Implementing Authenticated Encryption Algorithm MK-3 on FPGA

Gordon Werner, Steven Farris, Alan Kaminsky, Michael Kurdziel, Marcin Lukowiak, and Stanisław Radziszowski

Abstract—Authenticated encryption (AE) algorithms provide both data security and integrity. While a number of AE algorithms exist, they can be inefficient and difficult to use. Recent efforts have focused on the development of secure, efficient and easy to use AE algorithms. MK-3 is one such algorithm, developed through a joint effort between Rochester Institute of Technology (RIT) and Harris Corporation. It uses the duplex construction, which builds on the sponge primitive popularized by Keccak, the SHA-3 competition winner. MK-3 is intended for hardware implementations with a novelty being the use of 16-bit substitution boxes. This paper presents the first fully parallel hardware implementation of MK-3 using Field Programmable Gate Array (FPGA). We also lay the groundwork for future design optimizations.

I. INTRODUCTION

As more data moves into the digital realm, the need for security becomes increasingly necessary. Many people’s sensitive information is being stored and used digitally. Adequate security measures must be taken to prevent malicious attacks such as identity theft or fraudulent purchases. Cryptographic encryption is a commonly used method to obfuscate data such that the original value cannot be obtained without knowing the secret key that was used by the encryption function. Authenticated encryption algorithms add an additional level of security by providing both data encryption and authentication for plaintext and additional authenticated data (AAD) [1]. Data is authenticated through the use of a Message Authentication Code (MAC), or tag T. Decryption produces an identical tag from the ciphertext-AAD pair. The algorithm is defined such that it is not feasible to produce the same tag and ciphertext from different message or AAD. The most commonly used AE scheme is the AES in Galois Counter Mode (AES-GCM) [2].

Current AE algorithms can be difficult and inefficient to use as they usually require 2 passes over each block of data. This inspired the Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) which aims to provide a portfolio of AE algorithms for public use [3], [4].

In May 2014 a research group from RIT and Harris Corporation proposed a new authenticated encryption algorithm MK-3. The algorithm was designed with hardware implementation and efficiency in mind [5]. MK-3 uses a permutation function wrapped in a duplexed sponge construction. This cryptographic primitive, popularized by the SHA-3 winner Keccak, can be modified to allow for other functions including AE. The CAESAR features a number of algorithms utilizing the duplexed sponge construction for AE. MK-3 makes use of a substitution box (S-box) similar in design to those used by the AES but larger. The other steps of this permutation require bitwise xor and and operations as well as a bitwise reordering of the state. These design characteristics make MK-3 ideal for hardware implementation. Here, we present the first hardware implementation of the MK-3 algorithm.

The rest of this paper is organized as follows: Section 3 describes MK-3, detailing its permutation as well as the duplexed sponge construction. Section 4 discusses the hardware implementation of the algorithm and the resource requirements. Section 5 explores future work that can be done to improve performance, and Section 6 concludes the paper.

II. MK-3 AE ALGORITHM

The MK-3 algorithm is a single pass algorithm designed with hardware implementation in mind. Its computations consist mainly of bitwise xor, shift and and operations. The core of MK-3 is a permutation function f comprised of 4 steps. Operations are performed on a 512-bit state split into 16 bit words. The permutation function is used as the core of the duplexed sponge construction [6].

A. The Duplexed Sponge

The sponge is a relatively new cryptographic primitive made popular by Keccak, the winner of the SHA-3 competition [7]. The sponge is unique in that it can be configured to allow for a variety of cryptographic uses. The sponge construction operates using iterations of a transformation or permutation function f. The sponge has an internal state S comprised of b bits which is split into a rate r and a capacity c, such that b = r + c. A basic sponge construction consists of 2 stages. Data is “absorbed” into the sponge by passing it through the underlying function f in r length blocks. It is then "squeezed" out, providing output of variable length specified in the application. The duplex construction is a slight modification of the sponge which maintains state between calls while absorbing and squeezing the data during each iteration [6]. Data is still operated on in blocks, however output is produced after each call to f. The length of the output cannot exceed the rate r. Figure 1 illustrates this approach.
The duplex construction provides advantages for AE use as it is single-key and single-pass and has the capability to provide intermediate MACs. AE can be achieved through successive duplexing calls with the key, initialization vector (IV), AAD and plaintext. Figure 1 shows a typical execution flow of the duplexed sponge as it does AE.

