Systematic Use of Random Self-Reducibility in Cryptographic Code against Physical Attacks

Ferhat Erata[†], TingHung Chiu[★], Anthony Etim[†], Srilalith Nampally[★], Tejas Raju[★], Rajashree Ramu[★], Ruzica Piskac[†], Timos Antonopoulos[†], Wenjie Xiong[★] Jakub Szefer[†]

[†] *Yale University*, [★] *Virginia Tech*

2024 ACM/IEEE International Conference on Computer-Aided Design (ICCAD '24), October 17-31, 2024, New Jersey, USA

Random Self-Reducible Properties

[Blum, Luby, and Rubinfeld (BLR). 1993]

Blum - Turing award 1995

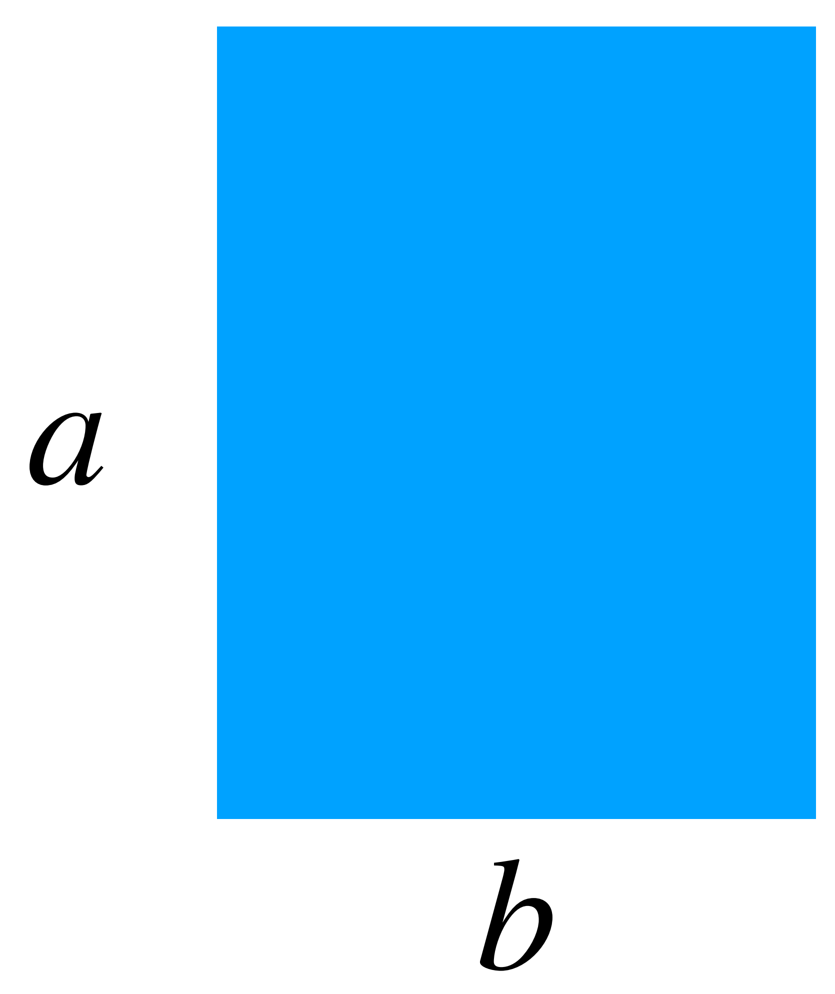
To calculate

$$P(a, b) = a \cdot b$$

you can instead calculate

$$P(a + r_1, b + r_2) - P(a, r_2) - P(b, r_1) - P(r_1, r_2)$$

for any random numbers r_1 **and** r_2



Random Self-Reducible Properties

[Blum, Luby, and Rubinfeld (BLR). 1993]

Blum - Turing award 1995

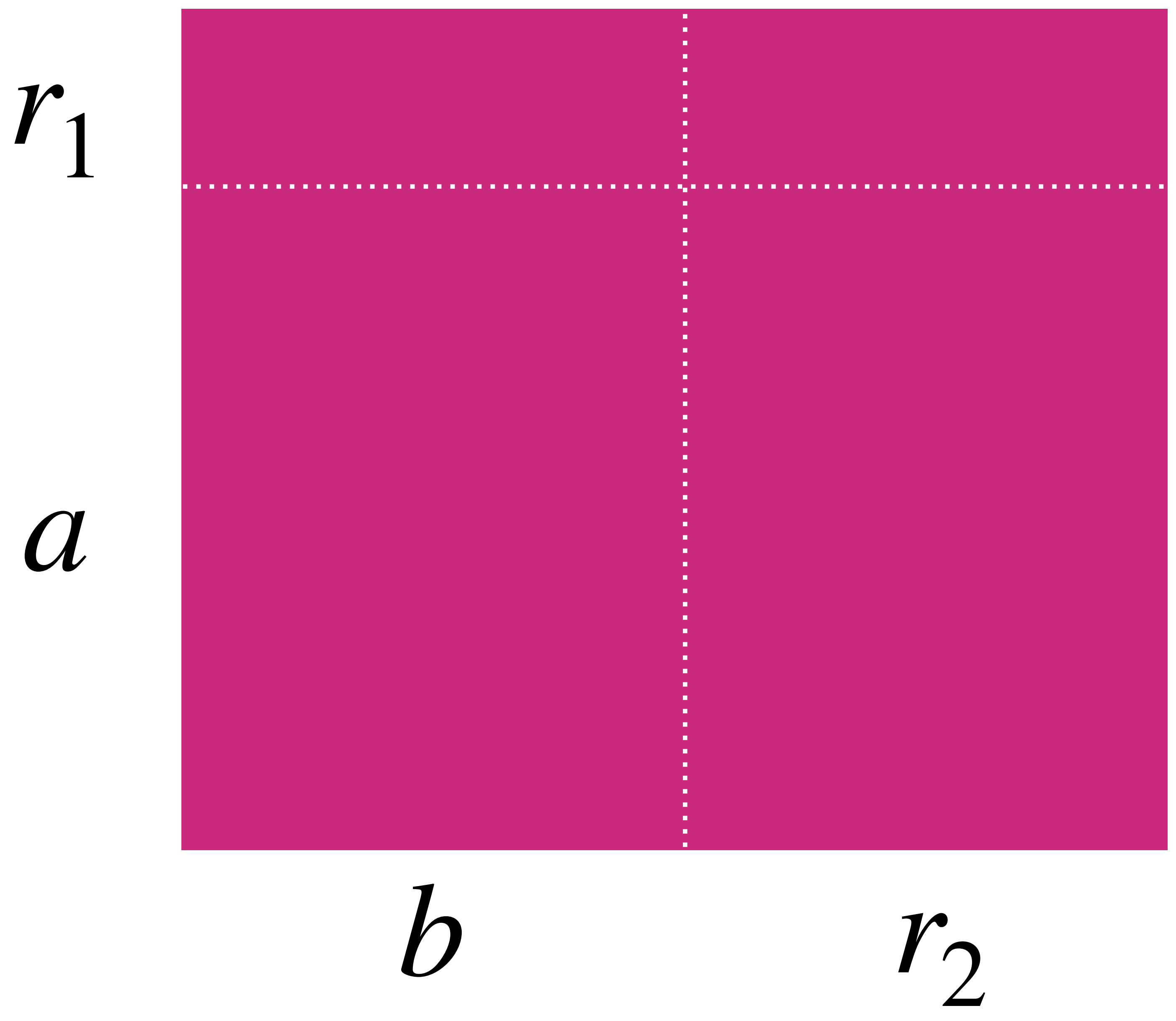
To calculate

$$P(a, b) = a \cdot b$$

you can instead calculate

$$\boxed{P(a + r_1, b + r_2)} - P(a, r_2) - P(b, r_1) - P(r_1, r_2)$$

for any random numbers r_1 **and** r_2



Random Self-Reducible Properties

[Blum, Luby, and Rubinfeld (BLR). 1993]

Blum - Turing award 1995

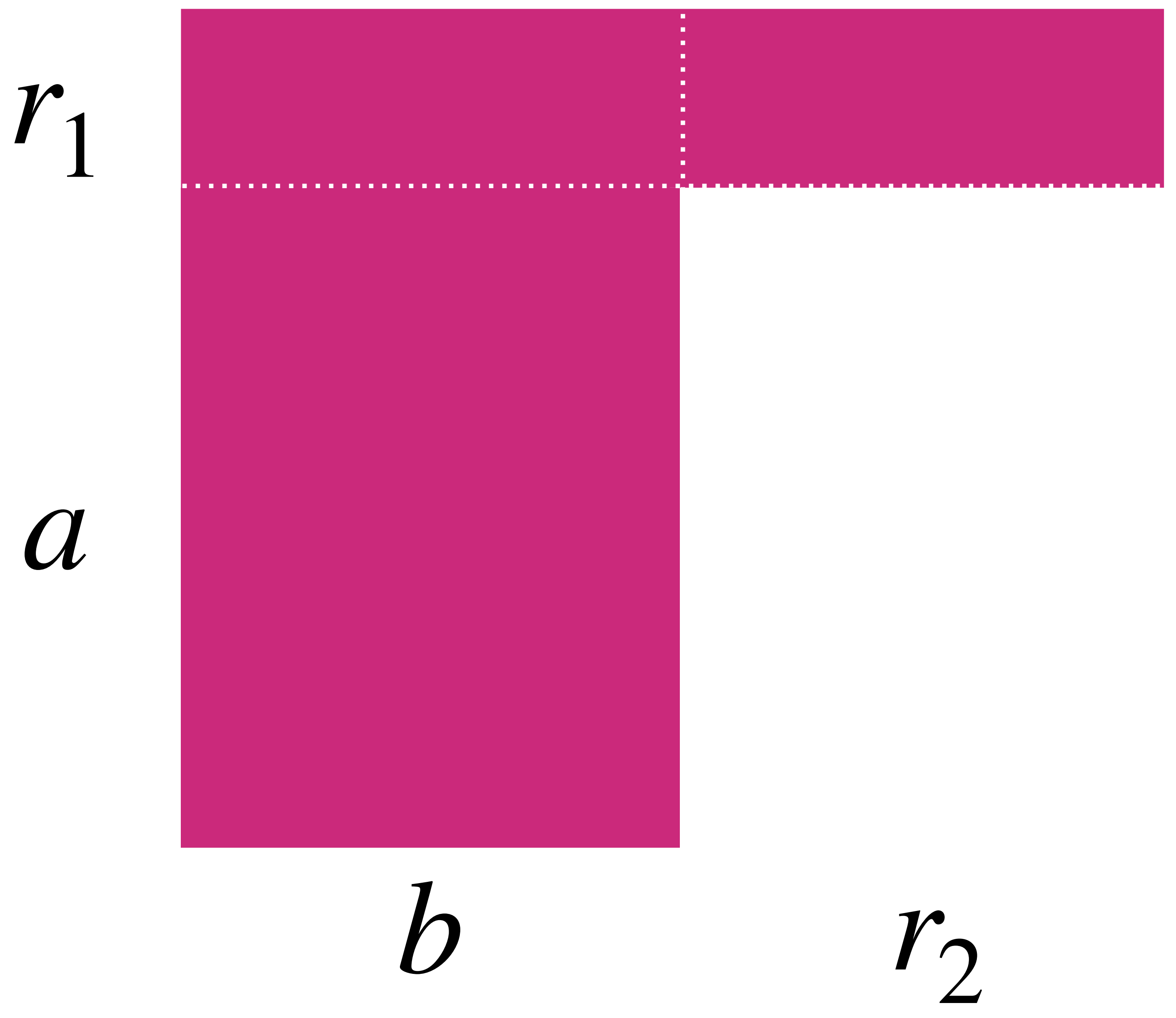
To calculate

$$P(a, b) = a \cdot b$$

you can instead calculate

$$P(a + r_1, b + r_2) \boxed{- P(a, r_2)} - P(b, r_1) - P(r_1, r_2)$$

for any random numbers r_1 and r_2



Random Self-Reducible Properties

[Blum, Luby, and Rubinfeld (BLR). 1993]

