Compiler Assisted Masking

A. Moss, E. Oswald, D. Page and M. Tunstall

School of Computing, Blekinge Institute of Technology,
Karlskrona, Sweden.
andrew.moss@bth.se

Department of Computer Science, University of Bristol,
Merchant Venturers Building, Woodland Road,
Bristol BS8 1UB, United Kingdom.
{eoswald,page,tunstall}@cs.bris.ac.uk

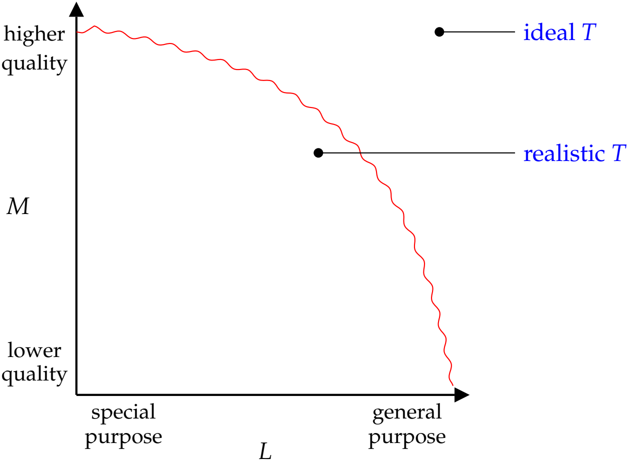
10/09/12

Context

- A **program transformation** T
  1. takes a source program S, then
  2. produces a target program $S' = T(S)$ st.
     - function is preserved, i.e., for all input x, $S'(x) \equiv S(x)$,
     - quality is improved, i.e., for some metric M, $M(S') > M(S)$.
- Clearly one can repeat as necessary: a **compiler** C for some source language L basically just means for $S \in L$,

$$C(S) = (T_{n-1} \circ \cdots \circ T_1 \circ T_0)(S).$$

Context



higher
quality
M
lower
quality
special
purpose
L
general
purpose
ideal T
realistic T

Context

- The premise
  1. secure software development is challenging, so
  2. automatic program transformation is of value since it reduces barriers which prevent use, plus workload and error

  motivated FP7 CACE project.
- Goal: security-conscious selections of T and hence M within some C ... or (much) more specifically

$$T \simeq \text{apply } a \text{ Boolean } \textbf{masking scheme}$$

$$M \simeq \textit{improve} \text{ resilience against } \textbf{DPA attack}\text{, while } \textit{reducing} \text{ associated user intervention}$$

Approach

- Option #1: **experimental** (cf. Bayrak et al. [1]).
- Option #2: **formal**, using a style of **information flow** by
  1. adding type annotation to support feed-forward type inference,
  2. appling recovery rules to cope with type errors, then
  3. generating target program for ARM, inc. support code where necessary

  with the underlying aim to *model* what a human programmer would do.

Approach

- The **type annotation** process marks each base type as
  1. **low-security**, which is the default, or
  2. **high-security** annotation, including a **mask set**,

  where masks can be
  1. **concrete**, say m, or
  2. **wildcard**, say m^*, allowing unification.
- Example:

Syntax: `byte x : { L }` $\equiv$ `byte x`

$$\downarrow$$

Type: $\mathcal{E} \vdash x : \mathbb{Z}_{256}{}^{L} \equiv \mathbb{Z}_{256}{}^{H:\emptyset}$

$$\uplus$$

Value: x

Approach

- The **type annotation** process marks each base type as
  1. **low-security**, which is the default, or
  2. **high-security** annotation, including a **mask set**,

  where masks can be
  1. **concrete**, say m, or
  2. **wildcard**, say m^*, allowing unification.
- Example:

Syntax: `byte x : { H< m0 > }`

$$\downarrow$$

Type: $\mathcal{E} \vdash x : \mathbb{Z}_{256}^{H:\langle m_0 \rangle}$

$$\Cup$$

Value: $x \oplus m_0$

Approach

- The **type annotation** process marks each base type as
  1. **low-security**, which is the default, or
  2. **high-security** annotation, including a **mask set**,

  where masks can be
  1. **concrete**, say m, or
  2. **wildcard**, say m^*, allowing unification.
- Example:

Syntax:

```
byte x : { H< m0, m1 > }
```

$$\downarrow$$

Type: $\mathcal{E} \vdash x : \mathbb{Z}_{256}^{H:\langle m_0, m_1 \rangle}$

$$\Downarrow$$

Value: $x \oplus m_0 \oplus m_1$

Approach

- The **type inference** processes propagates this annotation:
  - Case #1: low-security

$$\frac{\exists \oplus \in \mathcal{E} \quad \mathcal{E} \vdash \oplus : (\rightarrow T_r, T_0, T_1) \quad \mathcal{E} \vdash E_0 : {T_0}^{L} \quad \mathcal{E} \vdash E_1 : {T_1}^{L}}{\mathcal{E} \vdash (E_0 \oplus E_1) : {T_r}^{L}}$$

  - Case #2: mixed-security

$$\frac{\exists \oplus \in \mathcal{E} \quad \mathcal{E} \vdash \oplus : (\rightarrow T_r, T_0, T_1) \quad \mathcal{E} \vdash E_0 : {T_0}^{H:L_0} \quad \mathcal{E} \vdash E_1 : {T_1}^{L}}{\mathcal{E} \vdash (E_0 \oplus E_1) : {T_r}^{H:L_0}}$$

$$\frac{\exists \oplus \in \mathcal{E} \quad \mathcal{E} \vdash \oplus : (\rightarrow T_r, T_0, T_1) \quad \mathcal{E} \vdash E_0 : {T_0}^{L} \quad \mathcal{E} \vdash E_1 : {T_1}^{H:L_1}}{\mathcal{E} \vdash (E_0 \oplus E_1) : {T_r}^{H:L_1}}$$

  - Case #3: high-security

$$\frac{\exists \oplus \in \mathcal{E} \quad \mathcal{E} \vdash \oplus : (\rightarrow T_r, T_0, T_1) \quad \mathcal{E} \vdash E_0 : {T_0}^{H:L_0} \quad \mathcal{E} \vdash E_1 : {T_1}^{H:L_1}}{\mathcal{E} \vdash (E_0 \oplus E_1) : {T_r}^{H:((L_0 \cup L_1) \setminus (L_0 \cap L_1))}}$$

Approach

- Heuristic recovery rules allow correction of (some) **type errors**:
  - Error #1: a **weak map** wrt.

    ```
    r := m[ i ];
    ```

    is where a high-security `i` indexes a low-security `m`.
  - Error #2: a **weak store** wrt.

    ```
    r := x;
    ```

    is where a high-security `x` is assigned to a low-security `r`.
  - Error #3: a **mask collision** wrt.

    ```
    r := x;
    ```

    is where a high-security `x` is assigned to a high-security `r` with a different mask set.
  - Error #4: a **mask revelation** wrt.

