Agenda

1. Revisiting last week’s questions
2. Algebraic Complexity of AES-like S-boxes
3. Boolean Function Constructions
4. Software Optimizations for S-Box
5. 16-Bit Circuit for Multiplicative Inverse Calculation
Questions Answered

How many irreducible and primitive polynomials exist for extension fields $GF((2^n)^m)$?

- $(n, m) = (2, 2) = 18$
- $(n, m) = (2, 3) = 180$
- $(n, m) = (3, 2) = 504$
- $(n, m) = (2, 4) = 1800$
- $(n, m) = (4, 2) = 10800$
- ...
Determining the algebraic complexity

- The AES S-box is a function $S(x) = L(x) \oplus b$, where $L(x)$ is a linear function over $GF(2)$.
- There are many ways to represent $S(x)$ as a polynomial equation:
  - Lagrangian interpolation
  - Polynomial linearization
  - q-ary polynomial deduction
Lagrangian Interpolation

For any function $F : \mathbb{Z}_n \to \mathbb{Z}_n$ with input $x_1, \ldots, x_n$ and output $y_1, \ldots, y_n$, we may find a polynomial representation $P(x)$ as follows:

$$P(x) = \sum_{i=0}^{k-1} P_i(x),$$

where

$$P_i(x) = y_i \prod_{j=1, j \neq i}^{k} \frac{x - x_j}{x_i - x_j}.$$
A Simple Example

Let $F : GF(2^2) \to GF(2^2)$ be a function defined in $GF(2^2)/p(x) = x^2 + x + 1$ by the following map:

- $0 \to 1$
- $1 \to \alpha$
- $\alpha \to \alpha + 1$
- $\alpha + 1 \to 0$

For Lagrangian interpolation, we need polynomials $f_z(x)$ with the property $f_z(x) = 1$ and $f_z(y) = 0$ if $y \neq z$. 
A Simple Example

Start by constructing the polynomial $g(x) = (x - 1)(x - \alpha)(x - (\alpha + 1))$. Note that if $x \in GF(2^2) \setminus \{0\}$, then $g(x) = 0$.

Therefore, we pick $f_0(x) = g(x)/g(0)$, where $g(0) = 1 \cdot \alpha \cdot (\alpha + 1) = 1$.

Thus, $f_0(x) = g(x)$, which makes this very easy. Expanding out $g(x)$ yields:

$$g(x) = (x - 1)(x - \alpha)(x - (\alpha + 1))$$

$$= (x^2 - x - x\alpha + \alpha)(x - (\alpha + 1))$$

$$= x^3 - x^2 - x^2\alpha + x\alpha - x^2\alpha - x\alpha - x\alpha^2 - \alpha^2 + x^2 - x - x\alpha + \alpha = x$$

after reduction with $p(x) = x^2 + x + 1$, of course.
A Simple Example

We may find the other polynomials $f_1(x), f_\alpha(x), f_{\alpha+1}(x)$ by linear substitutions:

$$f_z(x) = f_0(x - z)$$

(A textbook informed me of this fact)
A Simple Example

Now we can do interpolation as follows:

\[ q(x) = F(0)f_0(x) + F(1)f_1(x) + F(\alpha)f_\alpha(x) + F(\alpha + 1)f_{\alpha+1}(x) \]
\[ = x^2(\alpha + 1) + 1 \]

A simple check...

\[ q(\alpha) = (\alpha)^2(\alpha + 1) + 1 = \alpha^3 + \alpha^2 + 1 = \alpha + 1 \]
\[ q(1) = (1)^2(\alpha + 1) + 1 = \alpha \]
\[ q(0) = (0)^2(\alpha + 1) + 1 = 1 \]
\[ q(\alpha + 1) = (\alpha + 1)^2(\alpha + 1) + 1 = \alpha^3 + \alpha + \alpha^2 = 0 \]
Lagrangian Lesson

The method is more symbolic than computational (at first glance), so perhaps there’s a better way to estimate the complexity...
Polynomial Linearization

- Any linear function $A$ over $GF(2^k)$ can be represented as a matrix multiplication.
- Similarly, such functions can be represented by a linearized polynomial:

$$f(\alpha) = \sum_{i=0}^{k-1} \lambda_i \alpha^{2^i}$$

- Solve for $\lambda_i$ by setting up and solving a system of linear equations.
  - Select some $\alpha$, compute $A(\alpha)$ and $\alpha^{2^i}$ for all $0 \leq i \leq k - 1$.
  - Solve for each $\lambda_i$ using Gaussian elimination.
Bounds on Algebraic Expression