Blum - Turing award 1995

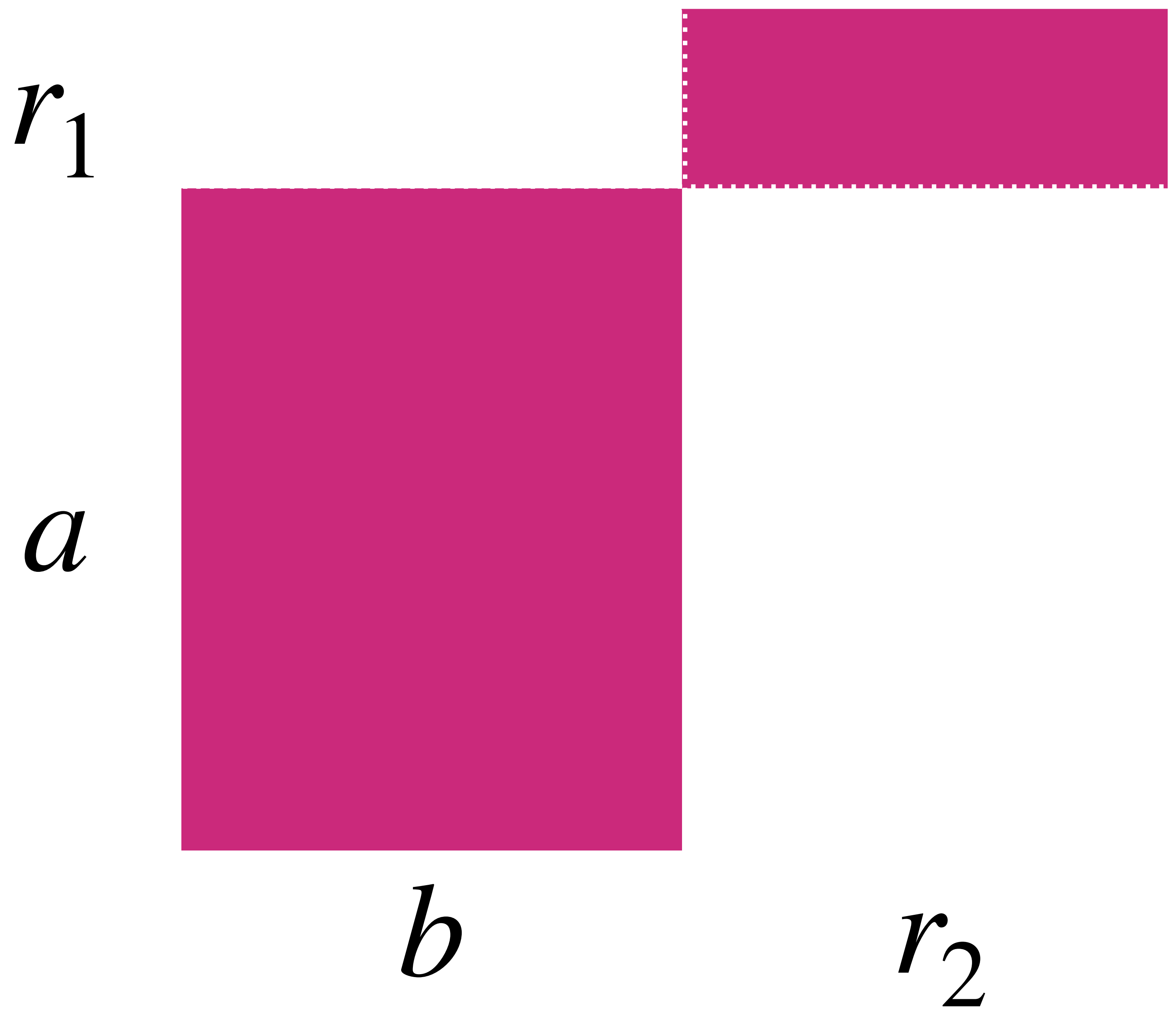
To calculate

$$P(a, b) = a \cdot b$$

you can instead calculate

$$P(a + r_1, b + r_2) - P(a, r_2) \boxed{- P(b, r_1)} - P(r_1, r_2)$$

for any random numbers r_1 and r_2



Random Self-Reducible Properties

[Blum, Luby, and Rubinfeld (BLR). 1993]

Blum - Turing award 1995

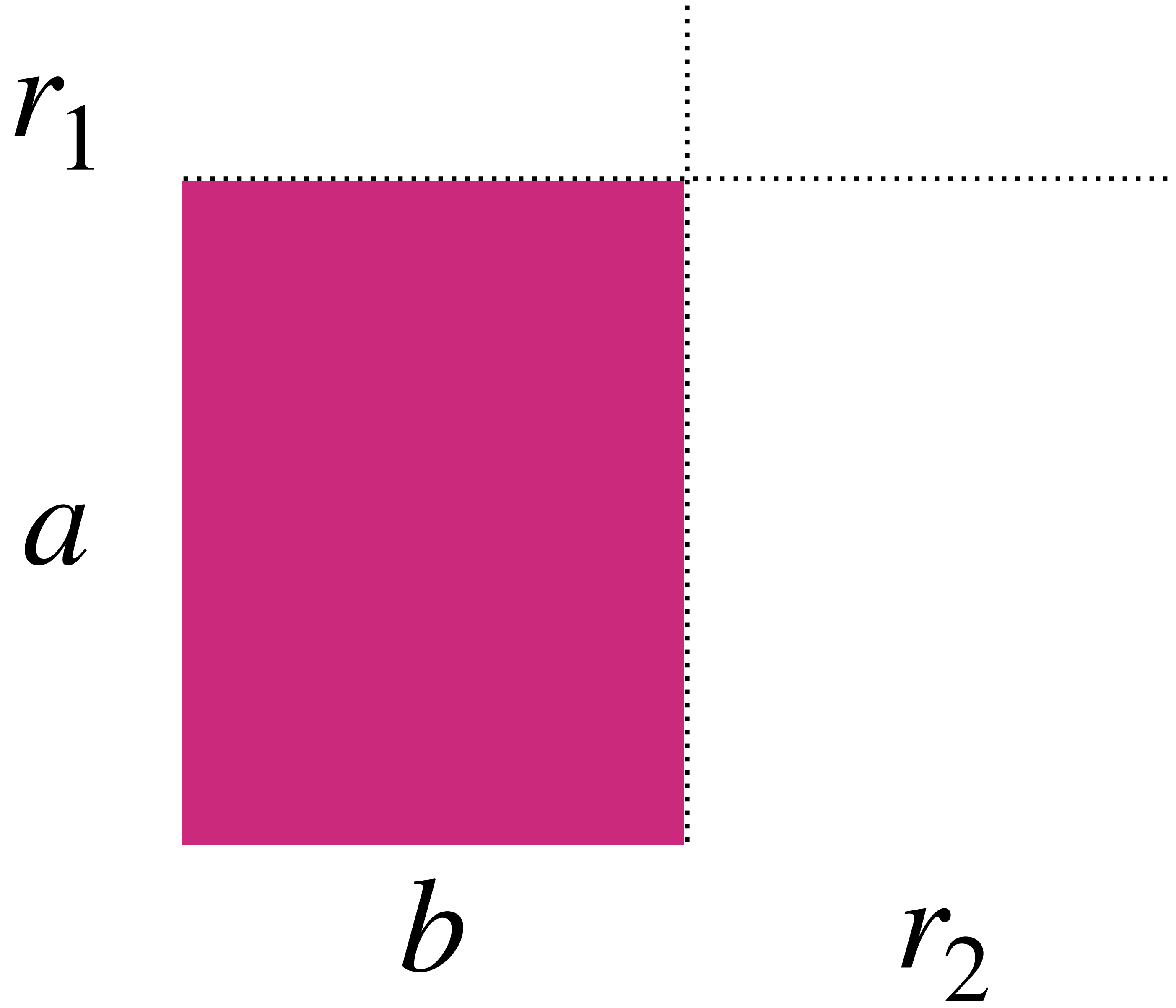
To calculate

$$P(a, b) = a \cdot b$$

you can instead calculate

$$P(a + r_1, b + r_2) - P(a, r_2) - P(b, r_1) \boxed{- P(r_1, r_2)}$$

for any random numbers r_1 **and** r_2



Random Self-Reducible Properties

[Blum, Luby, and Rubinfeld (BLR). 1993]

Blum - Turing award 1995

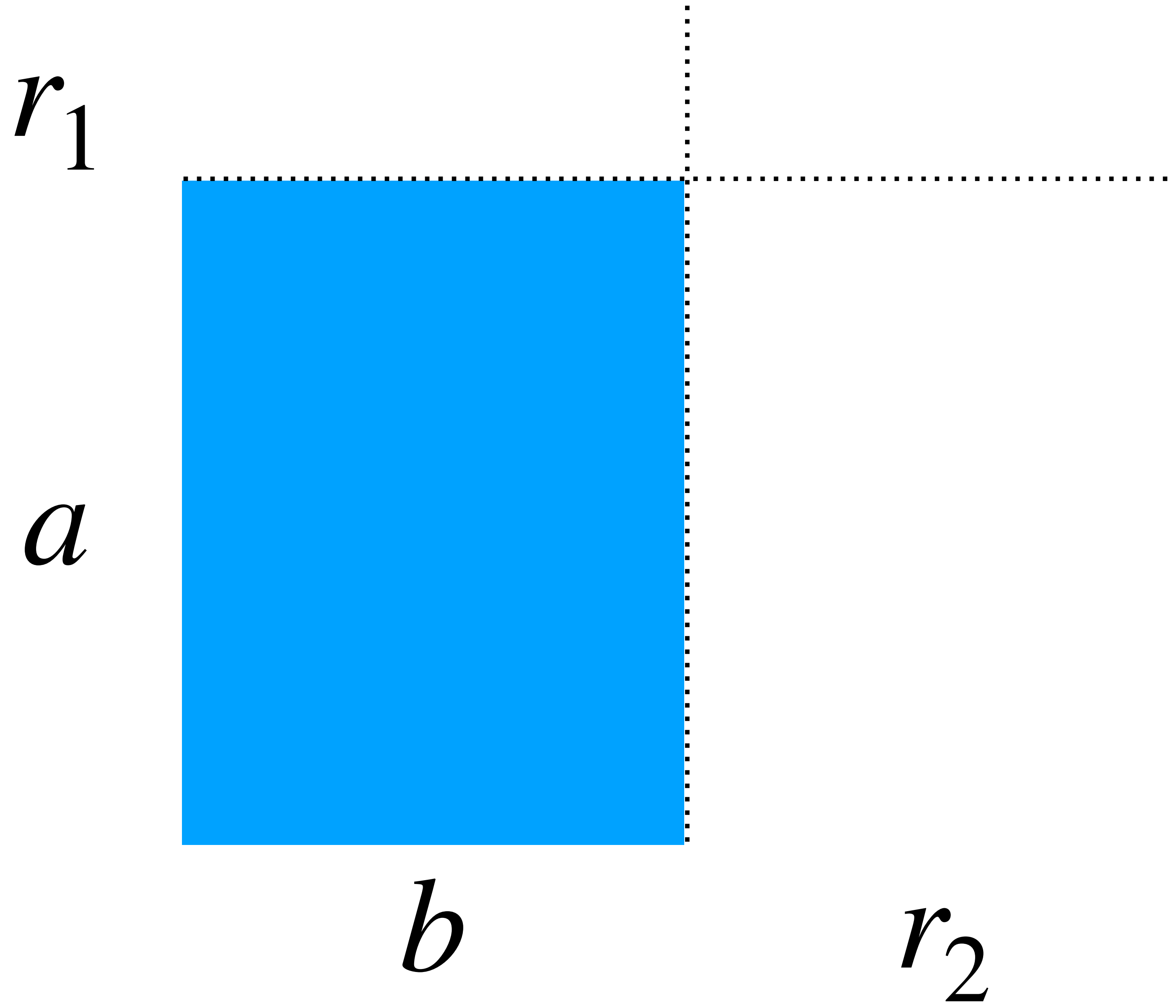
To calculate

$$P(a, b) = \boxed{a \cdot b}$$

you can instead calculate

$$P(a + r_1, b + r_2) - P(a, r_2) - P(b, r_1) - P(r_1, r_2)$$

for any random numbers r_1 **and** r_2



Random Self-Reducible Properties

[Blum, Luby, and Rubinfeld (BLR). 1993]