    ```
    r := x ^ y;
    ```

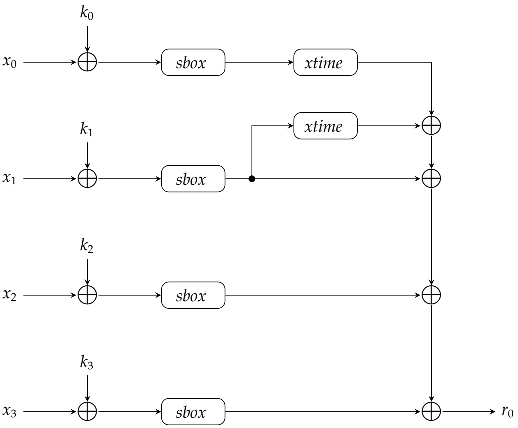
    is where computation involving a high-security `x` and `y` results in a low-security result.

Illustrative Example

```
typedef byte :=   bits[ 8 ];
typedef col  := vector[ 4 ] of byte;

def  sbox : map[ byte -> byte ];
def xtime : map[ byte -> byte ];

def mix( x : col { H<a> }, k : col ) : col : { H<b> } {

  def t : col, r : col { H<b> };

  seq i := 0 to 3 {
    t[ i ] := sbox[ x[ i ] ^ k[ i ] ];
  }

  seq i := 0 to 3 {
    r[ i ] := xtime[ t[ ( i + 0 ) % 4 ] ] ^ //   2 t_0 => 2 t_0
              xtime[ t[ ( i + 1 ) % 4 ] ] ^ // + 2 t_1 => 2 t_0 + 2 t_1
                     t[ ( i + 1 ) % 4 ]   ^ // +   t_1 => 2 t_0 + 3 t_1
                     t[ ( i + 2 ) % 4 ]   ^ // +   t_2 => 2 t_0 + 3 t_1 + t_2
                     t[ ( i + 3 ) % 4 ]   ; // +   t_3 => 2 t_0 + 3 t_1 + t_2 + t_3
  }

  return r;
}
```

Illustrative Example

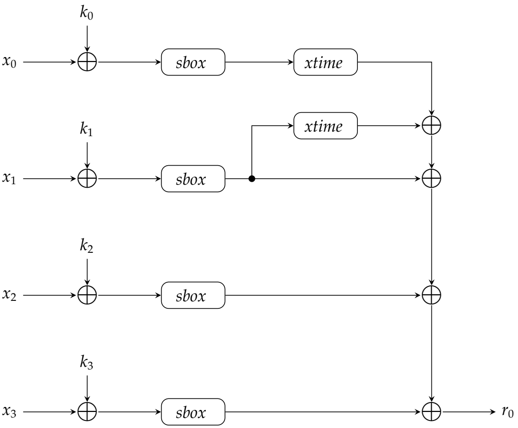
- Step #1: source program is unrolled and translated into a Data-Flow Graph (DFG) and symbol table.



$$\mathcal{E} = \left\{ \begin{array}{lcl} mix & : & {\mathbb{Z}_{256}^{4}}^{L} \times {\mathbb{Z}_{256}^{4}}^{H:\langle a \rangle} \rightarrow {\mathbb{Z}_{256}^{4}}^{H:\langle b \rangle} \\ sbox & : & \mathbb{Z}_{256}{}^{L} \rightarrow \mathbb{Z}_{256}{}^{L} \\ xtime & : & \mathbb{Z}_{256}{}^{L} \rightarrow \mathbb{Z}_{256}{}^{L} \\ x_0 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ x_1 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ x_2 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ x_3 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ k_0 & : & \mathbb{Z}_{256}{}^{L} \\ k_1 & : & \mathbb{Z}_{256}{}^{L} \\ k_2 & : & \mathbb{Z}_{256}{}^{L} \\ k_3 & : & \mathbb{Z}_{256}{}^{L} \\ t_0 & : & \mathbb{Z}_{256}{}^{L} \\ t_1 & : & \mathbb{Z}_{256}{}^{L} \\ t_2 & : & \mathbb{Z}_{256}{}^{L} \\ t_3 & : & \mathbb{Z}_{256}{}^{L} \\ r_0 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\ r_1 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\ r_2 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\ r_3 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \end{array} \right.$$

Illustrative Example

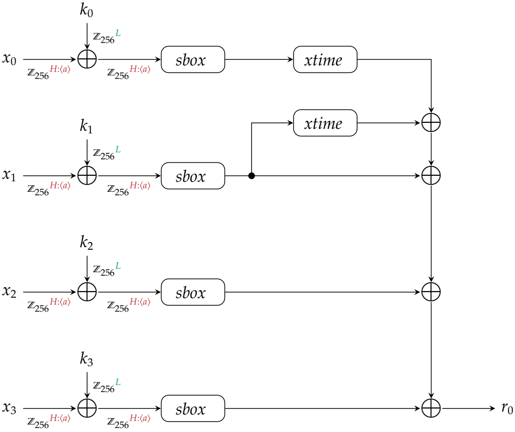
- Step #2: x_i is masked with a, k_i is unmasked; XOR output is masked with a.



$$\mathcal{E} = \left\{ \begin{array}{lcl} mix & : & \mathbb{Z}_{256}^{4}{}^{L} \times \mathbb{Z}_{256}^{4}{}^{H:\langle a\rangle} \rightarrow \mathbb{Z}_{256}^{4}{}^{H:\langle b\rangle} \\ sbox & : & \mathbb{Z}_{256}{}^{L} \rightarrow \mathbb{Z}_{256}{}^{L} \\ xtime & : & \mathbb{Z}_{256}{}^{L} \rightarrow \mathbb{Z}_{256}{}^{L} \\ x_0 & : & \mathbb{Z}_{256}{}^{H:\langle a\rangle} \\ x_1 & : & \mathbb{Z}_{256}{}^{H:\langle a\rangle} \\ x_2 & : & \mathbb{Z}_{256}{}^{H:\langle a\rangle} \\ x_3 & : & \mathbb{Z}_{256}{}^{H:\langle a\rangle} \\ k_0 & : & \mathbb{Z}_{256}{}^{L} \\ k_1 & : & \mathbb{Z}_{256}{}^{L} \\ k_2 & : & \mathbb{Z}_{256}{}^{L} \\ k_3 & : & \mathbb{Z}_{256}{}^{L} \\ t_0 & : & \mathbb{Z}_{256}{}^{L} \\ t_1 & : & \mathbb{Z}_{256}{}^{L} \\ t_2 & : & \mathbb{Z}_{256}{}^{L} \\ t_3 & : & \mathbb{Z}_{256}{}^{L} \\ r_0 & : & \mathbb{Z}_{256}{}^{H:\langle b\rangle} \\ r_1 & : & \mathbb{Z}_{256}{}^{H:\langle b\rangle} \\ r_2 & : & \mathbb{Z}_{256}{}^{H:\langle b\rangle} \\ r_3 & : & \mathbb{Z}_{256}{}^{H:\langle b\rangle} \end{array} \right.$$

Illustrative Example

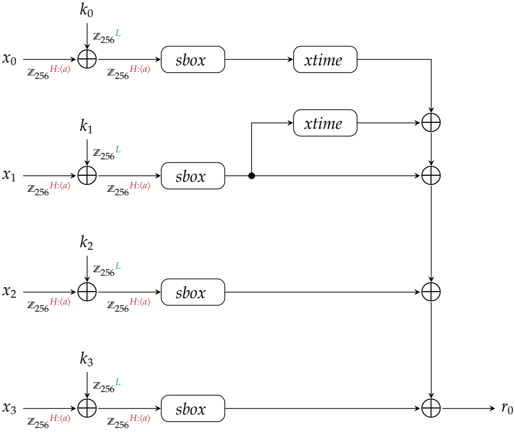
- **Step #2**: x_i is masked with a, k_i is unmasked; XOR output is masked with a.



