_out = 0;
end if
return ( bit cropuf_out );
end CROPUF

begin CONF_RING_OSC ( num m,
                      vec s, bit en )
$t_1$ = en and $t_1$;
parallel $i = 1$ to $m - 1$ do
    $nt_1$ = not $t_i$;
    $nt_2$ = not $t_i$;
    $t_{i+1}$ = MUX_2x1 $(nt_1, nt_2, s_i)$;
end parallel
$t_1 = t_m$;
return ( bit $t_m$ );
end CONF_RING_OSC
```

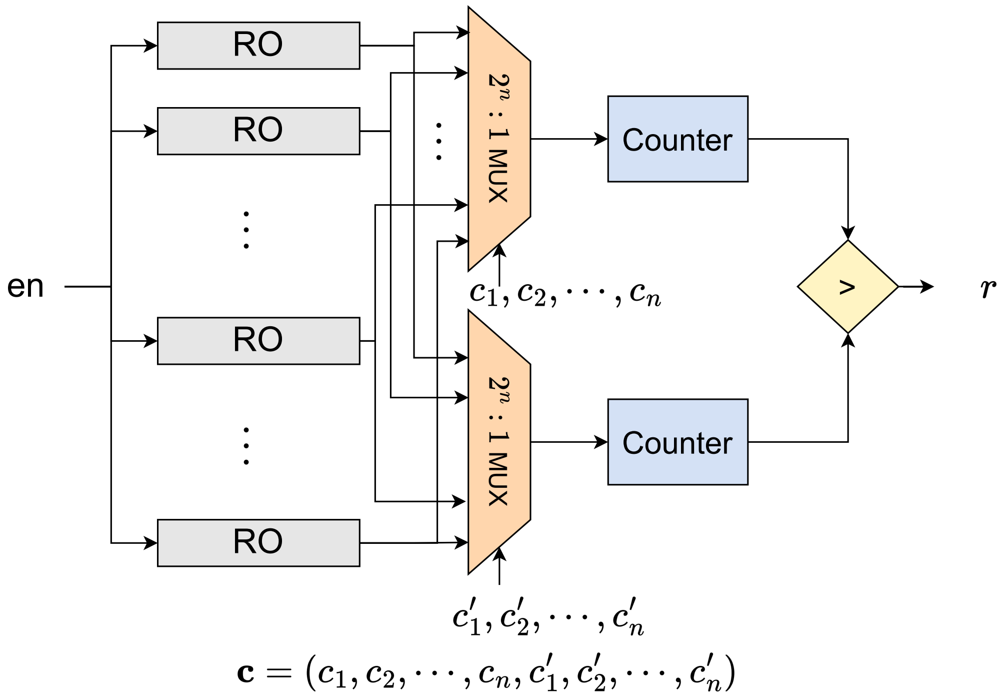

en
RO
RO
RO
RO
2^n : 1 MUX
2^n : 1 MUX
Counter
Counter
>
r
$c_1, c_2, \cdots, c_n$
$c'_1, c'_2, \cdots, c'_n$
$\mathbf{c} = (c_1, c_2, \cdots, c_n, c'_1, c'_2, \cdots, c'_n)$

Figure 14: Block diagram of Ring Oscillator PUF

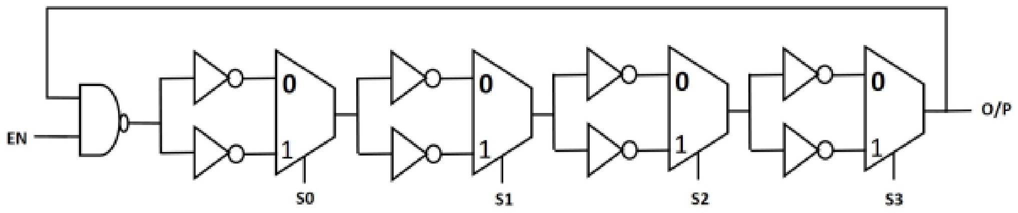


Figure 15: Configurable Ring Oscillator [11]

2.13 ColPUF [12]

Primitive components:

1. Congurable Ring Oscillator PUF (`CROPUF`)
2. LFSR (`FIBO_LFSR`)

Representation:

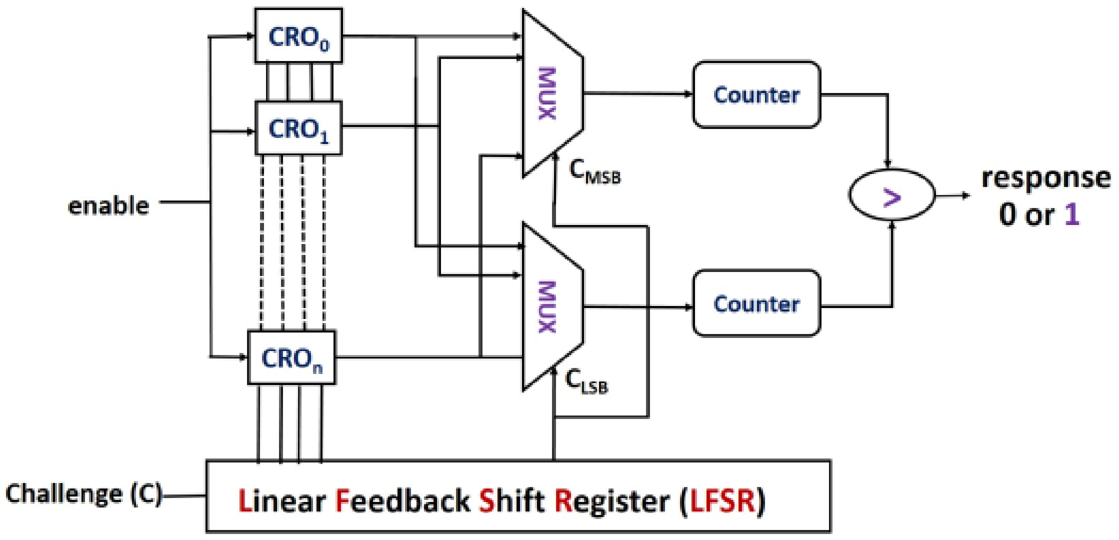


Figure 16: Block diagram of ColPUF [12]

Algorithm 20: Structural Representation of ColPUF