Blum - Turing award 1995

Program P	Function f	Property
Identity	$f(x) = x$	$P(x) = P(x + \tilde{r}) - P(\tilde{r})$
Integer Mult.	$f(x, y) = x \cdot y$	$P(x, y) = P(\tilde{x}_1, \tilde{y}_1) + P(\tilde{x}_1, \tilde{y}_2) + P(\tilde{x}_2, \tilde{y}_1) + P(\tilde{x}_2, \tilde{y}_2)$
Integer Mult.	$f(x, y) = x \cdot y$	$P(x, y) = P(x + \tilde{r}, y + \tilde{s}) - P(\tilde{r}, y + \tilde{s}) - P(x + \tilde{t}, \tilde{s}) + P(\tilde{t}, \tilde{s})$
Integer Div.	$f(x, R) = x \div R$	$P(x, y) = P(x_1, R) + P(x_2, R) + P\left(P_{\text{mod}}(x_1, R) + P_{\text{mod}}\,(x_2, R), R\right)$
Mod	$f(x, R) = x \bmod R$	$P(x, R) = P(\tilde{x}_1, R) +_R P(\tilde{x}_2, R)$
Mod Mult.	$f(x, y, R) = x \cdot_R y$	$P(x, y, R) = P(\tilde{x}_1, \tilde{y}_1, R) +_R P(\tilde{x}_2, \tilde{y}_1, R) +_R P(\tilde{x}_1, \tilde{y}_2, R) +_R P(\tilde{x}_2, \tilde{y}_2, R)$
Mod Exp.,	$f(a, x, R) = a^x \bmod R$	$P(a, x, R) = P(a, \tilde{x}_1, R) \cdot_R P(a, \tilde{x}_2, R)$
Mod Inv.,	$f(x, R) \cdot_R x = 1$	$P(x, R) = \tilde{w} \cdot_R P(x \cdot_R \tilde{w})$ where $P(\tilde{w}, R) \cdot_R \tilde{w} = 1$ and $P(x, R) \cdot_R x = 1$
Poly. Mult.	$f(p_x, q_x) = p_x \cdot q_x$	$P(p, q) = P(\tilde{p}_1, \tilde{q}_1) + P(\tilde{p}_2, \tilde{q}_1) + P(\tilde{p}_1, \tilde{q}_2) + P(\tilde{p}_2, \tilde{q}_2)$
Matrix Mult.	$f(A, B) = A \times B$	$P(A, B) \leftarrow P(\tilde{A}_1, \tilde{B}_1) + P(\tilde{A}_2, \tilde{B}_1) + P(\tilde{A}_1, \tilde{B}_2) + P(\tilde{A}_2, \tilde{B}_2)$
Matrix Inv.	$f(A) = A^{-1}$	$P(A) = \tilde{R} \times P(A \times \tilde{R})$ where A and $\tilde{R}$ are invertible n-by-n matrices.
Determinant	$f(A) = \det A$	$P(A) = P(\tilde{R})/P(A \times \tilde{R})$ where $\tilde{R}$ is invertible.

Systematic Use of Random Self-Reducibility against Physical Attacks

[Erata et al., ICCAD'24]

Program P	Function f	Property
Identity	$f(x) = x$	$P(x) = P(x + \tilde{r}) - P(\tilde{r})$
Integer Mult.	$f(x, y) = x \cdot y$	$P(x, y) = P(\tilde{x}_1, \tilde{y}_1) + P(\tilde{x}_1, \tilde{y}_2) + P(\tilde{x}_2, \tilde{y}_1) + P(\tilde{x}_2, \tilde{y}_2)$
Integer Mult.	$f(x, y) = x \cdot y$	$P(x, y) = P(x + \tilde{r}, y + \tilde{s}) - P(\tilde{r}, y + \tilde{s}) - P(x + \tilde{t}, \tilde{s}) + P(\tilde{t}, \tilde{s})$
Integer Div.	$f(x, R) = x \div R$	$P(x, y) = P(x_1, R) + P(x_2, R) + P\left(P_{\text{mod}}(x_1, R) + P_{\text{mod}}(x_2, R), R\right)$
Mod	$f(x, R) = x \bmod R$	$P(x, R) = P(\tilde{x}_1, R) +_R P(\tilde{x}_2, R)$
Mod Mult.	$f(x, y, R) = x \cdot_R y$	$P(x, y, R) = P(\tilde{x}_1, \tilde{y}_1, R) +_R P(\tilde{x}_2, \tilde{y}_1, R) +_R P(\tilde{x}_1, \tilde{y}_2, R) +_R P(\tilde{x}_2, \tilde{y}_2, R)$
Mod Exp.,	$f(a, x, R) = a^x \bmod R$	$P(a, x, R) = P(a, \tilde{x}_1, R) \cdot_R P(a, \tilde{x}_2, R)$
Matrix Inv.	$f(A) = A$	$P(A) = R \times P(A \times R)$ where A and R are invertible n-by-n matrices.
Determinant	$f(A) = \det A$	$P(A) = P(\tilde{R}) / P(A \times \tilde{R})$ where $\tilde{R}$ is invertible.

Erata, F., Chiu, T., Etim, A., Nampally, S., Raju, T., Ramu, R., Piskac, R., Antonopoulos, T., Xiong, W. and Szefer, J.
Systematic Use of Random Self-Reducibility against Physical Attacks.
2024 ACM/IEEE International Conference on Computer-Aided Design (ICCAD '24).
https://doi.org/10.1145/3676536.3689920

Overview of the Countermeasure

against power side-channels (*randomization*) & fault injections (*redundancy*)

$$f(x+y) = f(x) + f(y)$$

Overview of the Countermeasure

against power side-channels (*randomization*) & fault injections (*redundancy*)

$$f(x+y) = f(x) + f(y)$$

Random Self-Reducible Properties

[Blum, Luby, and Rubinfeld (BLR). 1993]

Blum - Turing award 1995

Let $x \in \mathbb{D}$ and $c > 1$ be an integer. We say that f is **c-random self-reducible** if f can be computed at any particular input x via:

$$f(x) = F(x, u_1, \ldots, u_c, f(u_1), \ldots, f(u_c))$$

where F can be computed asymptotically faster than f and the u_i's are uniformly distributed, although not necessarily independent; e.g., given the value of u_1 it is not necessary that u_2 be randomly distributed in $\mathbb{D}$

RSR against Power Side Channel Attacks

c-secure-countermeasure PSCA

Algorithm 1: c-secure-countermeasure PSCA (P, x, c).

Input : Program: P, Sensitive input: x, Security: c
Output: $P(x)$

1 Randomly split $a_1, \ldots, a_c$ based on x.
2 **for** $i = 1, \ldots, c$ **do**
3 $\quad \alpha_i \leftarrow P(a_i)$
4 **return** $F[x, a_1, \ldots, a_c, \alpha_1, \ldots, \alpha_c]$

Masking with Random Self-Reducibility

If a cryptographic operation has a *random self-reducible property*, then it is possible to protect it against power side-channel attacks by masking with arithmetic secret sharing.

RSR against Power Side Channel Attacks

c-secure-countermeasure PSCA

Algorithm 1: c-secure-countermeasure PSCA (P, x, c).

Input : Program: P, Sensitive input: x, Security: c
Output: $P(x)$

Randomly split $a_1, \ldots, a_c$ based on x.
for $i = 1, \ldots, c$ **do**
  $\alpha_i \leftarrow P(a_i)$
return $F[x, a_1, \ldots, a_c, \alpha_1, \ldots, \alpha_c]$

Black-box

If we replace the f function with a program P that computes the function f, then our countermeasure $\widetilde{C}$ access P as a black-box and computes the function f using the random self-reducible properties of f.

Self-Correctness against Fault Injections Attacks

n-secure countermeasure FIA

Algorithm 2: n-secure countermeasure FIA (P, x, n, c).

Input : Program: P, Sensitive input: x, Security: n, c
Output: $P(x)$

for $m = 1, \ldots, n$ **do**
  answer$_m$ $\leftarrow$ call c-secure-countermeasure(P, x, c)
return the majority in $\{\text{answer}_m \colon m = 1, \ldots, n\}$

Self-Correctness with Majority Voting

Fault injection attacks rely on faulty output. By majority voting, we can obtain correct results even if some results are incorrect.

Example

(c, n)-secure mod operation (P, R, x, c, n)

Algorithm 3: $(c,\ n)$-secure mod operation (P, R, x, c, n).

Input : Program: P, Sensitive input: x, Security: n, c
Output: $P(x)$

```
1 for m = 1, ..., n do
2     x_1, x_2, ..., x_c ←$ Random-Split(R2^n, x)
3     answer_m ← P(x_1, R) +_R P(x_2, R) ... +_R P(x_c, R)
4 return the majority in {answer_m: m = 1, ..., n}
```

This algorithm presents an example of a **combined** and **configurable** countermeasure, effective against both PSCA and FIA.

In Line 2, the algorithm divides the input x into c shares $x_1, x_2, \ldots, x_c$, satisfying

$$x = x_1 + x_2 + \cdots + x_c.$$

n ***and attacker's probability of success***

ε-fault tolerance

Let ε be the upper bound on the attacker's probability of injecting a fault successfully at an unprotected program P that correctly implements a function f.

Say that the program P is ε-fault tolerant for the function f provided $P(x) = f(x)$ for at least $1 - \varepsilon$ of any input x, which is $\Pr_{fault}[P(x) \neq f(x)] < \varepsilon$.

Lower bound for $\mathbf{n}$. The attacker's probability of success is ε, and for a c-secure countermeasure, the lower bound for n is defined as:
$n = \log(1/\delta)2(1 - \varepsilon c)/(\varepsilon c/2)^2$, where δ is the confidence parameter.

Random Split Function

Algorithm 4: Random-Split(m, x, c).

Input: modulus: m, input value: x, # of shares: c

Output: an array of shares $a_1, a_2, \ldots, a_c$.

```
Initialize an array s[1 ... c] and initialize sum ← 0
i ← 1
for i to c − 1 do
    s[i] ←$ random integer in Z_m
    sum ← sum + s[i]
s[c] ← x − sum (mod m)
return s
```

Majority Vote Algorithm

Boyer-Moore's algorithm

Algorithm 5: Majority Vote Algorithm

Input : A list of elements $a_1, a_2, \ldots, a_n$
Output: The majority element of the list

Initialize an element m and a counter i with $i = 0$;

for $j \leftarrow 1$ *to* n **do**
  if $i = 0$ **then** $m \leftarrow a_j$ and $i \leftarrow 1$
  else if $m = a_j$ **then** $i \leftarrow i + 1$
  else $i \leftarrow i - 1$

return m

Protected ***Majority Vote Algorithm***

Boyer-Moore's algorithm with Fisher-Yates shuffle

Algorithm 6: Protected Majority Vote $(\ell, \mathrm{majority}, n)$

Input : Votes $\ell = a_1, a_2, \ldots, a_n$, function majority, and n
Output: The majority element of the list, if it exists

```
m ← 1
for m to n do
    ℓ1 ←$ shuffle(ℓ)
    answer_m ← majority(ℓ1)
if m ≠ n then output "FAIL" and halt          ▷ verify loop
  completion
return the majority in {answer_m: m = 1, ..., n}
```

Fisher-Yates shuffle

Algorithm 7: Fisher-Yates Shuffle

Input : A list of elements $a_1, a_2, \ldots, a_n$

Output: A random permutation of the elements in the input list

for $i \leftarrow n - 1$ *down to* 1 **do**

  Choose a random integer j such that $0 \leq j \leq i$

  Swap a_i and a_j

return the shuffled list

Randomized Self-Reductions [BLR 1993]