$$\mathcal{E} = \left\{ \begin{array}{lcl} mix & : & \mathbb{Z}_{256}^{4}{}^{L} \times \mathbb{Z}_{256}^{4}{}^{H:\langle a\rangle} \rightarrow \mathbb{Z}_{256}^{4}{}^{H:\langle b\rangle} \\ sbox & : & \mathbb{Z}_{256}{}^{L} \rightarrow \mathbb{Z}_{256}{}^{L} \\ xtime & : & \mathbb{Z}_{256}{}^{L} \rightarrow \mathbb{Z}_{256}{}^{L} \\ x_0 & : & \mathbb{Z}_{256}{}^{H:\langle a\rangle} \\ x_1 & : & \mathbb{Z}_{256}{}^{H:\langle a\rangle} \\ x_2 & : & \mathbb{Z}_{256}{}^{H:\langle a\rangle} \\ x_3 & : & \mathbb{Z}_{256}{}^{H:\langle a\rangle} \\ k_0 & : & \mathbb{Z}_{256}{}^{L} \\ k_1 & : & \mathbb{Z}_{256}{}^{L} \\ k_2 & : & \mathbb{Z}_{256}{}^{L} \\ k_3 & : & \mathbb{Z}_{256}{}^{L} \\ t_0 & : & \mathbb{Z}_{256}{}^{L} \\ t_1 & : & \mathbb{Z}_{256}{}^{L} \\ t_2 & : & \mathbb{Z}_{256}{}^{L} \\ t_3 & : & \mathbb{Z}_{256}{}^{L} \\ r_0 & : & \mathbb{Z}_{256}{}^{H:\langle b\rangle} \\ r_1 & : & \mathbb{Z}_{256}{}^{H:\langle b\rangle} \\ r_2 & : & \mathbb{Z}_{256}{}^{H:\langle b\rangle} \\ r_3 & : & \mathbb{Z}_{256}{}^{H:\langle b\rangle} \end{array} \right.$$

Illustrative Example

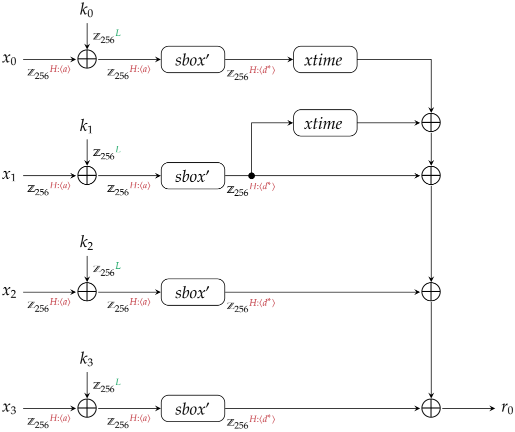
- Step #3: a weak map occurs since type of *sbox* input doesn't match;
  1. synthesise an $\mathcal{E} \vdash sbox' : \mathbb{Z}_{256}{}^{H:\langle c^* \rangle} \rightarrow \mathbb{Z}_{256}{}^{H:\langle d^* \rangle}$ inc. support code, then
  2. unify a and c^*.



$$\mathcal{E} = \left\{ \begin{array}{lcl} mix & : & \mathbb{Z}_{256}^{4}{}^{L} \times \mathbb{Z}_{256}^{4}{}^{H:\langle a \rangle} \rightarrow \mathbb{Z}_{256}^{4}{}^{H:\langle b \rangle} \\ sbox & : & \mathbb{Z}_{256}{}^{L} \rightarrow \mathbb{Z}_{256}{}^{L} \\ xtime & : & \mathbb{Z}_{256}{}^{L} \rightarrow \mathbb{Z}_{256}{}^{L} \\ x_0 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ x_1 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ x_2 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ x_3 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ k_0 & : & \mathbb{Z}_{256}{}^{L} \\ k_1 & : & \mathbb{Z}_{256}{}^{L} \\ k_2 & : & \mathbb{Z}_{256}{}^{L} \\ k_3 & : & \mathbb{Z}_{256}{}^{L} \\ t_0 & : & \mathbb{Z}_{256}{}^{L} \\ t_1 & : & \mathbb{Z}_{256}{}^{L} \\ t_2 & : & \mathbb{Z}_{256}{}^{L} \\ t_3 & : & \mathbb{Z}_{256}{}^{L} \\ r_0 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\ r_1 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\ r_2 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\ r_3 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \end{array} \right.$$

Illustrative Example

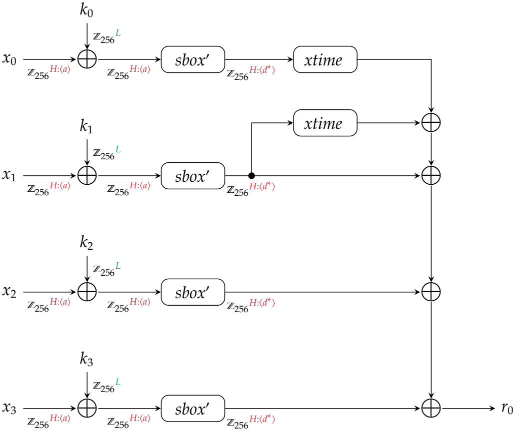
- Step #3: a weak map occurs since type of *sbox* input doesn't match;
  1. synthesise an $\mathcal{E} \vdash sbox' : \mathbb{Z}_{256}{}^{H:\langle c^* \rangle} \rightarrow \mathbb{Z}_{256}{}^{H:\langle d^* \rangle}$ inc. support code, then
  2. unify a and c^*.



$$\mathcal{E} = \left\{ \begin{array}{lcl} mix & : & \mathbb{Z}_{256}^{4}{}^{L} \times \mathbb{Z}_{256}^{4}{}^{H:\langle a \rangle} \rightarrow \mathbb{Z}_{256}^{4}{}^{H:\langle b \rangle} \\ sbox' & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \rightarrow \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \\ xtime & : & \mathbb{Z}_{256}{}^{L} \rightarrow \mathbb{Z}_{256}{}^{L} \\ x_0 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ x_1 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ x_2 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ x_3 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ k_0 & : & \mathbb{Z}_{256}{}^{L} \\ k_1 & : & \mathbb{Z}_{256}{}^{L} \\ k_2 & : & \mathbb{Z}_{256}{}^{L} \\ k_3 & : & \mathbb{Z}_{256}{}^{L} \\ t_0 & : & \mathbb{Z}_{256}{}^{L} \\ t_1 & : & \mathbb{Z}_{256}{}^{L} \\ t_2 & : & \mathbb{Z}_{256}{}^{L} \\ t_3 & : & \mathbb{Z}_{256}{}^{L} \\ r_0 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\ r_1 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\ r_2 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\ r_3 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \end{array} \right.$$