```
Input parameters:
    – Number of Configurable Ring Oscillators (CRO) (n)
    – Number of inverters in a CRO (m)
    – Challenge bits (cs = ⟨cs_1, . . . , cs_n⟩)
    – Enable bit (en)
Output parameters:
    – Response bit (colpuf_out)
Internal variables:
    – Signal line at the input of each CRO (t)
    – Challenge generated by LFSR (c)

Structural Design:
begin COLPUF ( num n, num m, vec cs, bit en )
c = FIBO_LFSR (n, cs);
colpuf_out = CROPUF (n, m, c, en);
return ( bit colpuf_out );
end COLPUF
```

2.14 Secure Configuration using Bent function [13]

Primitive components:

1. Arbiter PUF (APUF)
2. Bent function (BENT_FUNC)

Representation:

Algorithm 21: Structural Representation of Secure PUFs with Bent-Function

```
Input parameters:
    – Number of stages/switches in Arbiter PUF (n)
    – Challenge bits ($c = \langle c_1, \ldots, c_n \rangle$)
    – Number of Arbiter PUFs (k)
    – Enable bit (en)
Output parameters:
    – Response bit (bentpuf_out)
Internal variables:
    – Response from APUFs ($y = \langle y_1, \ldots, y_k \rangle$)

Structural Design:
begin BENTPUF ( num n, vec c,
                num k, bit en )
parallel $i = 1$ to k do
    $y_i$ = APUF (n, c, en);
end parallel
bentpuf_out = BENT_FUNC (k, y);
return ( bit bentpuf_out );
end BENTPUF