B. MK-3 Permutation

The core of MK-3 is its permutation function. A description of the permutation is given in Algorithm 1 and an illustration of a round operation can be found in Figure 2.

Algorithm 1 MK3 Permutation

1: procedure STATE INITIALIZATION(S, P)  
2: \[ S \leftarrow P \oplus S \] \( \Rightarrow \) 512 bit state with 16 bit words  
3: \[ r \leftarrow 0 \]  
4: procedure xSTAR(x) \( \Rightarrow \) Mix State word \( \begin{array}{c} x \ m \leftarrow x_{15} \\ \end{array} \)  
5: \( \text{\large Store MSB of } x \ x \leftarrow (x \ll 1) \oplus 0000000000m0m0m0m \)  
6: function \( f(S) \) \( \Rightarrow \) Permute State \( S \)  
7: \[ \text{State Initialization}(S, P) \]  
8: for \( r < 16 \) do  
9: for \( 0 \leq i < 32 \) do  
10: \[ S[i] \leftarrow SBox[i] \]  
11: \( \text{\large for } 0 \leq b < 512 \text{ do} \)  
12: \[ \text{new}_b \leftarrow (31b + 15) \mod 512 \]  
13: \( S_{\text{new}_b} \leftarrow S_b \)  
14: \( \text{\large for } 0 \leq i < 32, \ i += 2 \text{ do} \)  
15: \[ S[i] \leftarrow S[i] \oplus xSTAR(S[i + 1]) \]  
16: \[ S[i + 1] \leftarrow S[i] \oplus xSTAR(S[i + 1]) \oplus S[i + 1] \]  
17: \[ S \leftarrow S \oplus RC[r] \] \( \Rightarrow \) Add Round Constant

The step in line 10 is the only non-linear one, which passes each 16-bit state word through an S-box. The S-box returns the multiplicative inverse in \( GF(2^{16}) \) of the input multiplied by a transformation matrix and added to an offset vector. After substitution, the data is permuted by passing it to the affine function \( \pi(x) = 31x + 15 \mod 512 \). The data is then mixed, providing local diffusion. Neighboring pairs of words are mixed together to produce two new words which are passed onto the last step where a round constant is added to the state.

MK-3 allows for keys of length 128 and 256 bits; in both cases the rate \( r \) is kept at 128 bits. This allows for simple transitioning between key sizes. This construction is shown to be sufficiently secure against generic attacks. The security margin of a keyed sponge was shown by Jovanovic et al. [8] to be: \[ \min(2^{(r+c)/2}, 2^{r}, 2^{|K|}) \].

The only operational difference in MK-3 between key sizes is the number of rounds. Using a 128 bit key requires 10 while 256 bit keys require 16. The number of rounds is determined by calculating the linear and differential complexity over a number of rounds until it exceeds the generic security of the algorithm [5]. Differential and linear cryptanalysis of MK-3 showed that 6 and 12 rounds are required to obtain a level of security above \( 2^{128} \) and \( 2^{256} \), respectively. The additional 4 rounds for each key size are added to further increase the security margin.

1) Substitution Step: The substitution box provides a non-linear operation to a cryptographic function. Generally, a S-box takes in \( x \) bits of input and produces a value \( y \) bits in length. The S-box used by AES computes the multiplicative inverse of its input element \( \alpha \) in the Galois field \( GF(2^8) \), then passes it through an affine function as described in equation (1). Here \( \alpha \) is treated as a bit vector, and the result of a matrix \( M \) multiplication is added to a constant \( c \).

\[ S[\alpha] \leftarrow M \cdot \alpha^{-1} + c \quad (1) \]

MK-3 is the first algorithm to our knowledge to employ larger S-boxes than 8-bits [5]. The 16-bit S-box constructions were presented by Wood [9], and like in the AES they find the inverse of the input element and pass it through an affine function.
transformation as follows:

\[
\begin{pmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{pmatrix}^{-1}
\begin{pmatrix}
0 \\
1 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0
\end{pmatrix}
\]

A common way of finding the multiplicative inverse is to use the Extended Euclidean Algorithm (EEA). The EEA conducts a series of divisions to find the inverse of an element in a field. This is a fine solution for software, however it is very complex to implement in hardware especially over 16 bit elements. To reduce complexity, a non-standard field construction is used to directly solve for the inverse. Our field representation defines its operations so that they are more conducive to hardware.