Illustrative Example

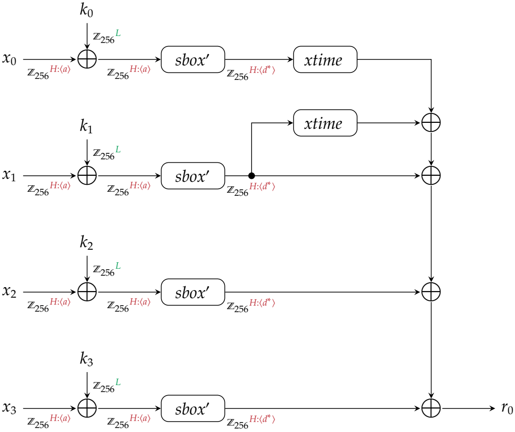
- Step #4: a weak store error occurs since type of t_i doesn't match;
  1. upgrade type $\mathcal{E} \vdash t_i : \mathbb{Z}_{256}{}^{H:\langle e^* \rangle}$, then
  2. unify d^* and e^*.



$$\mathcal{E} = \left\{ \begin{array}{lcl} mix & : & \mathbb{Z}_{256}^{4}{}^{L} \times \mathbb{Z}_{256}^{4}{}^{H:\langle a \rangle} \rightarrow \mathbb{Z}_{256}^{4}{}^{H:\langle b \rangle} \\ sbox' & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \rightarrow \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \\ xtime & : & \mathbb{Z}_{256}{}^{L} \rightarrow \mathbb{Z}_{256}{}^{L} \\ x_0 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ x_1 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ x_2 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ x_3 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ k_0 & : & \mathbb{Z}_{256}{}^{L} \\ k_1 & : & \mathbb{Z}_{256}{}^{L} \\ k_2 & : & \mathbb{Z}_{256}{}^{L} \\ k_3 & : & \mathbb{Z}_{256}{}^{L} \\ t_0 & : & \mathbb{Z}_{256}{}^{L} \\ t_1 & : & \mathbb{Z}_{256}{}^{L} \\ t_2 & : & \mathbb{Z}_{256}{}^{L} \\ t_3 & : & \mathbb{Z}_{256}{}^{L} \\ r_0 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\ r_1 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\ r_2 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\ r_3 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \end{array} \right.$$

Illustrative Example

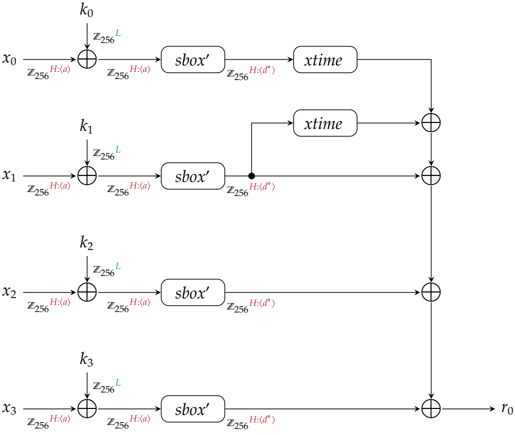
- Step #4: a weak store error occurs since type of t_i doesn't match;
  1. upgrade type $\mathcal{E} \vdash t_i : \mathbb{Z}_{256}{}^{H:\langle e^* \rangle}$, then
  2. unify d^* and e^*.



$$\mathcal{E} = \left\{ \begin{array}{lcl} mix & : & \mathbb{Z}_{256}^{4}{}^{L} \times \mathbb{Z}_{256}^{4}{}^{H:\langle a \rangle} \rightarrow \mathbb{Z}_{256}^{4}{}^{H:\langle b \rangle} \\ sbox' & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \rightarrow \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \\ xtime & : & \mathbb{Z}_{256}{}^{L} \rightarrow \mathbb{Z}_{256}{}^{L} \\ x_0 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ x_1 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ x_2 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ x_3 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ k_0 & : & \mathbb{Z}_{256}{}^{L} \\ k_1 & : & \mathbb{Z}_{256}{}^{L} \\ k_2 & : & \mathbb{Z}_{256}{}^{L} \\ k_3 & : & \mathbb{Z}_{256}{}^{L} \\ t_0 & : & \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \\ t_1 & : & \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \\ t_2 & : & \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \\ t_3 & : & \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \\ r_0 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\ r_1 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\ r_2 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\ r_3 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \end{array} \right.$$

Illustrative Example

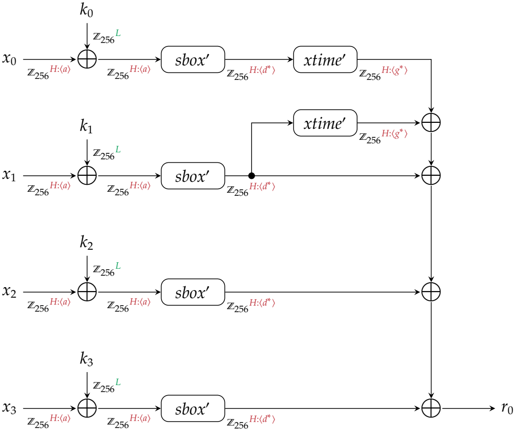
- Step #5: a weak map error occurs since type of *xtime* input doesn't match;
  1. synthesise an $\mathcal{E} \vdash xtime' : \mathbb{Z}_{256}{}^{H:\langle f^* \rangle} \rightarrow \mathbb{Z}_{256}{}^{H:\langle g^* \rangle}$ inc. support code, then
  2. unify d^* and f^*.



$$
\mathcal{E} = \left\{ \begin{array}{lcl}
mix & : & \mathbb{Z}_{256}^{4}{}^{L} \times \mathbb{Z}_{256}^{4}{}^{H:\langle a \rangle} \rightarrow \mathbb{Z}_{256}^{4}{}^{H:\langle b \rangle} \\
sbox' & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \rightarrow \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \\
xtime & : & \mathbb{Z}_{256}{}^{L} \rightarrow \mathbb{Z}_{256}{}^{L} \\
x_0 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\
x_1 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\
x_2 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\
x_3 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\
k_0 & : & \mathbb{Z}_{256}{}^{L} \\
k_1 & : & \mathbb{Z}_{256}{}^{L} \\
k_2 & : & \mathbb{Z}_{256}{}^{L} \\
k_3 & : & \mathbb{Z}_{256}{}^{L} \\
t_0 & : & \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \\
t_1 & : & \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \\
t_2 & : & \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \\
t_3 & : & \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \\
r_0 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\
r_1 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\
r_2 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\
r_3 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle}
\end{array} \right.
$$

Illustrative Example

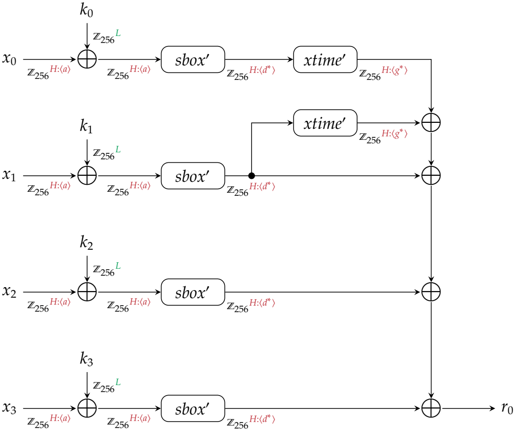
- Step #5: a weak map error occurs since type of *xtime* input doesn't match;
  1. synthesise an $\mathcal{E} \vdash xtime' : \mathbb{Z}_{256}{}^{H:\langle f^* \rangle} \rightarrow \mathbb{Z}_{256}{}^{H:\langle g^* \rangle}$ inc. support code, then
  2. unify d^* and f^*.