begin BENT_FUNC (num k, vec y)
parallel $i = 1$ to $k - 1$ do
    bf_out = bf_out $\oplus$ ($y_i$ and $y_{i+1}$);
    $i = i + 2$;
end parallel
return ( bit bf_out );
end BENT_FUNC
```

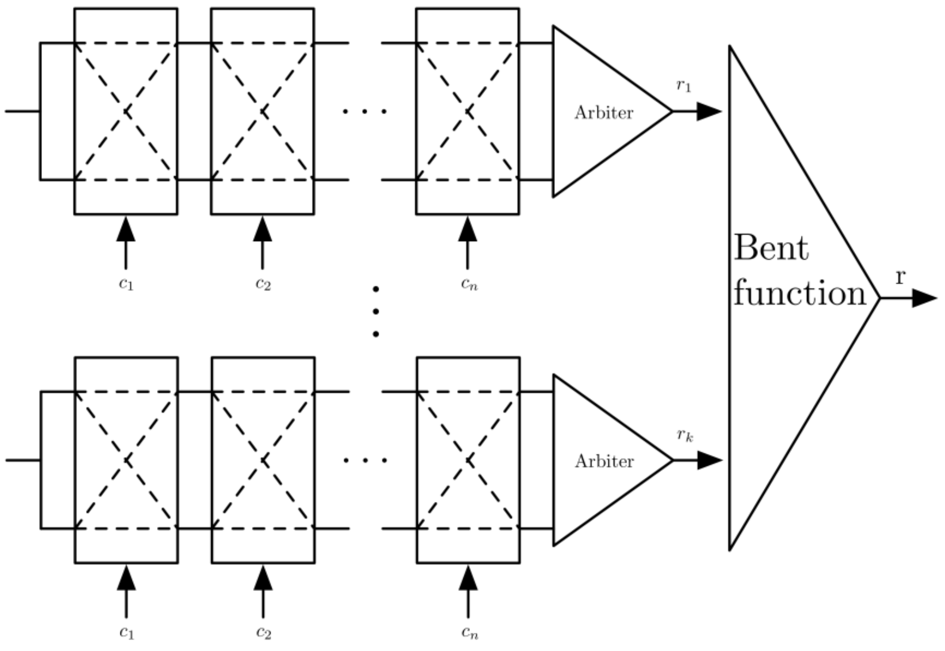


Figure 17: Secure PUF configuration using Bent function [13]

2.15 S_n-PUF construction using M-M Bent function [14]

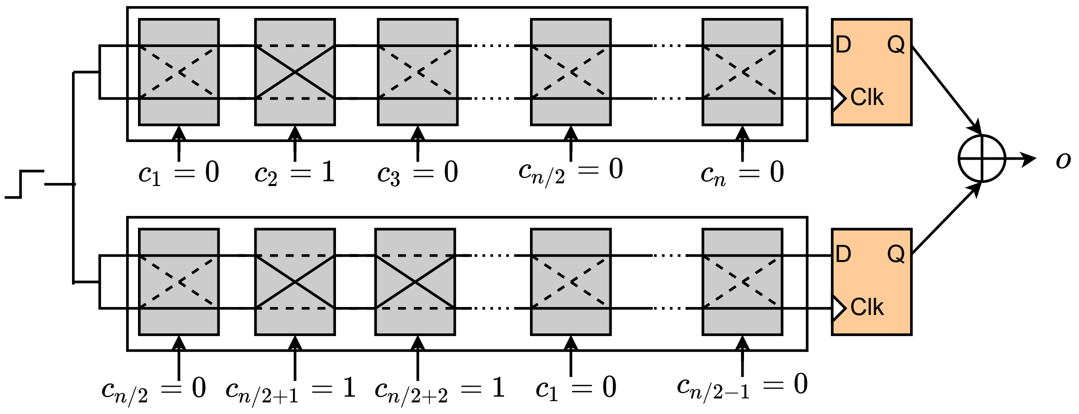
Primitive components:

1. Arbiter PUF (APUF)
2. M-M Bent function (BENT_FUNC)

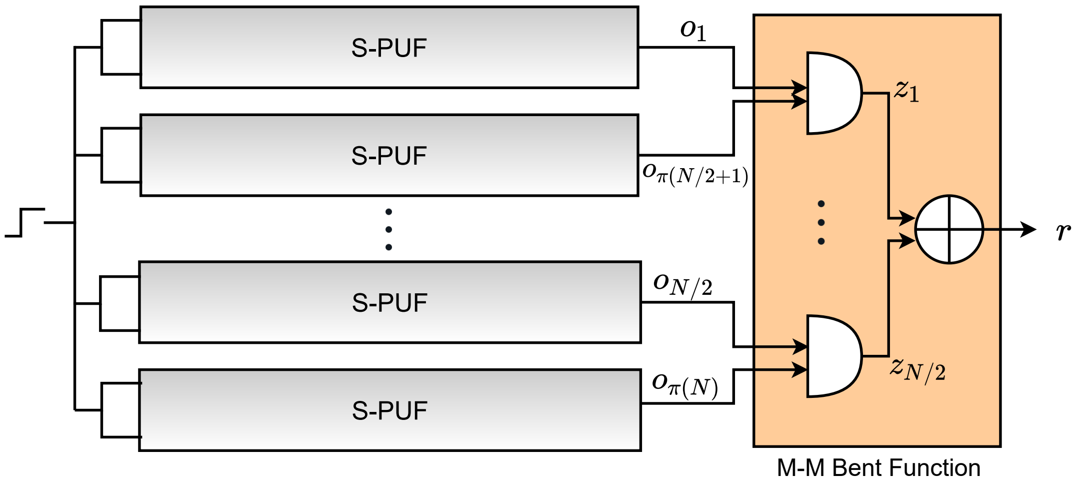
Representation:

Algorithm 22: Structural Representation of S_n-PUF

```
Input parameters:
    – Number of stages/switches in Arbiter PUF (N)
    – Challenge bits ($c = \langle c_1, \ldots, c_N \rangle$)
    – Number of SPUFs (n)
    – Enable bit (en)
Output parameters:
    – Response bit (snpuf_out)
Internal variables:
    – Response from SPUFs ($y = \langle y_1, \ldots, y_k \rangle$)

Structural Design:
begin SNPUF ( num n, num N, vec c,
              num k, bit en )
parallel $i = 1$ to n do
    $y_i$ = SPUF (N, c, en);
end parallel
snpuf_out = BENT_FUNC (n, y);
return ( bit snpuf_out );
end SNPUF

begin SPUF ( num N, vec c,
             bit en )
cshift = 0;
$y_1$ = APUF (N, c, en);
serial $i = 1$ to N do
    $cshift_i = c_{i+\frac{N}{2}}$;
end serial
$y_2$ = APUF (n, chift, en);
spuf_out = $y_1$XOR$y_2$;
return ( bit spuf_out );
end SNPUF

begin BENT_FUNC (num N, vec y)
parallel $i = 1$ to $N - 1$ do
    bf_out = bf_out $\oplus$ ($y_i$ and $y_{i+1}$);
    $i = i + 2$;
end parallel
return ( bit bf_out );
end BENT_FUNC
```



(a) S-PUF



(b) S_n-PUF

Figure 18: Block diagram of S-PUF and S_n-PUF

2.16 Bistable Ring PUF [15]

Primitive components:

1. NOR gate
2. MUX (`MUX`)
3. DEMUX (`DEMUX`)

Representation:

Algorithm 23: Structural Representation of Bistable Ring PUF