2) Permutation Step: After passing each word of the state through an S-box, the state bits are permuted, \( \pi(x) = 31x + 15 \mod 512 \) before being processed as words by the mixing function. In hardware this is a reordering of wires based on \( \pi(x) \).

3) Mixing Step: Mixing functions operate over pairs of 16 bit state words after permutation. The mixer’s operation computes the multiplication of a vector containing two state words and a \( 2 \times 2 \) matrix. The operation is done in \( GF(2^{16}) \) with irreducible polynomial \( q(x) = x^{16} + x^5 + x^3 + x^2 + 1 \).

\[
\begin{bmatrix}
A' \\
B'
\end{bmatrix} =
\begin{bmatrix}
1 & x \\
x & x + 1
\end{bmatrix}
\begin{bmatrix}
A \\
B
\end{bmatrix}
\tag{2}
\]

Carrying out the matrix multiplications provides the following functions for \( A' \) and \( B' \):

\[
A' = A \oplus B \cdot x \\
B' = A \cdot x \oplus B \cdot x \oplus B 
\tag{3}
\]

After the mix step, the entire state has a 512-bit constant added to it. The constant values are derived from Keccak, the SHA-3 competition winner. Each round constant is the output of Keccak calculated over the ascii value of the round number, \( RC_i = \text{Keccak-512(ASCII(i))} \) [5].

III. IMPLEMENTATION DETAILS

MK-3 was implemented using the Artix-7 XC7A100T FPGA using a fully parallel datapath. All 128 bits of the message are passed into the permutation function, where the entire 512 bit state is processed at a rate of one round per clock cycle. For each round, data is passed from the 512 bit state register through all steps of the round. Bit reordering in the permutation step is trivial to implement on a FPGA. Adding the round constant is also straightforward; the constants are stored in registers and added to the state.

A. S-Box Galois Field Construction

Our S-box finds the multiplicative inverse of its input in the Galois Field \( GF(2^{16}) \). Most commonly, the polynomial basis \([1, x, ... x^{k-1}]\) is used, with each element represented by a degree \( k \) polynomial, \( a \in GF(2^k) = a_0 + a_1x + a_2x^2 + \ldots + a_{k-1}x^{k-1} \). All operations are performed modulo an irreducible polynomial \( q(x) \). This can be difficult to implement in hardware, leading us to use the normal basis. For every Galois field, there is at least one normal element \( \theta \) such that \( \forall a \in GF(p^k), a = a_0 \theta + a_1 \theta^2 + \ldots + a_{k-1} \theta^{k-1} \). Using this element, the normal basis can be defined as \([\theta^2, \theta^3, ..., \theta^{2^k-1}]\). Normal representation of elements is more conducive to hardware implementations as squaring is a right cyclic shift, while multiplication is a series of their bitwise combinations.

The composite construction of a Galois field uses power extensions on an already extended field. If \( k = \ell m \), with \( \ell, m \in \mathbb{Z} \) then \( GF(2^k) \) can be defined as \( GF(2^\ell)^m \), an \( m \) degree extension of \( GF(2^\ell) \). As in any Galois field, operations are done on elements of the base field, \( GF(2^\ell) \). The composite field requires only \( m \) independent vectors as opposed to the \( k \) of a standard basis. By using such a construction \( GF(\ell((2^2)^2)^2) \) in our S-box operations in a field of order 16 behave as though it were a degree 2 extension field. While any tower construction can be used, our design uses degree 2 extensions of \( GF(2) \) as they require the fewest number of 2 input logic gates [9].

Our S-box design solves the inverse directly rather than using EEA. A non-zero element in \( GF(p^\ell), \epsilon = \delta_1 x + \delta_2 \) in a Galois field by definition has an inverse \( \epsilon^{-1} = \delta_1 x + \delta_4 \) such that \( \epsilon_1 \cdot \epsilon_2 = 1 \). Thus \( 1 = (\delta_1 x + \delta_2)(\delta_3 x + \delta_4) \). This equation can be manipulated to solve for \( \delta_3 \) and \( \delta_4 \) given only the element \( \epsilon \). This approach can be applied to other composite fields as well. Solving for the inverse in \( GF(2^2)^2 \) provides an equation that can be extended recursively by powers of two. The full reduction can be found in Wood’s thesis [10].