Program P	**Function** f	**Random Self-Reducible Property**
Mod Operation	$f(x,R) = x \bmod R$	$P(x,R) \leftarrow P(\tilde{x}_1,R) +_R P(\tilde{x}_2,R)$
Modular Multiplication	$f(x,y,R) = x \cdot_R y$	$P(x,y,R) \leftarrow P(\tilde{x}_1,\tilde{y}_1,R) +_R P(\tilde{x}_2,\tilde{y}_1,R) +_R P(\tilde{x}_1,\tilde{y}_2,R) +_R P(\tilde{x}_2,\tilde{y}_2,R)$
Modular Exponentiation	$f(a,x,R) = a^x \bmod R$	$P(a,x,R) \leftarrow P(a,\tilde{x}_1,R) \cdot_R P(a,\tilde{x}_2,R)$
Modular Inverse	$f(x,R) \cdot_R x = 1$	$P(x,R) \leftarrow \tilde{w} \cdot_R P(x \cdot_R \tilde{w})$ where $P(\tilde{w},R) \cdot_R \tilde{w} = 1$ and $P(x,R) \cdot_R x = 1$
Polynomial Multiplication	$f(p_x,q_x) = p_x \cdot q_x$	$P(p,q) \leftarrow P(\tilde{p}_1,\tilde{q}_1) + P(\tilde{p}_2,\tilde{q}_1) + P(\tilde{p}_1,\tilde{q}_2) + P(\tilde{p}_2,\tilde{q}_2)$
Number Theoretic Transform	$f(x_1,\ldots,x_n) = \cdots$	$P(x_1,\ldots,x_n) \leftarrow P(x_1+\tilde{r}_1,\ldots,x_n+\tilde{r}_n) - P(\tilde{r}_1,\ldots,\tilde{r}_n)$
Integer Multiplication	$f(x,y) = x \cdot y$	$P(x,y) \leftarrow P(\tilde{x}_1,\tilde{y}_1) + P(\tilde{x}_1,\tilde{y}_2) + P(\tilde{x}_2,\tilde{y}_1) + P(\tilde{x}_2,\tilde{y}_2)$
Integer Multiplication	$f(x,y) = x \cdot y$	$P(x,y) \leftarrow P(x+\tilde{r},y+\tilde{s}) - P(\tilde{r},y+\tilde{s}) - P(x+\tilde{t},\tilde{s}) + P(\tilde{t},\tilde{s})$
Integer Division	$f(x,R) = x \div R$	$P(x,y) \leftarrow P(x_1,R) + P(x_2,R) + P\left(P_{\text{mod}}(x_1,R) + P_{\text{mod}}(x_2,R),R\right)$
Matrix Multiplication	$f(A,B) = A \times B$	$P(A,B) \leftarrow P(\tilde{A}_1,\tilde{B}_1) + P(\tilde{A}_2,\tilde{B}_1) + P(\tilde{A}_1,\tilde{B}_2) + P(\tilde{A}_2,\tilde{B}_2)$
Matrix Inverse	$f(A) = A^{-1}$	$P(A) \leftarrow \tilde{R} \times P(A \times \tilde{R})$ where A and $\tilde{R}$ are invertible n-by-n matrices.
Matrix Determinant	$f(A) = \det A$	$P(A) \leftarrow P(\tilde{R})/P(A \times \tilde{R})$ where $\tilde{R}$ is invertible.

Protected ***Mod Operation***

2-secure protected mod operation

Algorithm 8: 2-secure protected mod operation (P, R, x)

---

1 $x_1, x_2 \xleftarrow{\$}$ Random-Split$(R2^n, x)$
2 **return** $P(x_1, R) +_R P(x_2, R)$

Protected **Mod Operation**

3-secure protected mod operation

Algorithm 9: 3-secure protected mod operation (P, R, x)

1. $x_1, x_2, x_3 \xleftarrow{\$} \text{Random-Split}(R2^n, x)$
2. **return** $P(x_1, R) +_R P(x_2, R) +_R P(x_3, R)$

Protected **Mod Multiplication** *and* **Exponentiation**

Algorithm 10: 2-secure mod. multiplication (P, R, x, y)

$x_1, x_2 \xleftarrow{\$}$ Random-Split$(R \times 2^n, x)$
$y_1, y_2 \xleftarrow{\$}$ Random-Split$(R \times 2^n, y)$
return $P(x_1, y_1, R) +_R P(x_2, y_1, R) +_R P(x_1, y_2, R) + P(x_2, y_2, R)$

Algorithm 11: 2-secure mod. exponentiation (P, R, a, x)

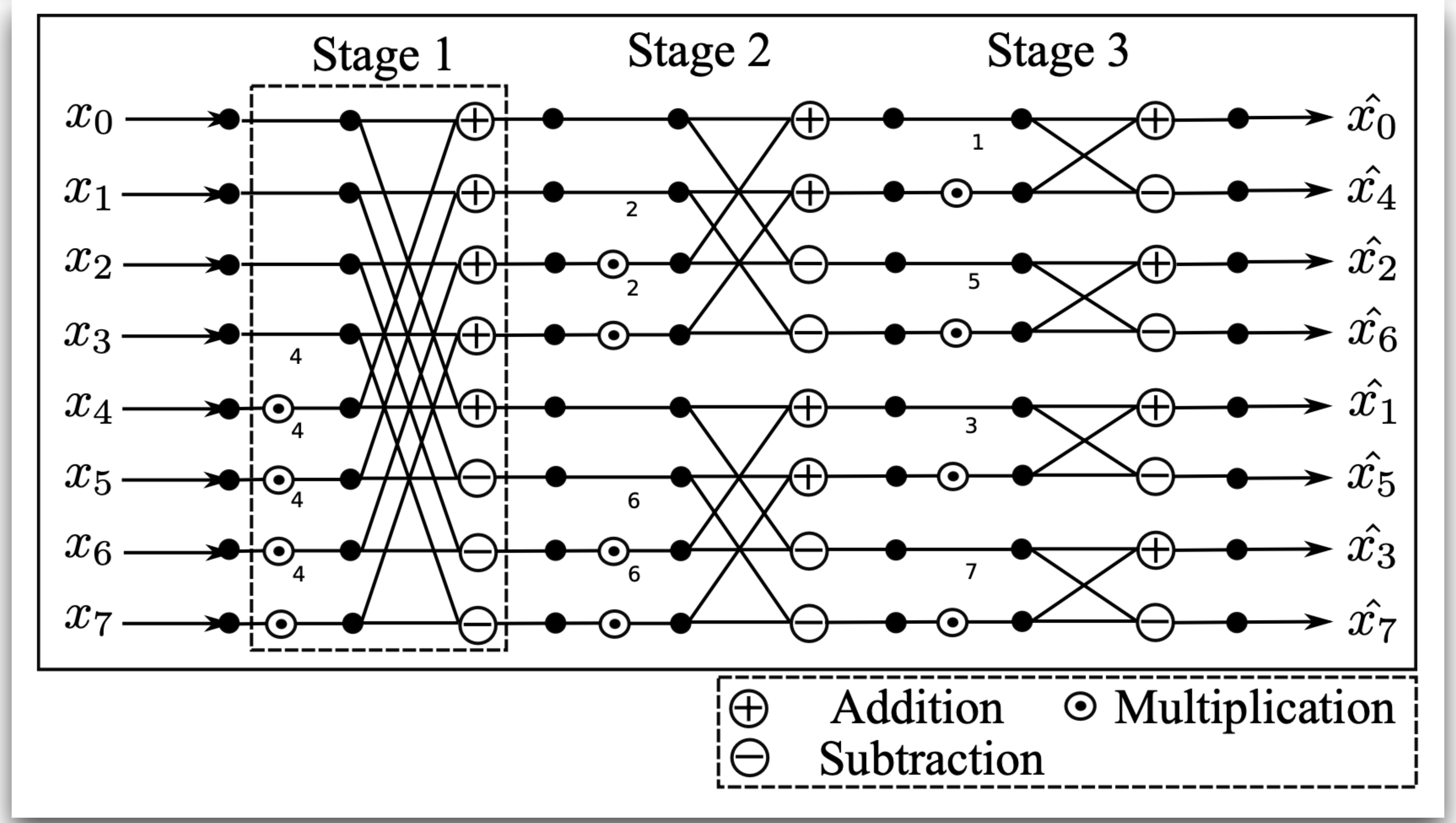
$x_1, x_2 \xleftarrow{\$}$ Random-Split$(\phi(R)2^n, x)$
return $\leftarrow P(a, x_1, R) \cdot_R P(a, x_2, R)$ ▷ calls Algo. 10

Number Theoretic Transforms (NTT)

Algorithm 13: 2-secure NTT $(P, x_1, \ldots, x_n \in \mathbb{Z}_q^2)$.

Choose $\tilde{r}_1, \ldots, \tilde{r}_n \in_{\mathbb{U}} \mathbb{Z}_q^2$

return $\mathrm{NTT}\left(x_1 + \tilde{r}_1, \ldots, x_n + \tilde{r}_n\right) - \mathrm{NTT}\left(\tilde{r}_1, \ldots, \tilde{r}_n\right)$



End-to-End Implementations

RSA-CRT Signature Generation Algorithm

Algorithm 14: RSA-CRT Signature Generation Algorithm

Input: A message M to sign, the private key (p, q, d), with $p > q$, pre-calculated values $d_p = d \mod (p-1)$, $d_q = d \mod (q-1)$, and $u = q^{-1} \mod p$.

Output: A valid signature S for the message M.

$m \leftarrow$ Encode the message M in $m \in \mathbb{Z}_N$

$s_p \leftarrow m^{d_p} \mod p$ ▷ Protection with Algorithm 11

$s_q \leftarrow m^{d_q} \mod q$ ▷ Protection with Algorithm 11

$t \leftarrow s_p - s_q$

if $t < 0$ **then**

  $t \leftarrow t + p$

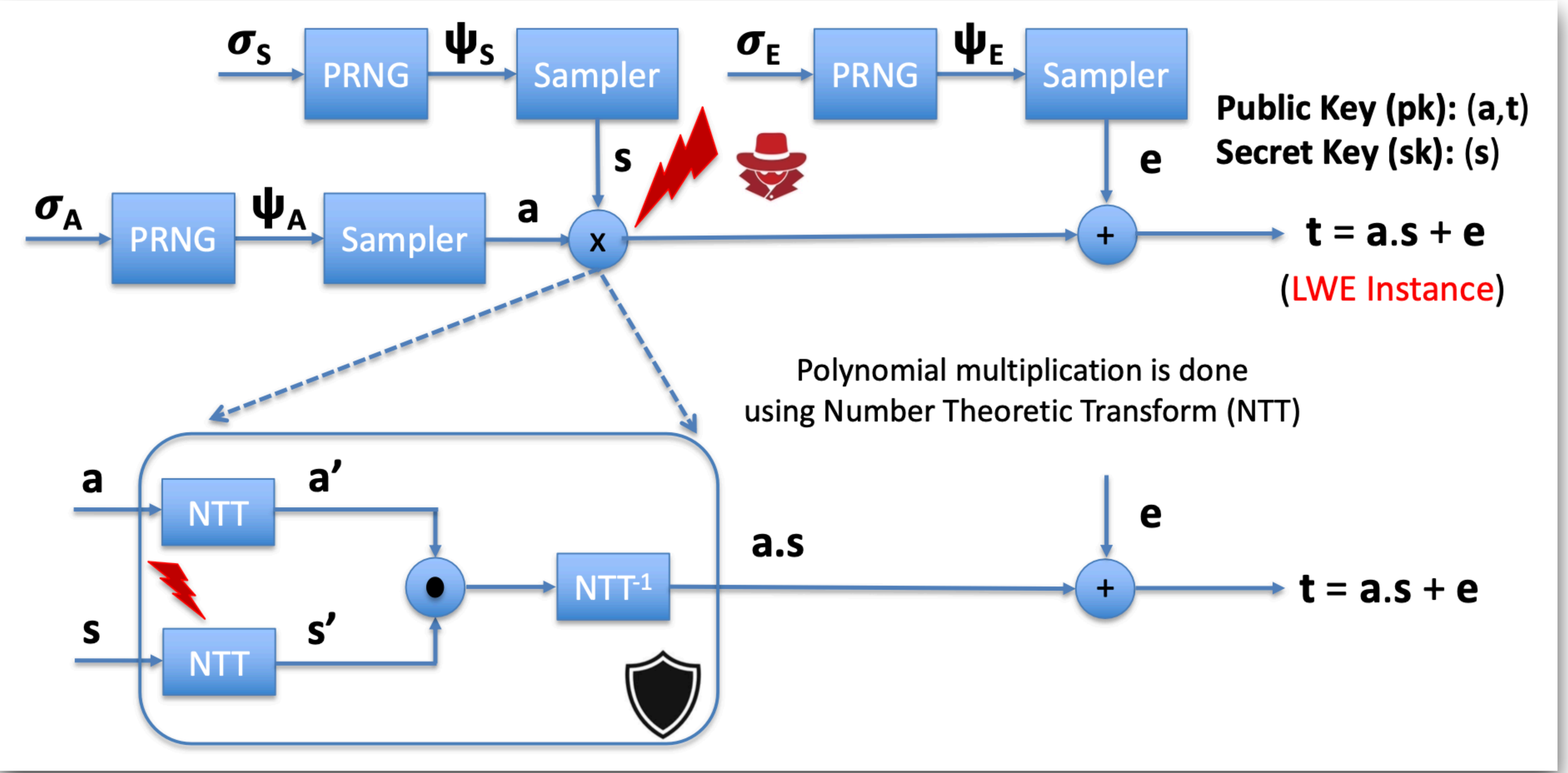
$S \leftarrow s_q + ((t \cdot u) \mod p) \cdot q$

return S *as a signature for the message* M

End-to-End Implementations

CPA Secure Kyber PKE

Existing Attack



Prasanna Ravi, Bolin Yang, Shivam Bhasin, Fan Zhang, and Anupam Chattopadhyay. *Fiddling the twiddle constants-fault injection analysis of the number theoretic transform*. IACR Transactions on Cryptographic Hardware and Embedded Systems, pages 447–481, 2023