```
*Input parameters:*
    – Number of stages in Bistable Ring PUF (n)
    – Challenge bits ($c = \langle c_1, \ldots, c_n \rangle$)
    – Reset bit (reset)
*Output parameters:*
    – Response bit (brpuf_out)

*Structural Design:*
**begin** BRPUF ( **num** n, **vec** c, **bit** reset )
$d_1 = 0$;
**serial** $i = 1$ **to** $n - 1$ **do**
    $(t_i, b_i) = \text{DEMUX}(d_i, c_i)$;
    $t'_i = \text{NOR}(t_i, \text{reset})$;
    $b'_i = \text{NOR}(b_i, \text{reset})$;
    $d_{i+1} = \text{MUX}(t'_i, b'_1, c_i)$;
**end serial**
$d_1 = d_n$;
$\text{brpuf\_out} = d_n$;
**return** ( **bit** brpuf_out );
**end** BRPUF
```

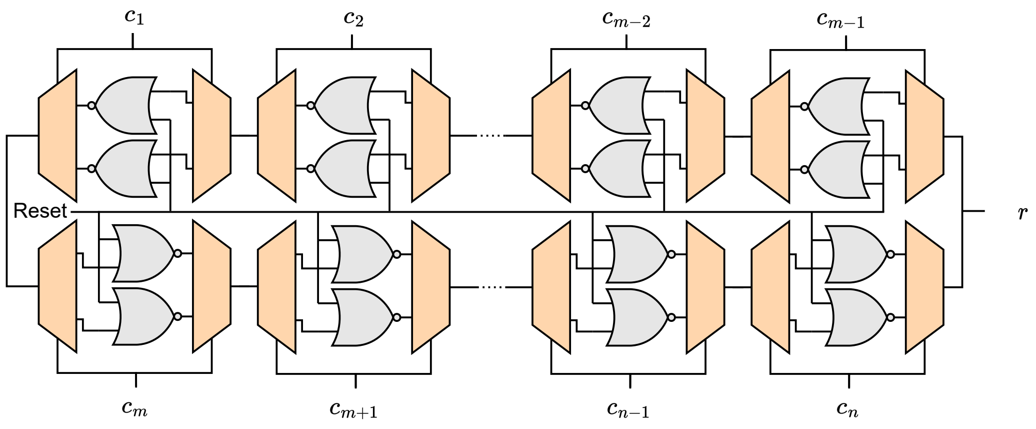


Figure 19: Block diagram of Bistable Ring PUF

3 Compositions of Weak and Strong PUFs

3.1 Multi-PUF (MPUF) using Pico-PUF and APUF [16]

Primitive components:

1. Pico PUF (PICOPUF)
2. XOR gate ($\oplus$)
3. APUF (APUF)

Representation:

Algorithm 24: Structural Representation of MultiPUF with Pico-PUF and APUF

Input parameters:
– Number of stages/switches in Arbiter PUF ($\mathtt{n}$)
– Challenge bits ($\mathtt{c} = \langle \mathtt{c}_1, \ldots, \mathtt{c}_\mathtt{n} \rangle$)
– Enable bit ($\mathtt{en}$)

Output parameters:
– Response bit (multipuf_out)

Internal variables:
– Response from Pico-PUF ($\mathtt{k}$)

Structural Design:

```
begin MULTIPUF ( num n, vec c,
                 bit en )
parallel $i = 1$ to n do
    $\mathtt{k_i} = \mathtt{PICOPUF}(\mathtt{en})$;
    $\mathtt{k_i} = \mathtt{k_i} \oplus \mathtt{c_i}$;
end parallel
multipuf_out = APUF (n, k, en);
return ( bit multipuf_out );
end MULTIPUF
```

```
begin PICOPUF ( bit en )
$\mathtt{a}_1 = \mathtt{ARBITER}(1, \mathtt{en})$;
$\mathtt{a}_2 = \mathtt{ARBITER}(1, \mathtt{en})$;
picopuf_out = NAND_LATCH ($\mathtt{a}_1, \mathtt{a}_2$);
return ( bit picopuf_out );
end PICOPUF