$$\mathcal{E} = \left\{ \begin{array}{lcl} mix & : & \mathbb{Z}_{256}^{4}{}^{L} \times \mathbb{Z}_{256}^{4}{}^{H:\langle a \rangle} \rightarrow \mathbb{Z}_{256}^{4}{}^{H:\langle b \rangle} \\ sbox' & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \rightarrow \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \\ xtime' & : & \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \rightarrow \mathbb{Z}_{256}{}^{H:\langle g^* \rangle} \\ x_0 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ x_1 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ x_2 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ x_3 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ k_0 & : & \mathbb{Z}_{256}{}^{L} \\ k_1 & : & \mathbb{Z}_{256}{}^{L} \\ k_2 & : & \mathbb{Z}_{256}{}^{L} \\ k_3 & : & \mathbb{Z}_{256}{}^{L} \\ t_0 & : & \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \\ t_1 & : & \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \\ t_2 & : & \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \\ t_3 & : & \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \\ r_0 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\ r_1 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\ r_2 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\ r_3 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \end{array} \right.$$

Illustrative Example

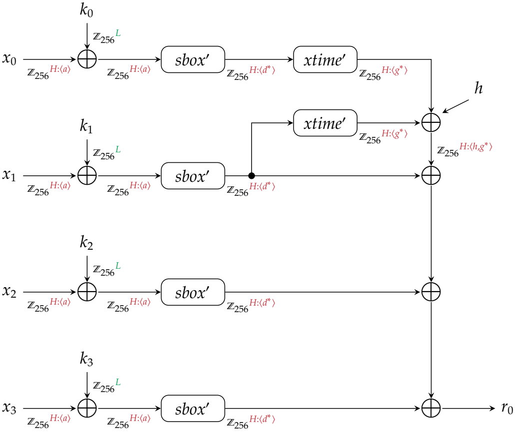
- Step #6: a mask revelation error occurs since XOR inputs have same mask;
  1. inject support code to add a new mask h to one operand, meaning
  2. the XOR output is masked with $h \oplus g^*$.



$$\mathcal{E} = \left\{ \begin{array}{lcl} mix & : & \mathbb{Z}_{256}^{4}{}^{L} \times \mathbb{Z}_{256}^{4}{}^{H:\langle a \rangle} \rightarrow \mathbb{Z}_{256}^{4}{}^{H:\langle b \rangle} \\ sbox' & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \rightarrow \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \\ xtime' & : & \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \rightarrow \mathbb{Z}_{256}{}^{H:\langle g^* \rangle} \\ x_0 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ x_1 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ x_2 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ x_3 & : & \mathbb{Z}_{256}{}^{H:\langle a \rangle} \\ k_0 & : & \mathbb{Z}_{256}{}^{L} \\ k_1 & : & \mathbb{Z}_{256}{}^{L} \\ k_2 & : & \mathbb{Z}_{256}{}^{L} \\ k_3 & : & \mathbb{Z}_{256}{}^{L} \\ t_0 & : & \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \\ t_1 & : & \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \\ t_2 & : & \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \\ t_3 & : & \mathbb{Z}_{256}{}^{H:\langle d^* \rangle} \\ r_0 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\ r_1 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\ r_2 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \\ r_3 & : & \mathbb{Z}_{256}{}^{H:\langle b \rangle} \end{array} \right.$$

Illustrative Example

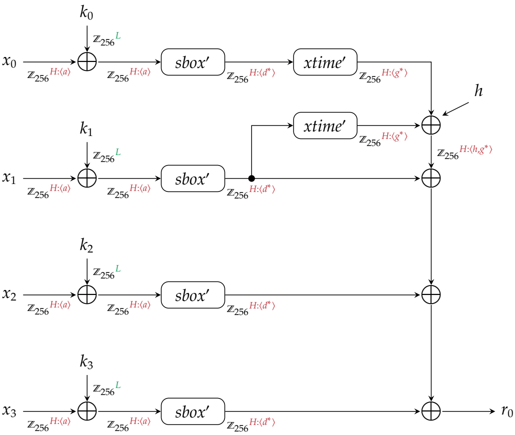
- Step #6: a mask revelation error occurs since XOR inputs have same mask;
  1. inject support code to add a new mask h to one operand, meaning
  2. the XOR output is masked with $h \oplus g^*$.



$$\mathcal{E} = \left\{ \begin{array}{lcl} mix & : & \mathbb{Z}_{256}^{4\,L} \times \mathbb{Z}_{256}^{4\,H:\langle a \rangle} \rightarrow \mathbb{Z}_{256}^{4\,H:\langle b \rangle} \\ sbox' & : & \mathbb{Z}_{256}^{H:\langle a \rangle} \rightarrow \mathbb{Z}_{256}^{H:\langle d^* \rangle} \\ xtime' & : & \mathbb{Z}_{256}^{H:\langle d^* \rangle} \rightarrow \mathbb{Z}_{256}^{H:\langle g^* \rangle} \\ x_0 & : & \mathbb{Z}_{256}^{H:\langle a \rangle} \\ x_1 & : & \mathbb{Z}_{256}^{H:\langle a \rangle} \\ x_2 & : & \mathbb{Z}_{256}^{H:\langle a \rangle} \\ x_3 & : & \mathbb{Z}_{256}^{H:\langle a \rangle} \\ k_0 & : & \mathbb{Z}_{256}^{L} \\ k_1 & : & \mathbb{Z}_{256}^{L} \\ k_2 & : & \mathbb{Z}_{256}^{L} \\ k_3 & : & \mathbb{Z}_{256}^{L} \\ t_0 & : & \mathbb{Z}_{256}^{H:\langle d^* \rangle} \\ t_1 & : & \mathbb{Z}_{256}^{H:\langle d^* \rangle} \\ t_2 & : & \mathbb{Z}_{256}^{H:\langle d^* \rangle} \\ t_3 & : & \mathbb{Z}_{256}^{H:\langle d^* \rangle} \\ r_0 & : & \mathbb{Z}_{256}^{H:\langle b \rangle} \\ r_1 & : & \mathbb{Z}_{256}^{H:\langle b \rangle} \\ r_2 & : & \mathbb{Z}_{256}^{H:\langle b \rangle} \\ r_3 & : & \mathbb{Z}_{256}^{H:\langle b \rangle} \end{array} \right.$$

Illustrative Example

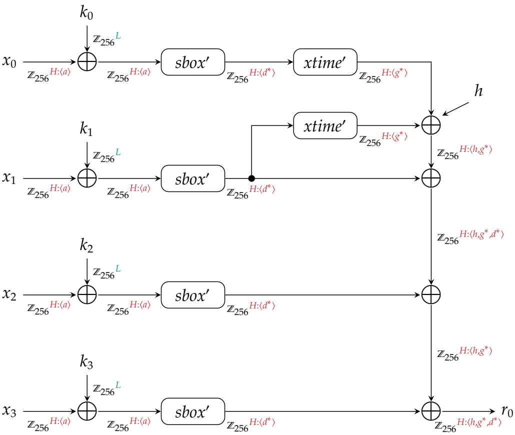
- Step #7: XOR (tree) output is masked with $h \oplus g^* \oplus d^*$.