End-to-End Implementations

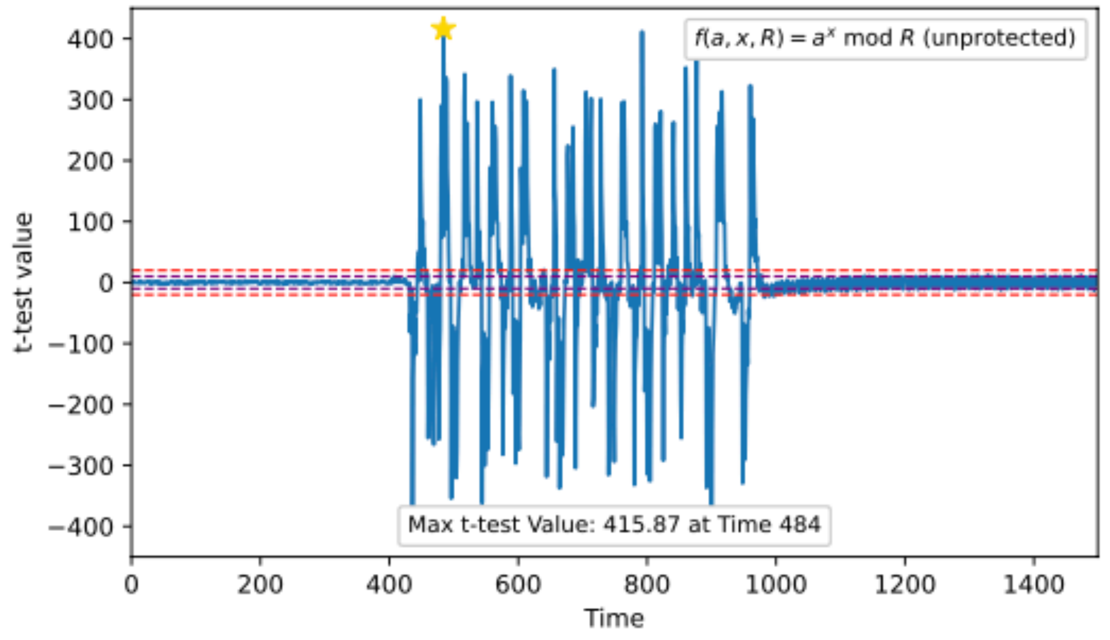
CPA Secure Kyber PKE

Algorithm 15: CPA Secure Kyber PKE (CPA.KeyGen)

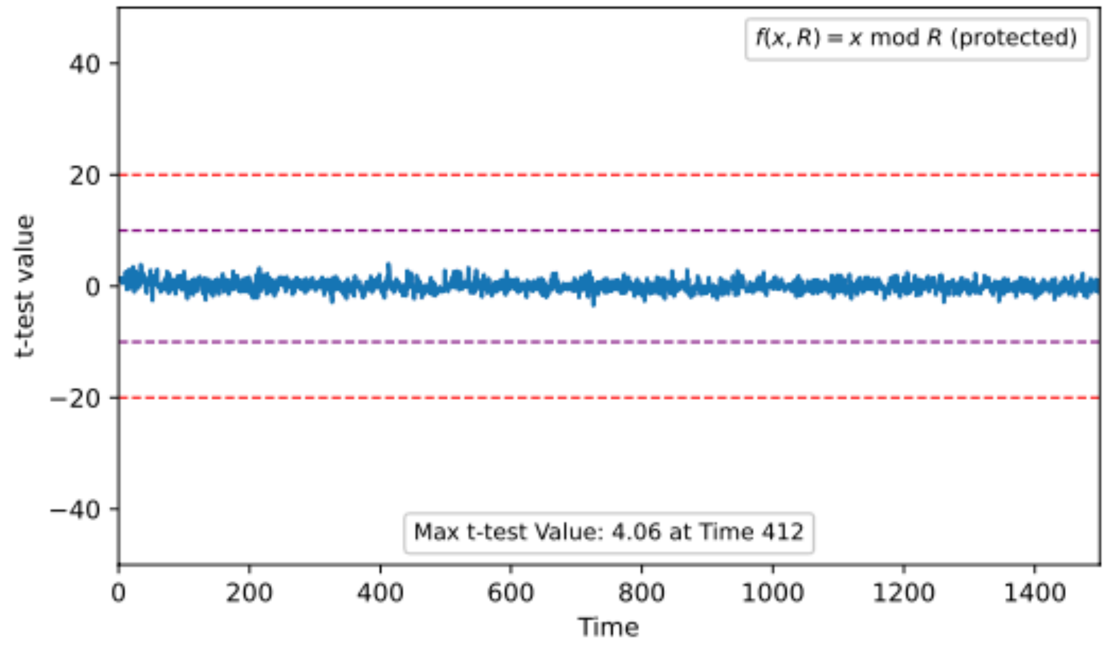
$seed_A \leftarrow \text{Sample}_U()$
$seed_B \leftarrow \text{Sample}_U()$
$\hat{A} \leftarrow \text{NTT}(A)$
$s \leftarrow \text{Sample}_B(seed_B, coins_s)$
$e \leftarrow \text{Sample}_B(seed_B, coins_e)$
$\hat{s} \leftarrow \text{NTT}(s)$ $\triangleright$ Protection with Algorithm 13
$\hat{e} \leftarrow \text{NTT}(e)$
$\hat{t} \leftarrow \hat{A} \odot \hat{s} + \hat{e}$
return $pk = (seed_A, \hat{t}), sk = (\hat{s})$

Evaluation

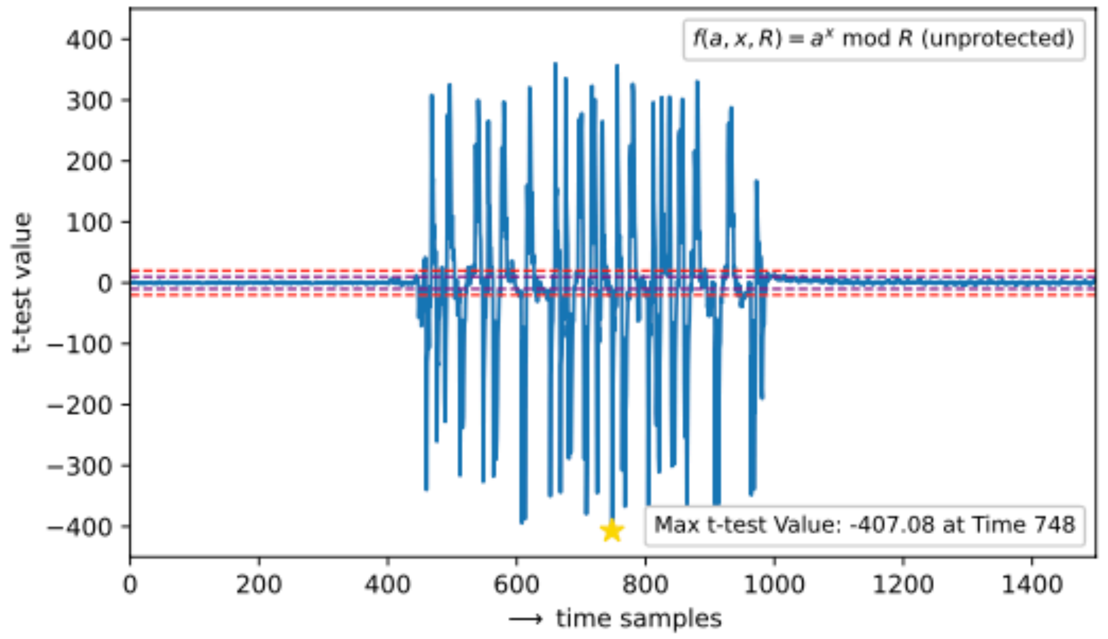
Power Side-Channel Attack Evaluation t-tests (TVLA)