The upper bound on the number of terms in an algebraic expression for affine-power functions

\[ F(x) = A(P(x)) \]

in \( GF(2^n) \) is \( n + 1 \)

The forward AES S-box, \( F(X) = L(x^{-1}) = L(x^{254}) \), has 9 terms:

\[ L(x) = \sum_{i=0}^{7} \lambda_i x^{2^i} \]
Increasing the Algebraic Complexity

- **Affine-power-affine functions:** $F(x) = A \circ P \circ A$
  - Increases algebraic complexity without affecting other cryptographic properties (strict avalanche, nonlinearity, differential uniformity, algebraic degree)
  - This increased the algebraic complexity from 9 to 253

- **Gray code augmentation:** $F(x) = L \circ P \circ G$
  - A *gray code* is a binary numeral system where two successive values differ by a single bit
  - $G$ is gray-code conversion from an element $x \in GF(2^n)$ to a corresponding gray-code
  - Conversion process: $y_i = x_{i+1} \oplus x_i$ and $y_n = x_n$

- **Möbius transformation:** $f(z) = \frac{az + b}{cz + d}$, where $a, b, c, d \in GF(2^k)$. 

Christopher A. Wood
General Maiorana-McFarland Construction

- Concatenate small affine functions to form higher-order functions
- (Hopefully) the result is an equally strong Boolean function
- All MM Boolean functions have an annihilator of degree \((n - r + 1)\), where \(r\) is the number of variables of affine functions which are used (concatenated) to construct the function
- As \(r\) decreases the annihilator degree increases, making algebraic attacks easier (it simplifies the equations)
Linear Codes

- A $[n, k, d]$-code (binary code) is a subspace of $\mathbb{F}_2^n = GF(2)^n$
  - $n$ is the length, $k$ is the rank, $d$ is the minimum (Hamming) distance between each codeword in the subspace

- The vectors of a binary linear code are called the codewords

- As a subspace, there exists a basis $B$ for the code, which is often represented as a generator matrix $G$

- Many codes of cryptographic interest: Hamming, Walsh-Hadamard, ...
Candidate Codes

- **Hamming Code**: a special type of binary \([n, k, 3]\) code
  - Mainly used for error detection/correction, but we can use it for resilient BF constructions

- **Hadamard Code**: a special type of binary \([2^k, k, 2^{k-1}]\) code
Construction Idea for $t$-resilient

- Let $f_1, \ldots, f_{2^n-r}$ be $2^{n-r}$ affine Boolean functions of length $2^r$ (i.e. the truth table has $2^r$ entries)
- Concatenating $f_1, \ldots, f_{2^n-r}$ yeilds a string of length $2^n$
- Let $g(x_n, \ldots, x_{r+1})$ be a nonlinear function and let $h(x_r, \ldots, x_1)$ be a linear (affine) function, and let $f(x_n, \ldots, x_1) = g(x_n, \ldots, x_{r+1}) \oplus h(x_r, \ldots, x_1)$

*Note: all Boolean functions are $(t+1)$ degenerate, for reasons that are discussed in the paper :-)*
Construction Idea for $t$-resilient

- Select a $[n = u, k = m, d = t + 1]$ code and construct a $(2^m - 1) \times m$ matrix with codewords from $C$ s.t.
  \[ \{ c_1 D_{i,1} \oplus \cdots \oplus c_m D_{i,m} : i \leq 1 \leq 2^m - 1 \} = C \setminus \{ \bar{0} \}. \]
  Let $L(C)$ be a $(2^m - 1) \times m$ matrix whose entries are $u$-variable functions defined by $L_{i,j}(x_1, \ldots, x_u))$

- Define an $(p, m)$ S-box with component functions $G_1, \ldots, G_m$, and let $L(C, k, l)$ be an $(l - k + 1) \times m$ matrix whose $i, j$th entry is
  \[ G_j(y_1, \ldots, y_p) \oplus L_{k+i-1,j}(x_1, \ldots, x_u). \]
Construction Continued

- If $l - k + 1 = 2^r$ then $G \oplus L(C, k, l)$ is an $(r + p + u, m)$ S-box:

$$F_j(z_1, \ldots, z_r, y_1, \ldots, y_p, x_1, \ldots, x_u) = G_j(y_1, \ldots, y_p) \oplus L_{k+i-1,j}(x_1, \ldots, x_u)$$