$$\mathcal{E} = \left\{ \begin{array}{lcl} mix & : & \mathbb{Z}_{256}^{4\ L} \times \mathbb{Z}_{256}^{4\ H:\langle a \rangle} \rightarrow \mathbb{Z}_{256}^{4\ H:\langle b \rangle} \\ sbox' & : & \mathbb{Z}_{256}^{H:\langle a \rangle} \rightarrow \mathbb{Z}_{256}^{H:\langle d^* \rangle} \\ xtime' & : & \mathbb{Z}_{256}^{H:\langle d^* \rangle} \rightarrow \mathbb{Z}_{256}^{H:\langle g^* \rangle} \\ x_0 & : & \mathbb{Z}_{256}^{H:\langle a \rangle} \\ x_1 & : & \mathbb{Z}_{256}^{H:\langle a \rangle} \\ x_2 & : & \mathbb{Z}_{256}^{H:\langle a \rangle} \\ x_3 & : & \mathbb{Z}_{256}^{H:\langle a \rangle} \\ k_0 & : & \mathbb{Z}_{256}^{L} \\ k_1 & : & \mathbb{Z}_{256}^{L} \\ k_2 & : & \mathbb{Z}_{256}^{L} \\ k_3 & : & \mathbb{Z}_{256}^{L} \\ t_0 & : & \mathbb{Z}_{256}^{H:\langle d^* \rangle} \\ t_1 & : & \mathbb{Z}_{256}^{H:\langle d^* \rangle} \\ t_2 & : & \mathbb{Z}_{256}^{H:\langle d^* \rangle} \\ t_3 & : & \mathbb{Z}_{256}^{H:\langle d^* \rangle} \\ r_0 & : & \mathbb{Z}_{256}^{H:\langle b \rangle} \\ r_1 & : & \mathbb{Z}_{256}^{H:\langle b \rangle} \\ r_2 & : & \mathbb{Z}_{256}^{H:\langle b \rangle} \\ r_3 & : & \mathbb{Z}_{256}^{H:\langle b \rangle} \end{array} \right.$$

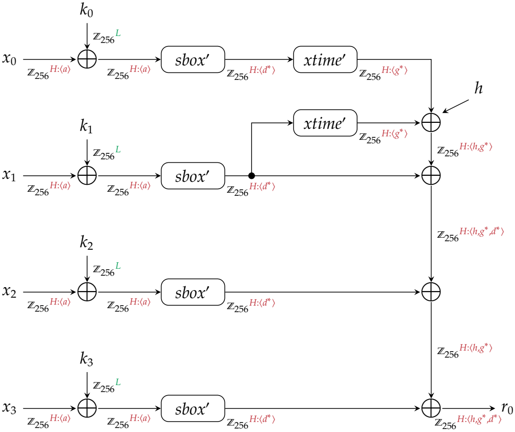
Illustrative Example

- **Step #7**: XOR (tree) output is masked with $h \oplus g^* \oplus d^*$.



$$\mathcal{E} = \left\{ \begin{array}{lcl}
mix & : & {\mathbb{Z}_{256}^4}^{L} \times {\mathbb{Z}_{256}^4}^{H:\langle a \rangle} \rightarrow {\mathbb{Z}_{256}^4}^{H:\langle b \rangle} \\
sbox' & : & \mathbb{Z}_{256}^{H:\langle a \rangle} \rightarrow \mathbb{Z}_{256}^{H:\langle d^* \rangle} \\
xtime' & : & \mathbb{Z}_{256}^{H:\langle d^* \rangle} \rightarrow \mathbb{Z}_{256}^{H:\langle g^* \rangle} \\
x_0 & : & \mathbb{Z}_{256}^{H:\langle a \rangle} \\
x_1 & : & \mathbb{Z}_{256}^{H:\langle a \rangle} \\
x_2 & : & \mathbb{Z}_{256}^{H:\langle a \rangle} \\
x_3 & : & \mathbb{Z}_{256}^{H:\langle a \rangle} \\
k_0 & : & \mathbb{Z}_{256}^{L} \\
k_1 & : & \mathbb{Z}_{256}^{L} \\
k_2 & : & \mathbb{Z}_{256}^{L} \\
k_3 & : & \mathbb{Z}_{256}^{L} \\
t_0 & : & \mathbb{Z}_{256}^{H:\langle d^* \rangle} \\
t_1 & : & \mathbb{Z}_{256}^{H:\langle d^* \rangle} \\
t_2 & : & \mathbb{Z}_{256}^{H:\langle d^* \rangle} \\
t_3 & : & \mathbb{Z}_{256}^{H:\langle d^* \rangle} \\
r_0 & : & \mathbb{Z}_{256}^{H:\langle b \rangle} \\
r_1 & : & \mathbb{Z}_{256}^{H:\langle b \rangle} \\
r_2 & : & \mathbb{Z}_{256}^{H:\langle b \rangle} \\
r_3 & : & \mathbb{Z}_{256}^{H:\langle b \rangle}
\end{array} \right.$$

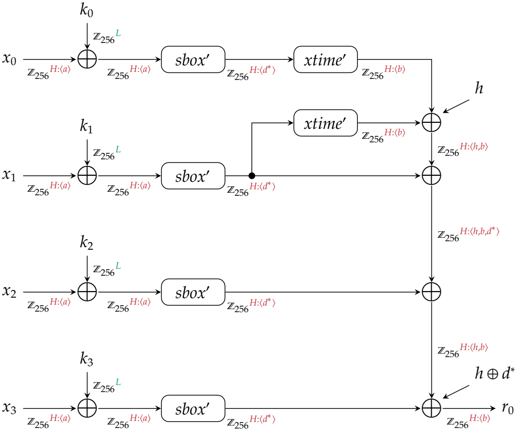
- Step #8: a mask collision occurs since r_i is masked with b;
  1. unify g^* and b, then
  2. inject support code to equalise masks.