Figure 3 shows generalized inversion and multiplication operations for any power 2 extension to a composite field with the normal basis representation. Power 2 extensions of a Galois field have irreducible polynomials of the form...
\( x^2 + x + \Sigma \). \( \Sigma \) is determined such that the field polynomial remains irreducible. In the base field \( GF(2^2) \) there is only one irreducible polynomial, \( x^2 + x + 1 \). With further composite extensions, \( \Sigma \) is some element of the current extension’s base field that does not violate irreducibility. Wood chose the values for \( \Sigma \) that minimize the additional bitwise computations required for multiplication [9].

By using only degree two extensions, the order of operation is unchanged except for the length of the operands. Inversion is similarly generalizable [9]. A general illustration of our inversion hardware is shown in Figure 4; the component follows the same operations as those detailed in Figure 3. Each extension \( GF(2^2)^2 \) is isomorphic to \( GF(2^k), k = 2\ell \). The inversion circuit takes in a \( k \) bit input and interprets it as two \( \ell \) bit elements. When a multiplication or inversion occurs within the circuit, a component is instantiated isomorphic to the current base field, \( GF(2^\ell) \). These components then split their inputs into two \( \ell \) elements and conduct the necessary computations. Addition is accomplished using a \( \ell \)-input \text{xor} gate which is simple enough that no reductions to sub-fields are required.

Field reductions continue until \( GF(2^2) \) components are instantiated. These produce 2 \( GF(2^2) \) elements as output that are then treated as a single \( GF(2^2)^2 \) element which are then passed up in pairs as a \( GF((2^2)^2)^2 \) element, and so on. A multiplication circuit is constructed in the same manner, using multipliers from the base field to recursively complete the calculation. This can be extended any number of times; \( GF((2^2)^2)^2 \) inversion circuits make use of \( GF(2^2)^2 \) components, just as \( GF(((2^2)^2)^2)^2 \) components make use of \( GF((2^2)^2)^2 \) inverter and multipliers. In our design, the circuit is extended until a composite construction isomorphic to \( GF(2^{16}) \) is completed.

By Fermat’s Little Theorem, the inverse of an element \( a \in GF(2^2) \) is \( a^{-1} = a^2 \), a simple bit swap in the normal basis. Multiplication follows the example described in Figure 3 and consists of a number of bitwise and \text{xor} operations between various pairs of bits as shown in Figure 6.

Using the normal basis representation requires a change of basis computation before and after the inverse operation as the rest of MK-3 operates in the field \( GF(2^{16}) \) using the polynomial basis. Once the inverse \( x^{-1} \) is obtained, another matrix multiplication is required followed by a 16-bit \text{xor} to compute the affine transformation.

Finally, the S-box acts as: \( S(a) = M \cdot a^{-1} \oplus x^{14} + x^7 + x^5 + x^4 + x^2 + x + 1 \)

B. Mixer Implementation

Figure 7 illustrates the mix step of the algorithm from a hardware perspective. Pairs of state words \( (A, B) \) are processed to produce \( (A', B') \) based on equation (2). The mixer requires only additions and multiplication in \( GF(2^{16}) \) by the element \( x \). Using the Polynomial basis, multiplication by \( x \) in hardware can be implemented by conducting a single left shift. To account for potential reduction by the irreducible polynomial, a rotational left shift is used instead. The most significant bit (MSB) of the input word is also \text{xor}-ed with the bits associated with the powers of \( q(x) \) after the rotational shift. Multiplication by \( x \) is realized in hardware through the \( x^* \) component. An illustration of the \( x^* \) function is shown in figure 8.
TABLE I
MK-3 HARDWARE IMPLEMENTATION RESULTS FOR DIFFERENT S-BOX CONSTRUCTIONS