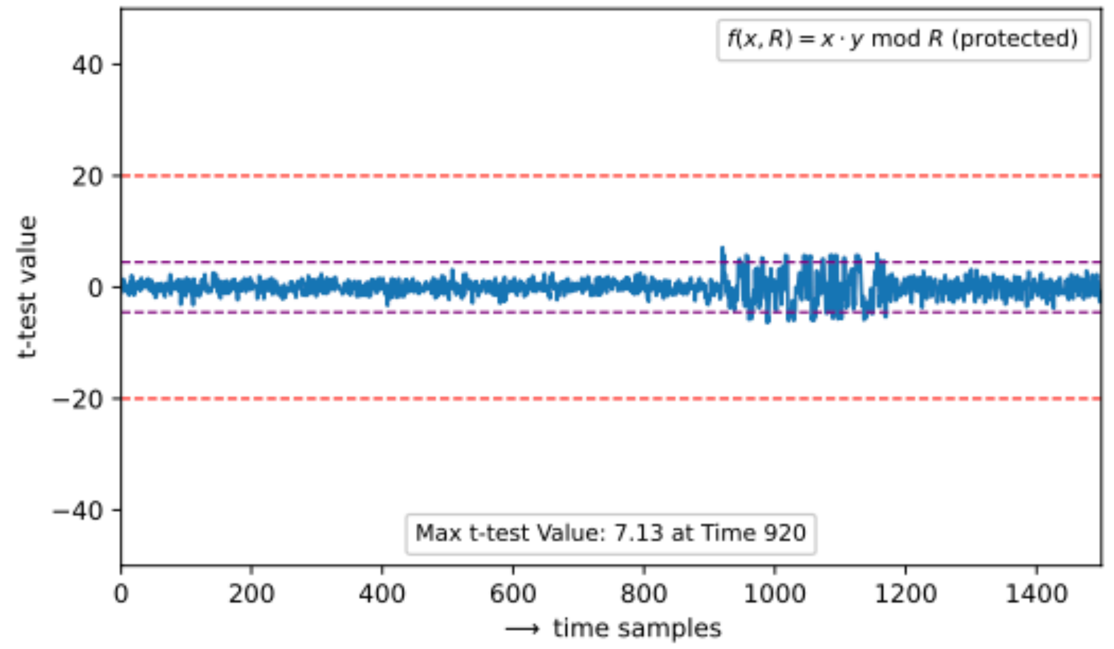
(a) Unprotected Mod Operation



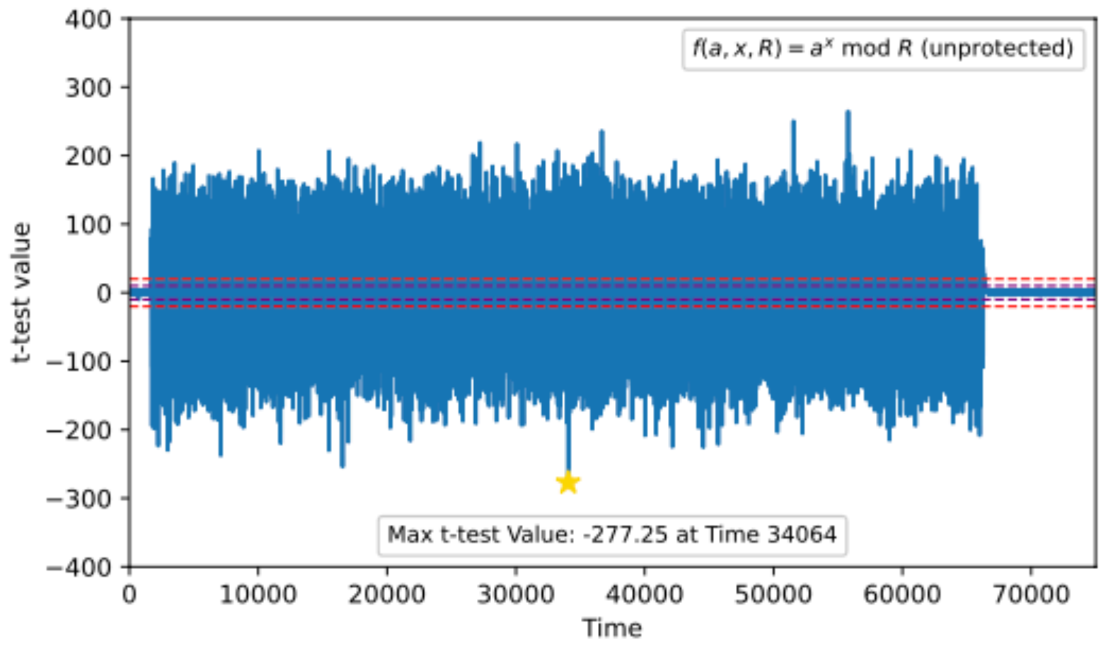
(b) Protected Mod Operation



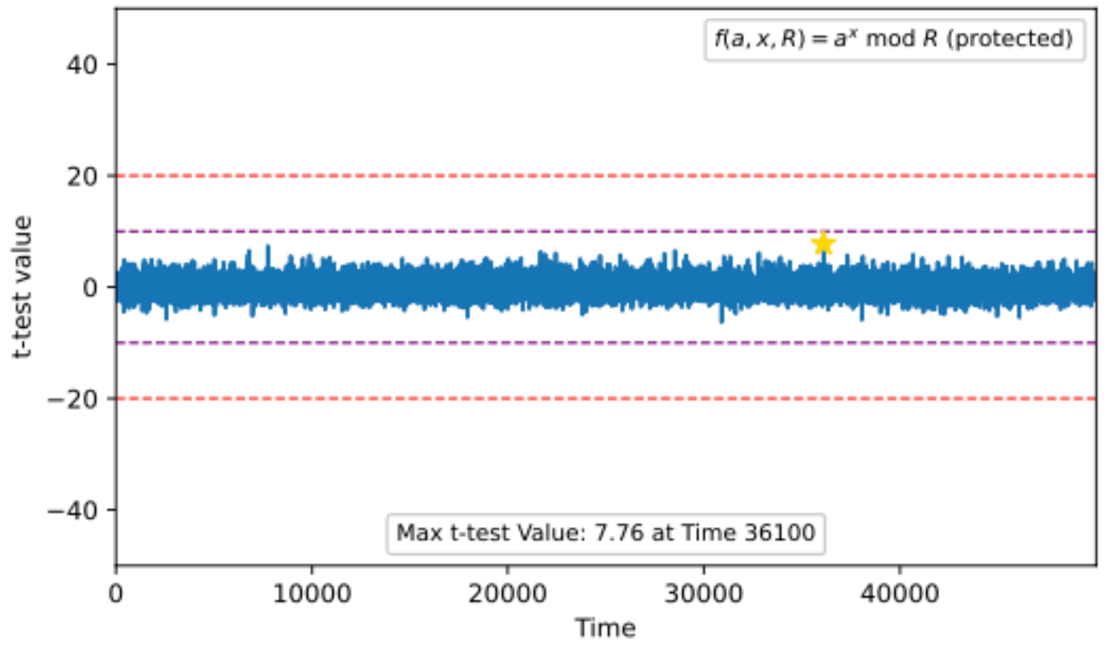
(c) Unprotected Mod. Mult.



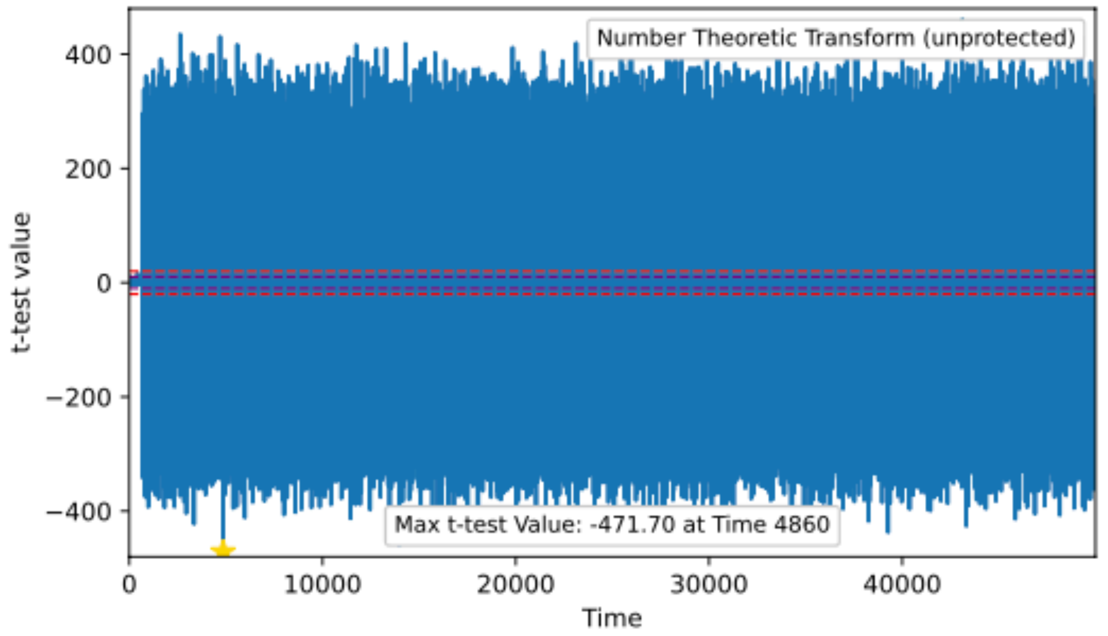
(d) Protected Mod. Mult.



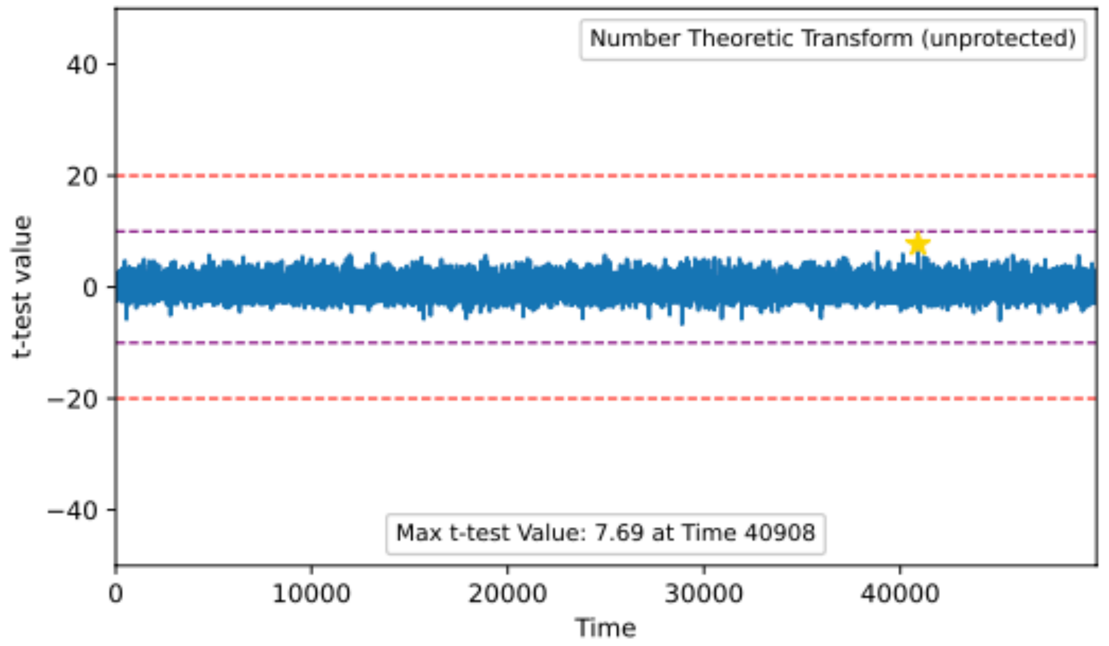
(e) Unprotected Mod. Exp.



(f) Protected Mod. Exp.