$$
\mathcal{E} = \left\{
\begin{array}{lcl}
mix & : & \mathbb{Z}_{256}^{4}{}^{L} \times \mathbb{Z}_{256}^{4}{}^{H:\langle a\rangle} \rightarrow \mathbb{Z}_{256}^{4}{}^{H:\langle b\rangle} \\
sbox' & : & \mathbb{Z}_{256}{}^{H:\langle a\rangle} \rightarrow \mathbb{Z}_{256}{}^{H:\langle d^*\rangle} \\
xtime' & : & \mathbb{Z}_{256}{}^{H:\langle d^*\rangle} \rightarrow \mathbb{Z}_{256}{}^{H:\langle g^*\rangle} \\
x_0 & : & \mathbb{Z}_{256}{}^{H:\langle a\rangle} \\
x_1 & : & \mathbb{Z}_{256}{}^{H:\langle a\rangle} \\
x_2 & : & \mathbb{Z}_{256}{}^{H:\langle a\rangle} \\
x_3 & : & \mathbb{Z}_{256}{}^{H:\langle a\rangle} \\
k_0 & : & \mathbb{Z}_{256}{}^{L} \\
k_1 & : & \mathbb{Z}_{256}{}^{L} \\
k_2 & : & \mathbb{Z}_{256}{}^{L} \\
k_3 & : & \mathbb{Z}_{256}{}^{L} \\
t_0 & : & \mathbb{Z}_{256}{}^{H:\langle d^*\rangle} \\
t_1 & : & \mathbb{Z}_{256}{}^{H:\langle d^*\rangle} \\
t_2 & : & \mathbb{Z}_{256}{}^{H:\langle d^*\rangle} \\
t_3 & : & \mathbb{Z}_{256}{}^{H:\langle d^*\rangle} \\
r_0 & : & \mathbb{Z}_{256}{}^{H:\langle b\rangle} \\
r_1 & : & \mathbb{Z}_{256}{}^{H:\langle b\rangle} \\
r_2 & : & \mathbb{Z}_{256}{}^{H:\langle b\rangle} \\
r_3 & : & \mathbb{Z}_{256}{}^{H:\langle b\rangle}
\end{array}
\right.
$$

Illustrative Example

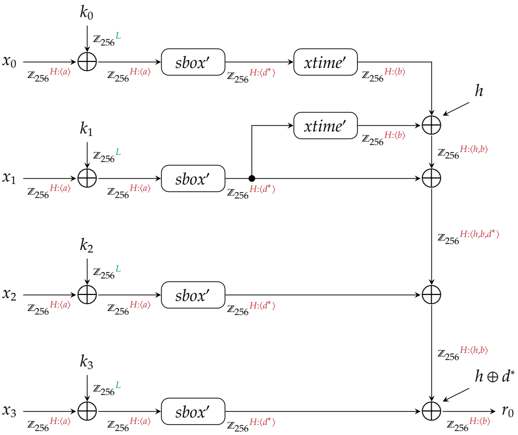
- Step #8: a mask collision occurs since r_i is masked with b;
  1. unify g^* and b, then
  2. inject support code to equalise masks.



$$\mathcal{E} = \begin{cases} mix & : \ \mathbb{Z}^4_{256}{}^{L} \times \mathbb{Z}^4_{256}{}^{H:\langle a\rangle} \to \mathbb{Z}^4_{256}{}^{H:\langle b\rangle} \\ sbox' & : \ \mathbb{Z}_{256}{}^{H:\langle a\rangle} \to \mathbb{Z}_{256}{}^{H:\langle d^*\rangle} \\ xtime' & : \ \mathbb{Z}_{256}{}^{H:\langle d^*\rangle} \to \mathbb{Z}_{256}{}^{H:\langle b\rangle} \\ x_0 & : \ \mathbb{Z}_{256}{}^{H:\langle a\rangle} \\ x_1 & : \ \mathbb{Z}_{256}{}^{H:\langle a\rangle} \\ x_2 & : \ \mathbb{Z}_{256}{}^{H:\langle a\rangle} \\ x_3 & : \ \mathbb{Z}_{256}{}^{H:\langle a\rangle} \\ k_0 & : \ \mathbb{Z}_{256}{}^{L} \\ k_1 & : \ \mathbb{Z}_{256}{}^{L} \\ k_2 & : \ \mathbb{Z}_{256}{}^{L} \\ k_3 & : \ \mathbb{Z}_{256}{}^{L} \\ t_0 & : \ \mathbb{Z}_{256}{}^{H:\langle d^*\rangle} \\ t_1 & : \ \mathbb{Z}_{256}{}^{H:\langle d^*\rangle} \\ t_2 & : \ \mathbb{Z}_{256}{}^{H:\langle d^*\rangle} \\ t_3 & : \ \mathbb{Z}_{256}{}^{H:\langle d^*\rangle} \\ r_0 & : \ \mathbb{Z}_{256}{}^{H:\langle b\rangle} \\ r_1 & : \ \mathbb{Z}_{256}{}^{H:\langle b\rangle} \\ r_2 & : \ \mathbb{Z}_{256}{}^{H:\langle b\rangle} \\ r_3 & : \ \mathbb{Z}_{256}{}^{H:\langle b\rangle} \end{cases}$$

Illustrative Example

- **Result**: all intermediates have non-empty mask sets ∴ success.



$$\mathcal{E} = \left\{ \begin{array}{lcl}
mix & : & \mathbb{Z}_{256}^{4}{}^{L} \times \mathbb{Z}_{256}^{4}{}^{H:\langle a\rangle} \rightarrow \mathbb{Z}_{256}^{4}{}^{H:\langle b\rangle} \\
sbox' & : & \mathbb{Z}_{256}{}^{H:\langle a\rangle} \rightarrow \mathbb{Z}_{256}{}^{H:\langle d^*\rangle} \\
xtime' & : & \mathbb{Z}_{256}{}^{H:\langle d^*\rangle} \rightarrow \mathbb{Z}_{256}{}^{H:\langle b\rangle} \\
x_0 & : & \mathbb{Z}_{256}{}^{H:\langle a\rangle} \\
x_1 & : & \mathbb{Z}_{256}{}^{H:\langle a\rangle} \\
x_2 & : & \mathbb{Z}_{256}{}^{H:\langle a\rangle} \\
x_3 & : & \mathbb{Z}_{256}{}^{H:\langle a\rangle} \\
k_0 & : & \mathbb{Z}_{256}{}^{L} \\
k_1 & : & \mathbb{Z}_{256}{}^{L} \\
k_2 & : & \mathbb{Z}_{256}{}^{L} \\
k_3 & : & \mathbb{Z}_{256}{}^{L} \\
t_0 & : & \mathbb{Z}_{256}{}^{H:\langle d^*\rangle} \\
t_1 & : & \mathbb{Z}_{256}{}^{H:\langle d^*\rangle} \\
t_2 & : & \mathbb{Z}_{256}{}^{H:\langle d^*\rangle} \\
t_3 & : & \mathbb{Z}_{256}{}^{H:\langle d^*\rangle} \\
r_0 & : & \mathbb{Z}_{256}{}^{H:\langle b\rangle} \\
r_1 & : & \mathbb{Z}_{256}{}^{H:\langle b\rangle} \\
r_2 & : & \mathbb{Z}_{256}{}^{H:\langle b\rangle} \\
r_3 & : & \mathbb{Z}_{256}{}^{H:\langle b\rangle}
\end{array} \right.$$

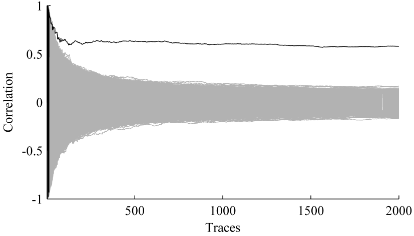
Evaluation

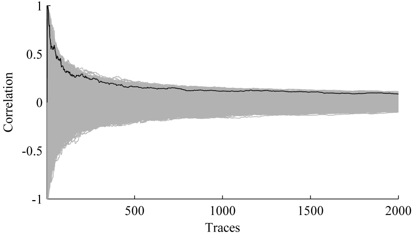
- There's no magic here, the result
  1. is *at best* as good as the masking scheme (which resists 1-st order DPA only), plus
  2. *only* guarantees necessary condition of mask validity, not (various other) sufficient conditions.
- Even so, we'd like to evaluate the result wrt. some M:
  - Option #1: **formal verification** (cf. Briais at al. [2]), to check if
    1. function is preserved, e.g., automatically masked AES still computes AES,
    2. quality is improved, i.e., automatically masked AES gives intended security advantage.
  - Option #2: **user studies**!
  - Option #3: **experimental**:
    1. apply representative DPA attack, and
    2. measure performance

    relative to unprotected reference, plus hand-written version.