begin NAND_LATCH (bit $\mathtt{a}_1$, bit $\mathtt{a}_2$)
$\mathtt{b}_1$ = not $\mathtt{a}_1$ or ($\mathtt{a}_2$ and $\mathtt{b}_1$);
return ( bit $\mathtt{b}_1$ );
end NAND_LATCH
```

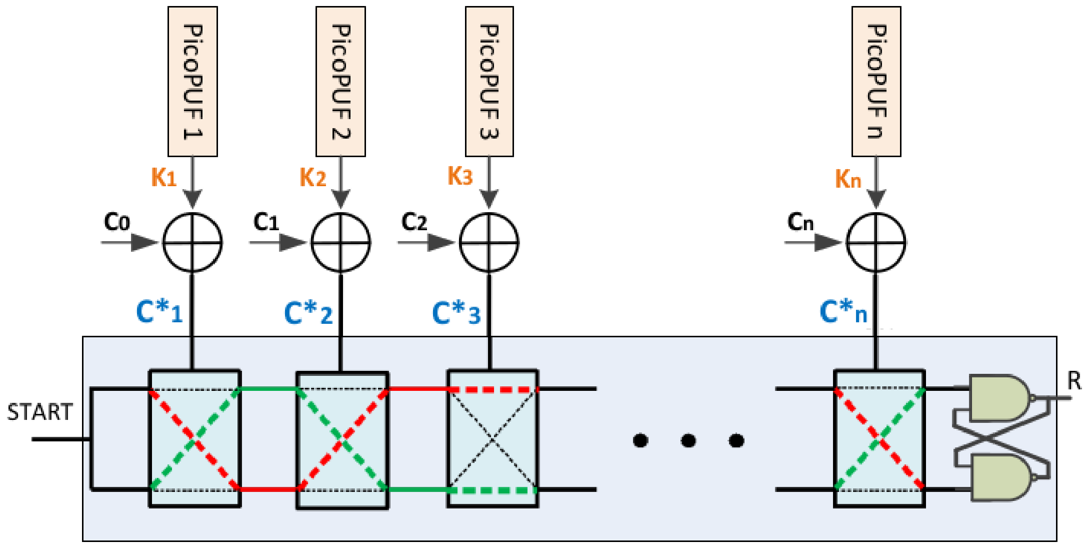


Figure 20: Block diagram of Multi-PUF with Pico-PUF and APUF [16]

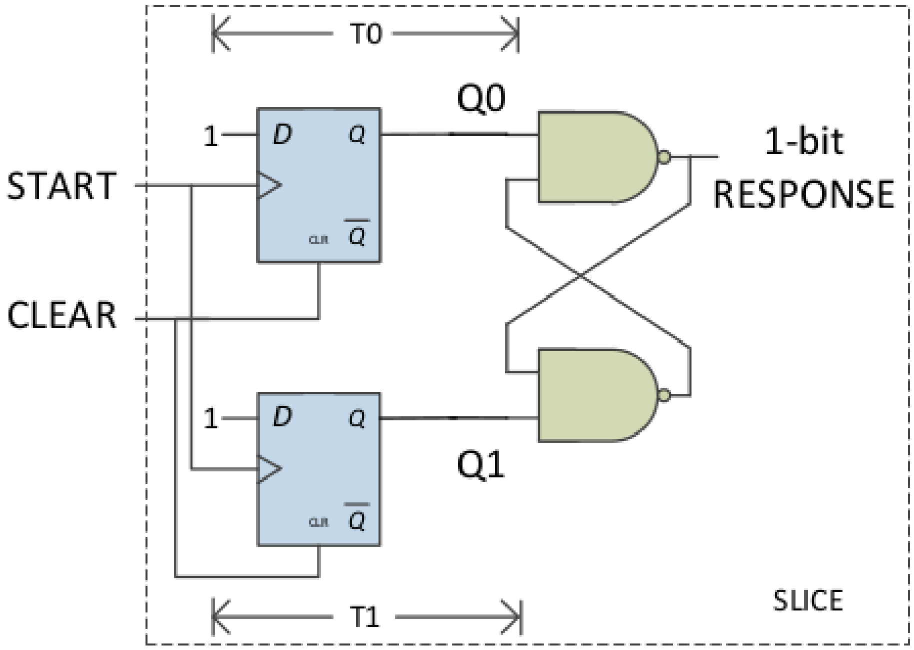


Figure 21: Block diagram of Pico-PUF [16]

4 Recurrent Composition of PUFs

4.1 Recurrent composition of DAPUF

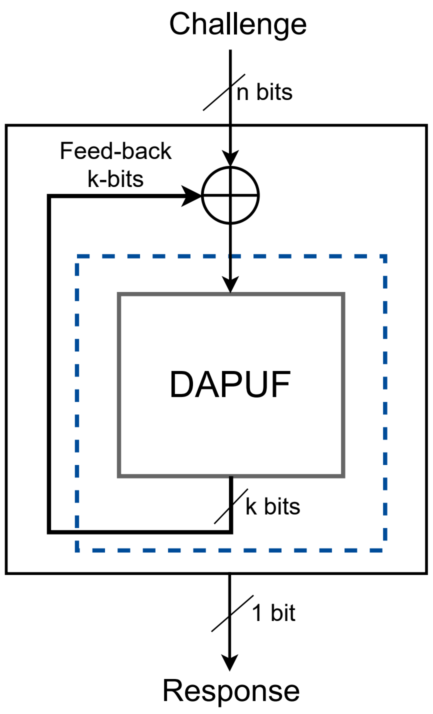


Figure 22: Block diagram of Recurrent-DAPUF

Primitive components:

1. DAPUF (`DAPUF`)
2. XOR gate ($\oplus$)

Representation:

Algorithm 25: Structural Representation of Recurrent-DAPUF