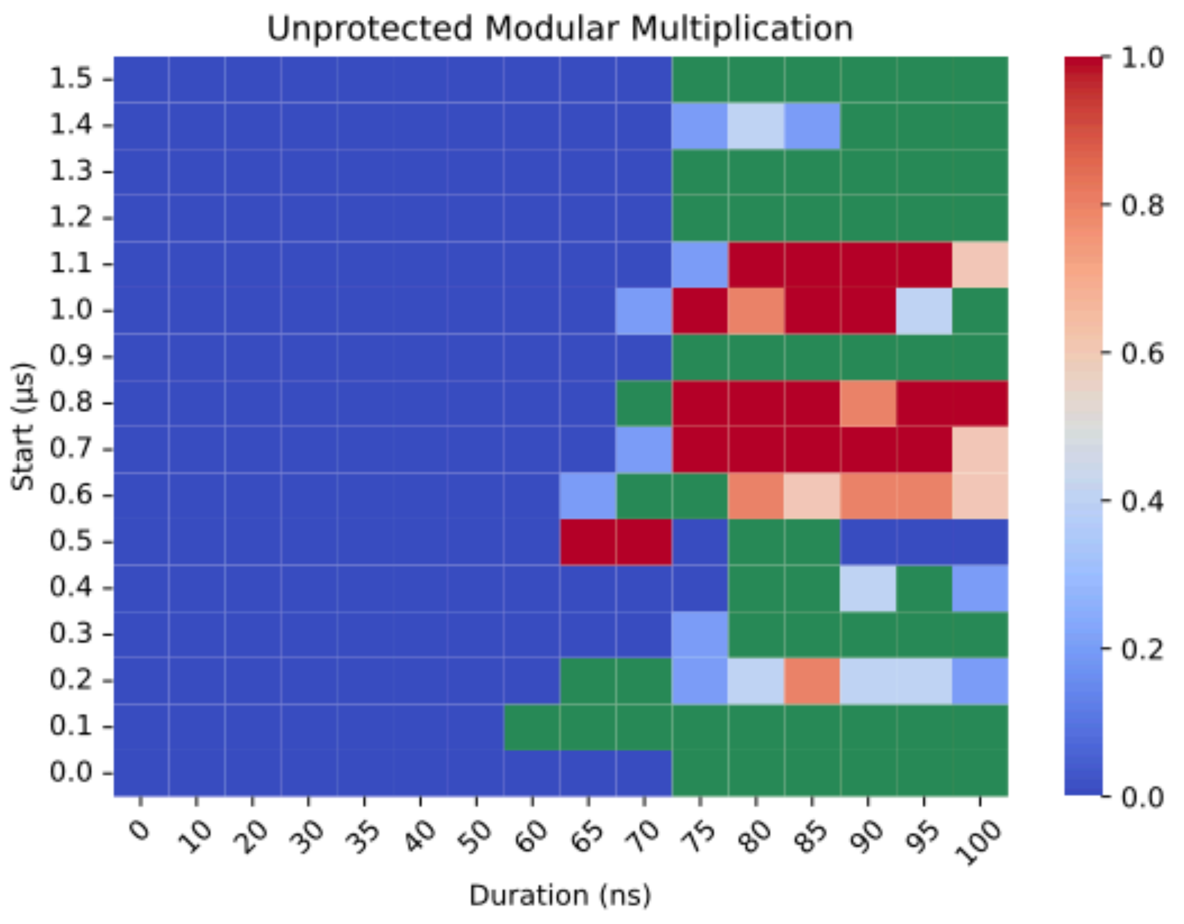
(g) Unprotected NTT



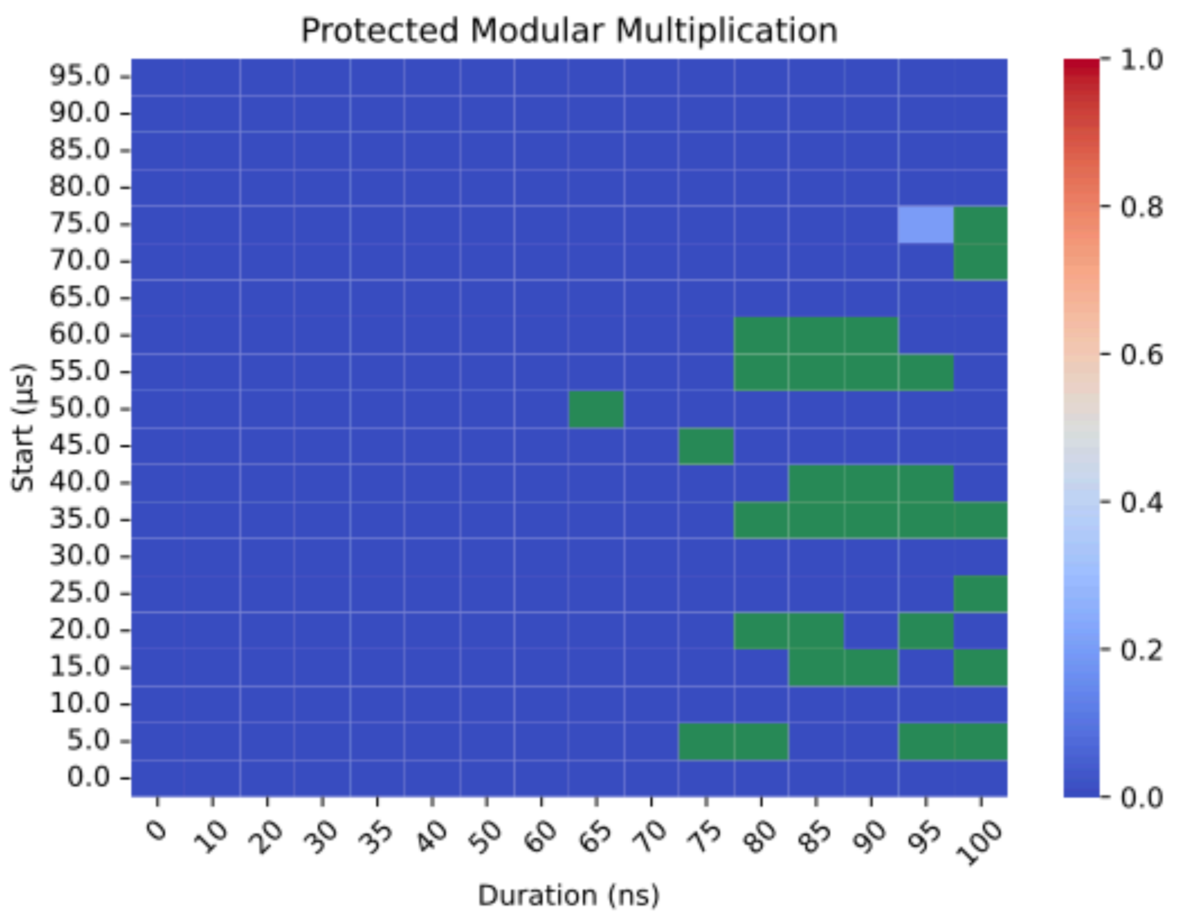
(h) Protected NTT

Evaluation

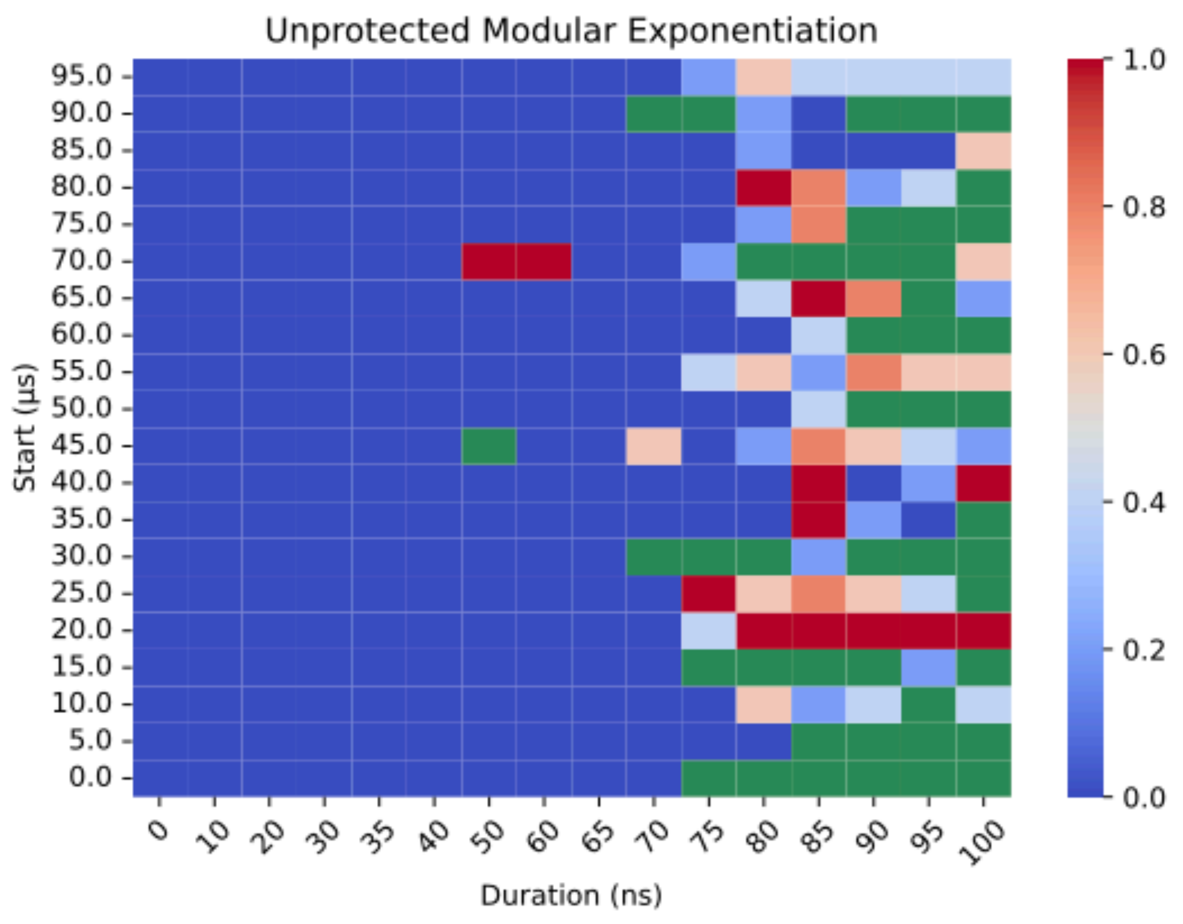
Fault Injection Attack Evaluation Heatmaps



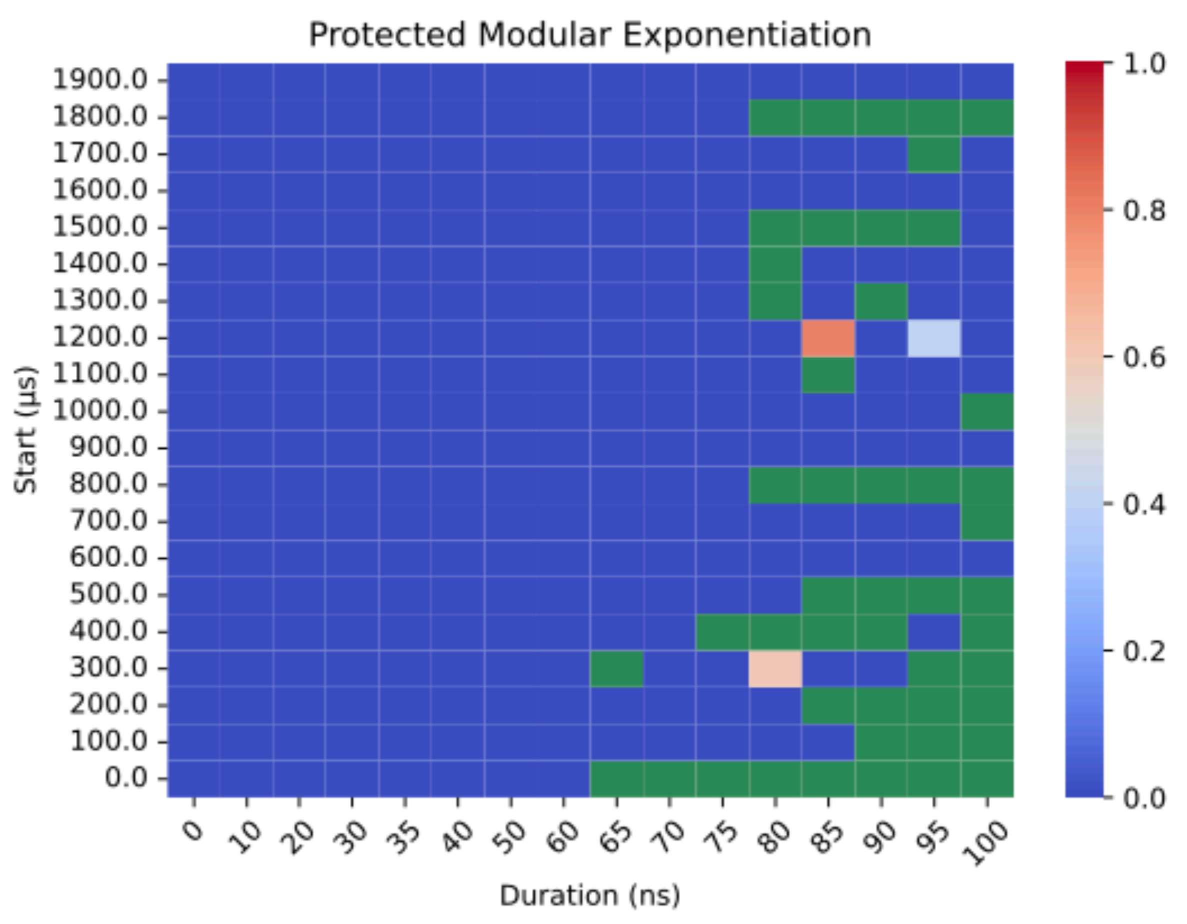
(a) Unprotected Mod. Mult.



(b) Protected Mod. Mult.



(c) Unprotected Mod. Exp.

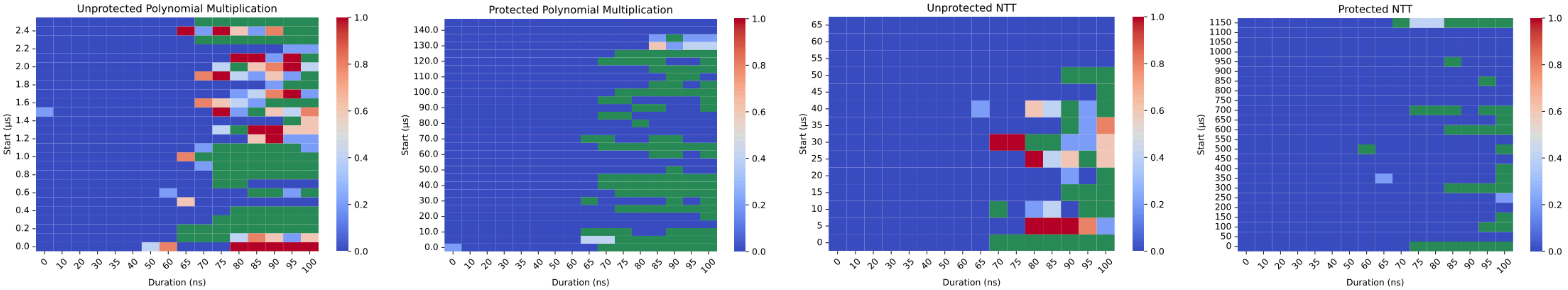


(d) Protected Mod. Exp.

We used $(2,10)$-secure countermeasure in the experiments.