Evaluation

- The good:





Evaluation

- The bad: even with a small example, we migrate towards a special-purpose language:
  1. we already need annotation (although this could be performed via `pragma` or similar) and somewhat rich type system,
  2. masking (full) AES requires description of interface with caller,
  3. masking PRESENT, for example, requires description of permutation.

Conclusions

- This first step
  - satisfies *most* initial goals, but
  - implies some unattractive residual requirements.
- Improvements could target a host of **open questions**, e.g., how to
  1. integrate within a non-prototype compiler (e.g., LLVM),
  2. deal efficiently with control-flow,
  3. generalise to other and mixed forms of masking, and
  4. control interaction between other tool-chain components (e.g., instruction scheduling, register allocation)

  some of which hint at a need to *unify* underlying theme of "**computing on encrypted data**" (cf. FHE and friends).

Conclusions

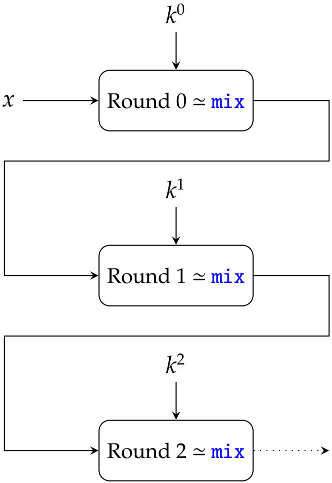
Questions?

References and Further Reading

[1] A.G. Bayrak, F. Regazzoni, P. Brisk, F.-X. Standaert, and P. Ienne.
A first step towards automatic application of power analysis countermeasures.
In *Design Automation Conference (DAC)*, pages 230–235, 2011.

[2] S. Briais, S. Guilley, and J.-L. Danger.
A formal study of two physical countermeasures against side channel attacks.
In *Security Proofs for Embedded Systems (PROOFS)*, 2012.

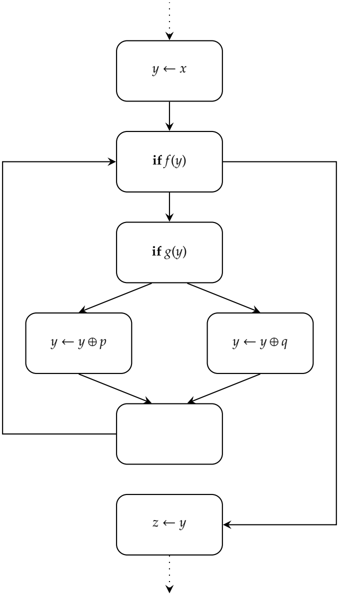
Extra: extending to full AES



- Problem: the type inference works as before, but we get *multiple* masked *sbox′* maps.
- Either
  - solution #1: delegate to programmer so k^i is masked suitably, or
  - solution #2: delegate to programmer to guide compiler into injection of inter-round remasking

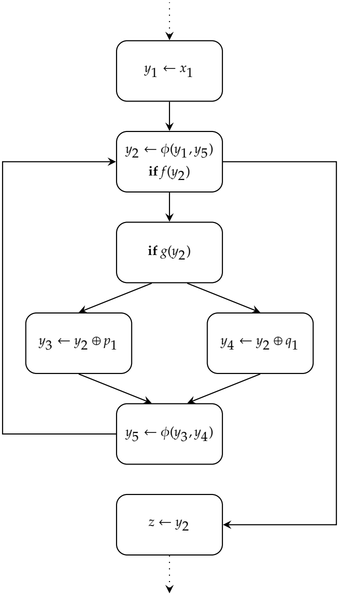
  st. all inputs to single *sbox′* are under the same mask: *neither* is very satisfactory.

Extra: coping with control-flow



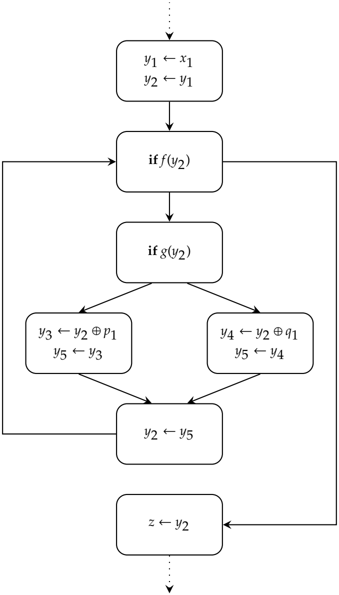
- To analyse a program in SSA form we
  1. start with a normal DFG,
  2. give unique labels to each symbol and insert "magic" ϕ-nodes to support data-flow analysis, then
  3. once the analysis is finished, eliminate ϕ-nodes by pushing a "patch" assignment into parent nodes.

Extra: coping with control-flow



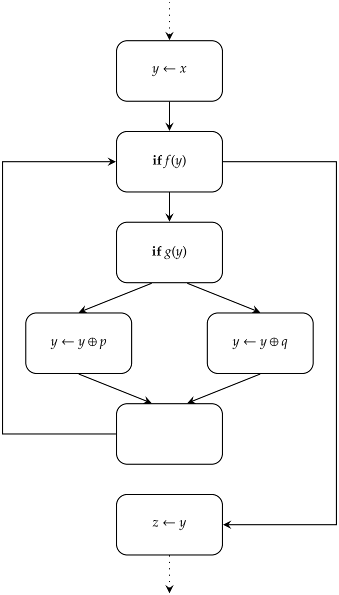
- To analyse a program in SSA form we
  1. start with a normal DFG,
  2. give unique labels to each symbol and insert "magic" ϕ-nodes to support data-flow analysis, then
  3. once the analysis is finished, eliminate ϕ-nodes by pushing a "patch" assignment into parent nodes.

Extra: coping with control-flow



- To analyse a program in SSA form we
  1. start with a normal DFG,
  2. give unique labels to each symbol and insert “magic” ϕ-nodes to support data-flow analysis, then
  3. once the analysis is finished, eliminate ϕ-nodes by pushing a “patch” assignment into parent nodes.

Extra: coping with control-flow



- To analyse a program in SSA form we
  1. start with a normal DFG,
  2. give unique labels to each symbol and insert "magic" ϕ-nodes to support data-flow analysis, then
  3. once the analysis is finished, eliminate ϕ-nodes by pushing a "patch" assignment into parent nodes.
- Idea: it *feels* something similar could be done with mask annotation:
  1. start with a normal DFG,
  2. insert "magic" ψ-nodes to support data-flow analysis, then
  3. once the analysis is finished, eliminate ψ-nodes by pushing a "patch" mask update into parent nodes.