```
Input parameters:
    – Number of stages/switches in DAPUF ($\mathtt{n}$)
    – Number of delay chains in DAPUF ($\mathtt{m}$)
    – Challenge bits ($\mathtt{c} = \langle \mathtt{c_1}, \ldots, \mathtt{c_n} \rangle$)
    – Enable bit ($\mathtt{en}$)
Output parameters:
    – Response bit ($\mathtt{r\_out}$)
Internal variables:
    – Response bits from DAPUF obtained in first iteration ($\mathtt{r_{int}} = \langle \mathtt{r_{int,1}}, \mathtt{r_{int,2}}, \ldots, \mathtt{r_{int,m}} \rangle$)
    – Internal Challenge ($\mathtt{c_{int}}$)

Structural Design:

begin Recurrent-DAPUF ( num n, vec c, bit en )
$\mathtt{r_{int}} = \mathtt{DAPUF(k, n, c, en)}$;
parallel $i = 1$ to $n/m$ do
    parallel $j = 1$ to $m$ do
        $\mathtt{c_{int,(i-1)*m+j}} = \mathtt{c_{int,(i-1)*m+j}} \oplus \mathtt{r_{int,j}}$;
    end parallel
end parallel
$\mathtt{r_{out}} = \mathtt{DAPUF(k, n, c_{int}, en)}$;
return ( bit r_out );
end Recurrent-DAPUF