Evaluation

Fault Injection Attack Evaluation Heatmaps

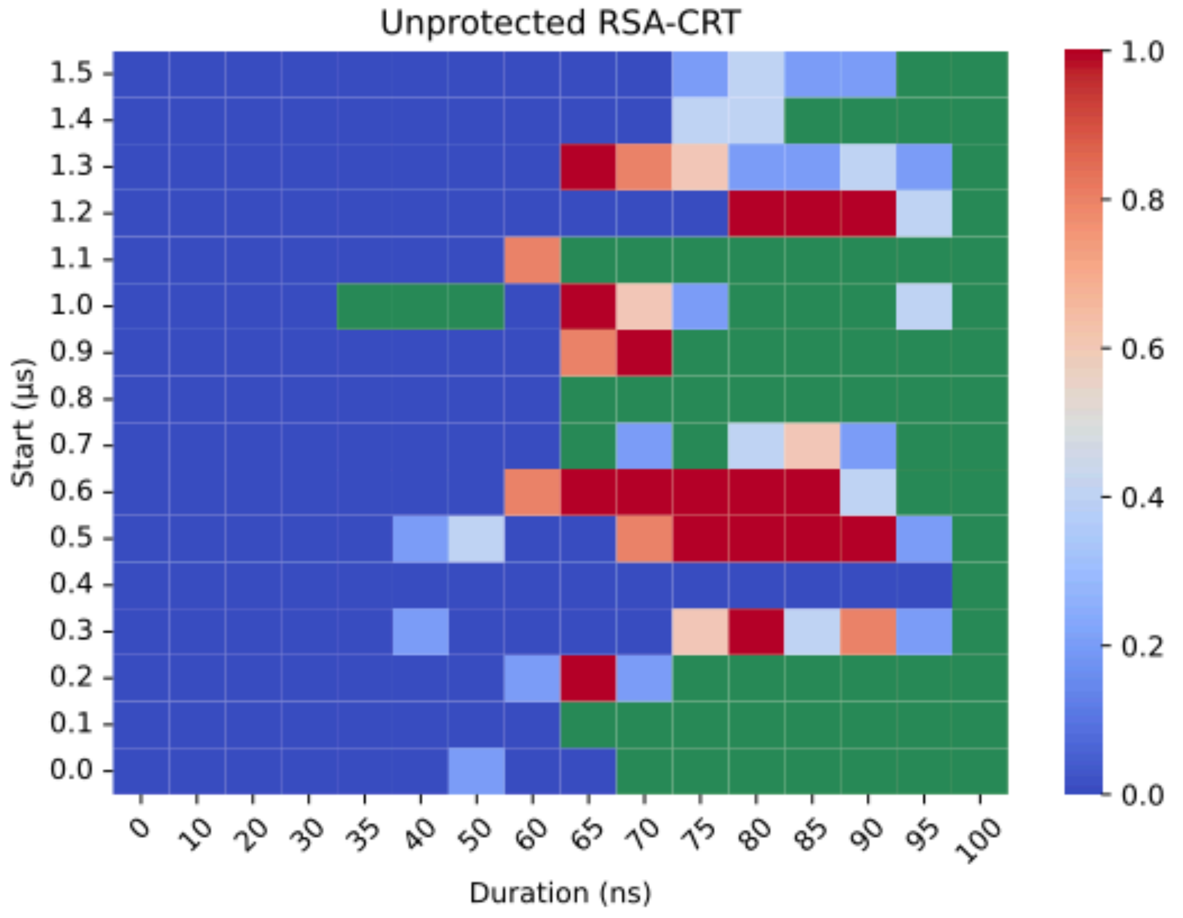


(e) Unprotected Poly. Mult. (f) Protected Poly. Mult. (g) Unprotected NTT (h) Protected NTT

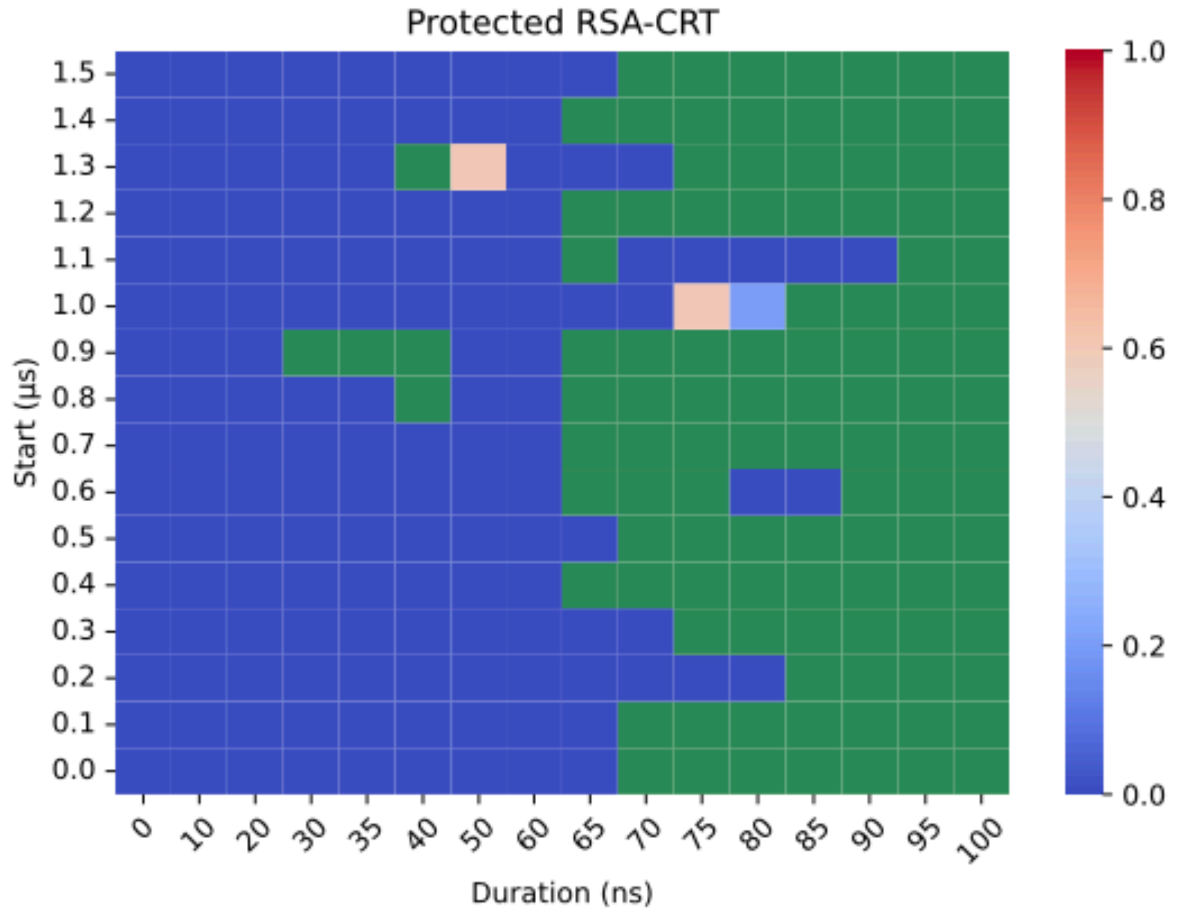
We used $(2,10)$-secure countermeasure in the experiments.

Evaluation

Fault Injection Attack Evaluation Heatmaps

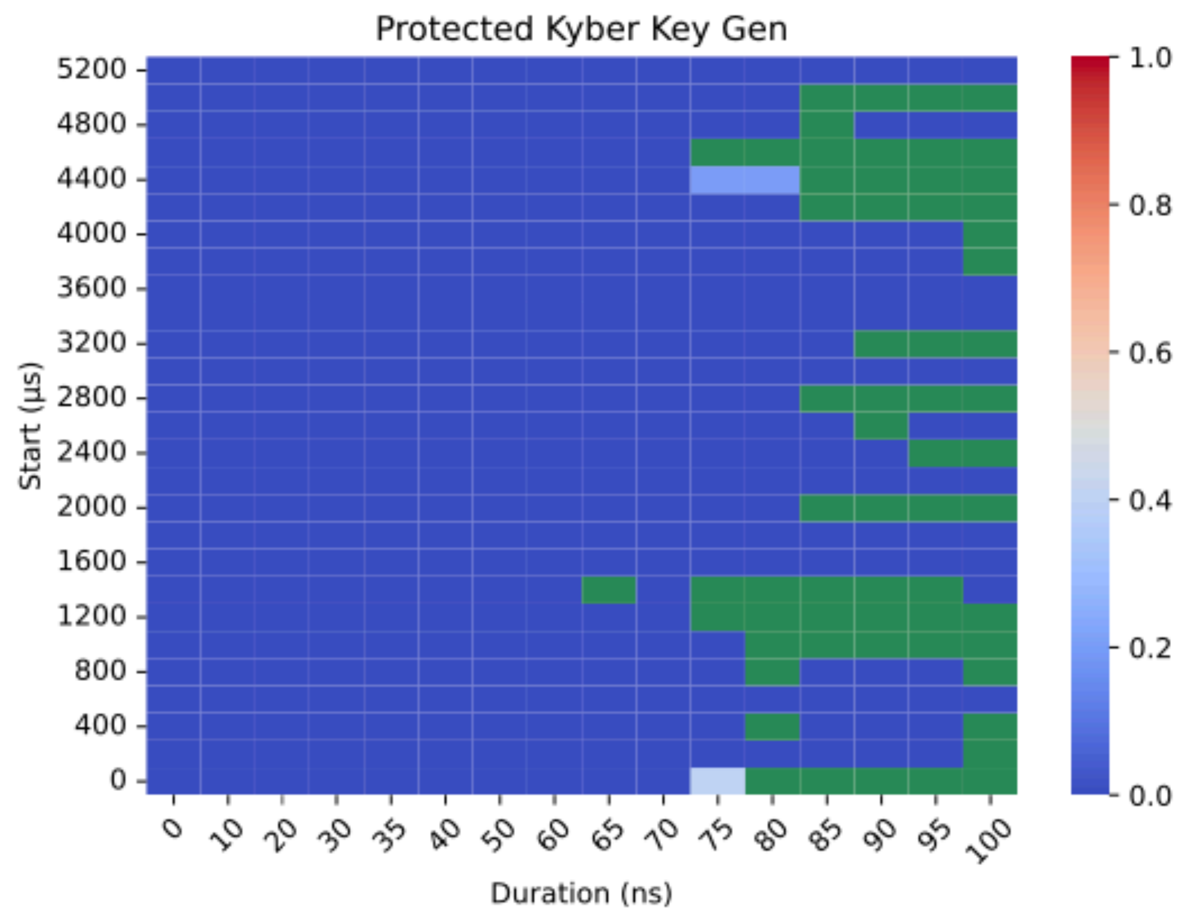


(i) Unprotected RSA-CRT



(j) Protected RSA-CRT

(k) Unprotected Kyber Key Gen.



(l) Protected Kyber Key Gen.

We used $(2,10)$-secure countermeasure in the experiments.

Evaluation

Reduction in Faults for Different Operations

Operation	Unprotected	Protected	Reduction
Mod. exponentiation	165	9	94.55%
Mod. multiplication	168	1	99.4%
NTT	63	5	92.06%
Poly. multiplication	196	14	92.86%
RSA-CRT	168	7	95.83%
Kyber Key. Gen.	172	4	97.67%

Limitations

The countermeasure's effectiveness is intrinsically linked to the random self-reducibility of the function being protected. This dependency means that our approach may not be universally applicable to all cryptographic operations.

Redundancy and randomness inevitably introduce computational overhead. Nevertheless, each call to original function P can be easily parallelized in hardware or vectorized software implementations.

Our approach is not tailored to defend against attacks targeting the random number generator itself.

Future Work

Compare it to Masked Implementations from Power Side-Channel Perspective such as complex NTT circuits.

Vectorized or Hardware support to cope with extra latency

Erata, F., Chiu, T., Etim, A., Nampally, S., Raju, T., Ramu, R., Piskac, R., Antonopoulos, T., Xiong, W. and Szefer, J.
Systematic Use of Random Self-Reducibility against Physical Attacks.
2024 ACM/IEEE International Conference on Computer-Aided Design (ICCAD '24).
https://doi.org/10.1145/3676536.3689920