<table>
<thead>
<tr>
<th>S-box Construction</th>
<th>Multi. LUT</th>
<th>Inv. LUT</th>
<th>Slices</th>
<th>LUTs</th>
<th>FFs</th>
<th>Frequency(MHz)</th>
<th>Throughput(Mb/s)</th>
<th>Throughput/slice(Kb/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(GF((2^5)^2)^2)</td>
<td>none</td>
<td>none</td>
<td>5742</td>
<td>14966</td>
<td>648</td>
<td>74.2</td>
<td>527.7</td>
<td>141.0</td>
</tr>
<tr>
<td>(GF((2^3)^2)^2)</td>
<td>none</td>
<td>(GF(2^4))</td>
<td>5645</td>
<td>14580</td>
<td>648</td>
<td>74.3</td>
<td>528.3</td>
<td>144.9</td>
</tr>
<tr>
<td>(GF((2^3)^2)^2)</td>
<td>(GF(2^4))</td>
<td>(GF(2^4))</td>
<td>6578</td>
<td>26310</td>
<td>648</td>
<td>58.3</td>
<td>414.6</td>
<td>63.0</td>
</tr>
<tr>
<td>(GF((2^2)^2)^2)</td>
<td>none</td>
<td>(GF(2^4))</td>
<td>3230</td>
<td>12920</td>
<td>648</td>
<td>88.0</td>
<td>626.0</td>
<td>193.8</td>
</tr>
<tr>
<td>(GF((2^2)^2)^2)</td>
<td>(GF(2^4))</td>
<td>(GF(2^4))</td>
<td>4834</td>
<td>19333</td>
<td>648</td>
<td>73.8</td>
<td>525.2</td>
<td>108.6</td>
</tr>
</tbody>
</table>

Fig. 7. Illustration of mix step hardware

Fig. 8. Illustration of \(x^*\) function

C. Implementation Results

MK-3 was implemented in hardware using the Artix-7 FPGA. Utilizing serial communication, input and output data could be transferred from the MK-3 core to a processor executing a Matlab script. All results are based on this implementation.

In hardware, the S-box construction accounts for most of the area, requiring 427 LUTs to implement all of its logic. To shrink the design, LUTs can be utilized at the intermediate extensions to return the multiplicative inverse without completing the operations of the lower extensions. Table I shows implementation results of MK-3 with LUT utilization at various levels of the tower construction. Implementations of MK-3 were made at each construction utilizing LUTs for inversion only, as well as both inversion and multiplication. This was done to explore the impact the full reduction to \(GF(2^2)\) has on the system. All throughput \(T\) calculations assume constant operation of data with no delays between blocks. Throughput for MK-3 is calculated as

\[
T = \frac{128 \times \text{frequency}}{\text{Cycles per block operation}}
\]

The S-box field construction utilized was chosen to minimize the number of 2 input gates [9]. Each S-box requires 1238 \text{ } xor and 144 \text{ } and operations. When compared to the composite S-boxes of AES which can be optimized to require only 68 \text{ } xor and 36 \text{ } and gates [9] the exponential impact of doubling the field size can be seen. For FPGA implementation this may not be optimal. FPGAs utilize 5 and 6 input LUTs to realize their functionality. Reducing all logic down to 2 input logic operations is unnecessary in this environment. As seen in Table I using a LUT to return the inverse of an 8-bit element rather than traversing down and back up the tower allows for nearly a 100 Mb/s increase in throughput. Interestingly, using LUTs for multiplication between two \(GF(2^4)\) elements causes a drop in performance. The area impact is expected as there are 12 \(GF(2^4)\) multiplier components instantiated in each S-box compared to the single inverter component. Each \(GF(2^4)\) multiplier requires \(2^7\) bytes of storage to cover all pairs of operands.

IV. FUTURE WORK

We expect that further enhancements to the design can be made to reduce area and increase efficiency. Currently in our design, the round constants are stored in on chip registers. However, if a generated BRAM core were to be used to store the data, the overall footprint could be reduced. The S-box construction can also be analyzed further. While \(GF(((2^2)^2)^2)^2\) was used for its computational simplicity, any other tower constructions could be used. While reducing operations to 2 input logic is ideal for ASIC design, FPGA devices do not necessarily benefit from such a reduction in logic. FPGAs make use of 5 or 6 input LUTs which can be used to quickly compute the results of larger logic blocks. Using LUTs at intermediate points in inversion showed a boost in performance and area. Reconstructing the field to have a degree 4 or 8 base extension could provide better performance.

Resistance to side channel attacks has not been considered by this paper, and it is an area for future research. With the rise of hardware and side channel attacks, it would be prudent to analyze MK-3 as well as the S-box used to ensure there are no vulnerabilities at the hardware level. Algebraic and differential analysis does not take this into account as it focuses on data confusion and diffusion through the algorithm; so while MK-3 and other algorithms are cryptographically secure, without proper countermeasures they are all susceptible to side channel attacks.
V. Conclusion

The first hardware implementation of the MK-3 authenticated encryption algorithm has been developed and presented. Due to its novel 16-bit S-box MK-3 requires more area, however allows for a higher throughput than initial implementations of AES on FPGA. We expect that with further optimization the speed of the algorithm can be increased, while the hardware footprint can be reduced.

References