begin DAPUF ( num k, num n,
              vec c, bit en )
parallel $i = 1$ to $k$ do
    $\langle \mathtt{t_i}, \mathtt{b_i} \rangle = \mathtt{DELAY\text{-}CHAIN\ (n, en, c)}$;
end parallel
$\mathtt{xor\_cnt} = \mathtt{MATH\text{-}CEIL\ (k * (k-1)/m)}$;
$g = 1$; $h = 1$; $\mathtt{r_h} = 0$;
serial $i = 1$ to $k - 1$ do
    serial $j = i + 1$ to $k$ do
        $\mathtt{a^{top}} = \mathtt{ARBITER\ (t_i, t_j)}$;
        $\mathtt{r_h} = \mathtt{r_h} \oplus \mathtt{a^{top}}$;
        $g = g + 1$;
        if $g > \mathtt{xor\_cnt}$ then
            $h = h + 1$; $r_h = 0$; $g = 1$;
        end if
    end serial
end serial
serial $i = 1$ to $k - 1$ do
    serial $j = i + 1$ to $k$ do
        $\mathtt{a^{bot}} = \mathtt{ARBITER\ (b_i, b_j)}$;
        $\mathtt{r_h} = \mathtt{r_h} \oplus \mathtt{a^{bot}}$;
        $g = g + 1$;
        if $g > \mathtt{xor\_cnt}$ then
            $h = h + 1$; $r_h = 0$; $g = 1$;
        end if
    end serial
end serial
return ( vec $\langle \mathtt{r_1}, \ldots, \mathtt{r_m} \rangle$ );
end DAPUF
```

5 Learnability Analysis of PUF Compositions

Several PUF architectures have been analysed in the PAC Learning framework [17,18]. Here we present the PAC learnability bounds for a recurrent composition of DAPUF, termed as Recurrent-DAPUF.

5.1 PAC Learning Analysis of Recurrent Composition of PUF

We show that a Recurrent-DAPUF can be represented using a Linear Threshold Function (LTF) and can be learned using the PAC variant of Perceptron algorithm. We then derive the PAC learnability bounds for the composition using the mistake bound of the learning algorithm. A k-chain DAPUF, taking an n-bit challenge and producing an m-bit response consists of $k(k-1)$ arbiters whose outputs are fed into m XOR gates. Thus it is mathematically equivalent to m XOR-APUFs, each consisting of $k(k-1)/m$ APUFs. Linear Threshold function (LTF)-based representation for XOR APUFs has been widely adopted in [19,20], and can be used to represent DAPUF as well. Analogous to PAC learning of XOR-APUF, a DAPUF represented by m independent LTFs can be learned by the PAC variant of Perceptron Algorithm. The sample complexity of the PAC learning algorithm depends on the mistake bound of the Perceptron algorithm and is given by $\mathcal{O}(1/\epsilon(log(1/\delta) + N_{mis}))$. The upper bound on the number of mistakes that can be made by the Perceptron

Algorithm is $N_{mis} = \big(R/\epsilon\big)^2$ where R is the length of the transformed challenge vector ϕ and ϵ is the error bound specific to PAC model. Therefore the number of CRPs required to learn one bit of DAPUF response is $\mathcal{O}\Big(\frac{1}{\epsilon}\big(log(\frac{1}{\delta}) + \frac{(nd)^2(n+1)^{k(k-1)/m}}{\epsilon}\big)\Big)$ where d is the discretized delay value calculated as given in [20].

In case of Recurrent-DAPUF, the challenge (c) is XORed with the intermediate response (r_{int}) obtained from the DAPUF, and fed to the DAPUF to obtain the final response. Since r_{int} is hidden, the challenge applied in the second iteration (c_{int}) gets obfuscated thereby increasing the complexity of challenge response relationship. For the analysis, we have used 5-4 DAPUF as the core PUF which takes a 64-bit challenge and returns a 4-bit response. In this composition, each response bit is XORed with 16 consecutive challenge bits. This implies that each of the 16-bit sub challenge fed to the core DAPUF is either equal to or is a complement of the corresponding part of the challenge given to the Rec-DAPUF, depending on whether the XORed response bit is 0 or 1. Thus, the challenge applied after XOR operation has either 0, 16, 32, 48 or 64 bits flipped as compared to the Rec-DAPUF input, depending on the Hamming weight of r_{int}.

Extending the response bias calculation presented in [14], we obtain the bias[1] of a single response bit of k-chain DAPUF, when b (even) consecutive challenge bits are flipped to be $\eta = \frac{1}{2} + 2^{(k(k-1)/m)-1}\Big(\frac{1}{2} - \frac{2}{\pi}tan^{-1}\sqrt{\frac{b}{2n-b}}\Big)^{k(k-1)/m}$, where m is the length of r_{int}. For 0, 16, 32, 48 and 64 bit flips, let us denote the corresponding response bias as $\eta_0 = 1$, η_1, η_2, η_3 and η_4 respectively.

With the knowledge of intermediate challenge (c_{int}), Recurrent-DAPUF can be accurately modelled using LTF representation as described above. Since we do not have the c_{int}, we assume the intermediate challenge to be equal to Recurrent-DAPUF input c, on the same lines as given in [21]. This assumption eliminates the feedback and reduces the Recurrent-DAPUF to a DAPUF. Hence, the resultant PUF can be represented by an LTF, as we select one out of the 4 output bits. The next step is to calculate the impact of the difference in the assumed (c) and the actual challenge (c_{int}) on the response bit (r). We estimate the impact on the final response using the response bias as explained below. Let h be the hypothesis obtained from the learning algorithm for Recurrent-DAPUF, after observing a set of labelled examples (training set) of the form (c, r). The probability that an example (c, r) (not belonging to the training set) disagrees with hypothesis h is calculated as follows:

$$\begin{aligned} Pr[h(c) \neq r] &= \sum_{i=0}^{4} Pr[h(c) \neq r \,|\, |r_{int}| = i].Pr[|r_{int}| = i] \\ &= \sum_{i=0}^{4} \binom{4}{i}.\frac{\big(\epsilon.\eta_i + (1-\epsilon)(1-\eta_i)\big)}{16} \\ &= \epsilon\Big(\sum_{i=0}^{4} \binom{4}{i}.\frac{(2\eta_i - 1)}{16}\Big) \\ &+ \Big(\sum_{i=0}^{4} \binom{4}{i}.\frac{(1-\eta_i)}{16}\Big) = \epsilon.\eta' + \eta'' \end{aligned} \tag{1}$$

For this analysis, it is assumed that the distribution of r_{int} is uniform. It is to be noted that Recurrent-DAPUF returns r when core-DAPUF is given challenge c_{int}. Thus, the error of the hypothesis (h) is estimated for c_{int} and the bits differing between c_{int} and $c_{int} \oplus c$ can be considered as noise. The probability of mismatch between the predicted and actual response for a given r_{int} is the sum of two probabilities: i) probability that the predicted and actual response differ on challenge c_{int} and the bits differing between c and c_{int} do not impact the final response bit. ii) probability that $h(c_{int})$ is equal to r, however the response gets flipped due to the difference between c and c_{int}. When hypothesis h becomes equal to the target function, we have $(\epsilon = 0) \implies Pr[h(c) \neq r] = \eta''$, since error occurs only due to the difference between c and c_{int}. Thus the updated margin of the hypothesis becomes $\epsilon.\eta'$.