- Goal: Let $m = 16$, find other parameters that make the construction "work"

- Need to select good $(p, 16)$ S-boxes $G_1, \ldots, G_m$ and find a good $[n, 16, t + 1]$ code word
Software Optimizations for S-Box

- Extended Euclidean Algorithm - Straightforward
- Binary Extended Euclidean Algorithm - Optimized version of EEA for fields of characteristic 2
- Normal basis conversion with Fermat’s Theorem - Two matrix multiplications with some shifting and multiplying
- Almost Inverse Algorithm - Compute $A^{-1}x^k \mod f(x)$ and then reduce by $x^k$
- Bitsliced implementation - Carnright investigates this technique with his normal basis optimizations
- LUTs - Not a goal, but always an option...
Software Optimizations for S-Box - Metrics

These can be captured with gprof for different platforms...

- Extended Euclidean Algorithm - TODO
- Binary Extended Euclidean Algorithm - TODO
- Normal basis conversion with Fermat’s Theorem - TODO
- Almost Inverse Algorithm - TODO
- Bitsliced implementation - TODO
- LUTs - ;-}
Complexity of Finite Field Multipliers

- Claim: for small fields (e.g. $GF(2^k)$, $k \leq 32$) the arithmetic procedures for software implementations are not affected by the field polynomial.
  - Advanced algorithms such as the “comb” multiplier target fields where single elements cannot fit within a single word
- This is not true for hardware...
  - If we’re going for area optimized designs, we want serial modules, otherwise we want parallel modules
  - Some bases yield more efficient arithmetic operations than others
  - This leads us to Optimal Normal Bases
Inverse by Fermat’s Theorem

By Fermat’s Theorem, $\alpha^{-1} \equiv \alpha^{2^k-2}$

$$2^{m-2} = 2 + 2^2 + 2^3 + \cdots + 2^{m-1}$$

This leads us to a simple square and multiply algorithm...

$$\alpha^{-1} = \alpha^2 \cdot \alpha^{2^2} \cdot \alpha^{2^3} \cdots \alpha^{2^{m-1}}$$

In a normal basis the cycle complexity is $O(k)$ for computing the successive powers of $\alpha$, but the area complexity depends on the type of multiplier used (e.g. using a ONB Type II basis one can implement a parallel multiplier with $1.5(k^2 - k)$ XOR gates [1])
Inverse by Composite Field Computation

\[(bx + c)^{-1} = b(b^2B + bcA + c^2)^{-1}x + (c + bA)(b^2B + bcA + c^2)^{-1}\]

with \(A = 1\) and \(B = \lambda\)
Inverse by Composite Field Computation (continued)

5-stage pipeline design
Optimal Pipeline Selection Strategy (for FPGAs)

Algorithm 1 Pipeline Optimization Strategy

1:  \( E_c = \frac{\text{Throughput (Mbits/s)}}{\text{Area}} \)
2:  \( \text{Opt} \leftarrow \text{False} \)
3:  \textbf{while} \( \text{Opt} = \text{False} \) \textbf{do} 
4:      Remove the pipeline state that contributes the lowest frequency reduction
5:  Reimplement and resynthesize the design
6:  \( E_n = \frac{\text{Throughput (Mbits/s)}}{\text{Area}} \)
7:  \textbf{if} \( E_c > E_n \) \textbf{then}
8:      \( \text{Opt} = \text{True} \)
9:  \textbf{end if}
10: \textbf{end while}
Inverse by Composite Field Computation (continued)

The next step is to synthesize the design and gather hardware metrics.

- LUT count (FPGA - captured with Xilinx tools)
- Register count (FPGA - captured with Xilinx tools)
- Slice count (FPGA - captured with Xilinx tools)
- Throughput in cycles/byte (FPGA - captured with Xilinx tools)
- Power consumption (ASIC - captured with Synopsys) :-(
References

Action Items (perhaps overly ambitious...)

- Optimize Galois field software for more efficient calculation of polynomials and transformation matrices
- Finish composite field decomposition chapter
- Polynomial and normal basis conversion code and preparation for OSG execution
- Literature survey of S-box constructions and code for estimating algebraic complexity
- Complete the exhaustive list of all polynomials $P(x)$, $Q(y)$, and $R(z)$ and the corresponding list of all transformation matrices (using OSG!)
- Hardware metrics of regular and non-pipelined 16-bit inverse of composite field inverse
- Implement Carnright’s normal basis S-box
- (16, 16)-Boolean function code using the prescribed approach