The maximum number of mistakes that can be made by the Perceptron algorithm is polynomial in the separation $(\epsilon.\eta')$ and is given by $N_{mis} = \Big(\frac{R}{\epsilon.\eta'}\Big)^2$ where $R = (n+1)^{k(k-1)/m}$ is the length of transformed chal-

[1]Bias refers to the probability that the PUF response remains unchanged on modifying one or more bits of the input challenge and is denoted by η.

lenge vector and the length of weight vector corresponding to hypothesis output by the Perceptron algorithm is one. Thus the sample complexity to PAC learn Recurrent-DAPUF is $\mathcal{O}\Big(\frac{1}{\epsilon.\eta'}\big(log(\frac{1}{\delta})+\frac{(nd)^2(n+1)^{k(k-1)/m}}{\epsilon^2.\eta'^2}\big)\Big)$, where d is the discretized delay value and ϵ, δ are the PAC model parameters. Since $\eta' < 1$, the number of CRPs required to learn Recurrent-DAPUF is approximately $1/\eta'$ times more than DAPUF, thus proving that addition of feedback increases ML robustness.

References

[1] Daihyun Lim, Jae W Lee, Blaise Gassend, G Edward Suh, Marten Van Dijk, and Srinivas Devadas. Extracting secret keys from integrated circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(10):1200–1205, 2005.

[2] G Edward Suh and Srinivas Devadas. Physical unclonable functions for device authentication and secret key generation. In *2007 44th ACM/IEEE Design Automation Conference*, pages 9–14. IEEE, 2007.

[3] Jae W Lee, Daihyun Lim, Blaise Gassend, G Edward Suh, Marten Van Dijk, and Srinivas Devadas. A technique to build a secret key in integrated circuits for identification and authentication applications. In *2004 Symposium on VLSI Circuits. Digest of Technical Papers (IEEE Cat. No. 04CH37525)*, pages 176–179. IEEE, 2004.

[4] SV Sandeep Avvaru, Ziqing Zeng, and Keshab K Parhi. Homogeneous and heterogeneous feed-forward xor physical unclonable functions. *IEEE Transactions on Information Forensics and Security*, 15:2485–2498, 2020.

[5] Takanori Machida, Dai Yamamoto, Mitsugu Iwamoto, and Kazuo Sakiyama. A new mode of operation for arbiter puf to improve uniqueness on fpga. In *Fed. Conf. on Comp. Sc. and Infor. Sys.*, pages 871–878, 2014.

[6] Durga Prasad Sahoo, Debdeep Mukhopadhyay, Rajat Subhra Chakraborty, and Phuong Ha Nguyen. A multiplexer-based arbiter puf composition with enhanced reliability and security. *IEEE Transactions on Computers*, 67(3):403–417, 2017.

[7] Phuong Ha Nguyen, Durga Prasad Sahoo, Chenglu Jin, Kaleel Mahmood, Ulrich Rührmair, and Marten van Dijk. The interpose puf: Secure puf design against state-of-the-art machine learning attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 243–290, 2019.

[8] Durga Prasad Sahoo, Sayandeep Saha, Debdeep Mukhopadhyay, Rajat Subhra Chakraborty, and Hitesh Kapoor. Composite puf: A new design paradigm for physically unclonable functions on fpga. In *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 50–55. IEEE, 2014.

[9] Mehrdad Majzoobi, Farinaz Koushanfar, and Miodrag Potkonjak. Lightweight secure pufs. In *2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 670–673. IEEE, 2008.

[10] Elena Dubrova, Oscar Näslund, Bernhard Degen, Anders Gawell, and Yang Yu. Crc-puf: A machine learning attack resistant lightweight puf construction. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 264–271. IEEE, 2019.

[11] Abhranil Maiti and Patrick Schaumont. Improved ring oscillator puf: An fpga-friendly secure primitive. *Journal of cryptology*, 24(2):375–397, 2011.

[12] B Srinivasu, P Vikramkumar, Anupam Chattopadhyay, and Kwok-Yan Lam. Colpuf: a novel configurable lfsr-based puf. In *2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pages 358–361. IEEE, 2018.

[13] On Arbiter PUFs. Security analysis of strong physical unclonable functions. 2017.

[14] Akhilesh Anilkumar Siddhanti, Srinivasu Bodapati, Anupam Chattopadhyay, Subhamoy Maitra, Dibyendu Roy, and Pantelimon Stanica. Analysis of the strict avalanche criterion in variants of arbiter-based physically unclonable functions. In *Progress in Cryptology - INDOCRYPT 2019 - 20th International Conference on Cryptology in India, Hyderabad, India, December 15-18, 2019, Proceedings*, volume 11898 of *Lecture Notes in Computer Science*, pages 556–577. Springer, 2019.

[15] Qingqing Chen, György Csaba, Paolo Lugli, Ulf Schlichtmann, and Ulrich Rührmair. The bistable ring puf: A new architecture for strong physical unclonable functions. In *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*, pages 134–141. IEEE, 2011.

[16] Qingqing Ma, Chongyan Gu, Neil Hanley, Chenghua Wang, Weiqiang Liu, and Maire O'Neill. A machine learning attack resistant multi-puf design on fpga. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 97–104. IEEE, 2018.

[17] Durba Chatterjee, Debdeep Mukhopadhyay, and Aritra Hazra. PUF-G: A CAD framework for automated assessment of provable learnability from formal PUF representations. In *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2020, San Diego, CA, USA, November 2-5, 2020*, pages 48:1–48:9. IEEE, 2020.

[18] Fatemeh Ganji. *On the Learnability of Physically Unclonable Functions.* Springer, 2018.

[19] U. Rührmair, F. Sehnke, J. Sölter, G. Dror, S. Devadas, and J. Schmidhuber. Modeling Attacks on Physical Unclonable Functions. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 237–249, New York, NY, USA, 2010. ACM.

[20] Fatemeh Ganji, Shahin Tajik, and Jean-Pierre Seifert. Why attackers win: on the learnability of xor arbiter pufs. In *International Conference on Trust and Trustworthy Computing*, pages 22–39. Springer, 2015.

[21] Durba Chatterjee, Debdeep Mukhopadhyay, and Aritra Hazra. Interpose PUF can be PAC learned. *IACR Cryptol. ePrint Arch.*, 2020:471, 2020.