Small Scale AES Toolbox: Algebraic and Propositional Formulas, Circuit-Implementations and Fault Equations

Maël Gay[2], Jan Burchard[1], Jan Horáček[2], Ange-Salomé Messeng Ekossono[2], Tobias Schubert[1], Bernd Becker[1], Martin Kreuzer[2], Ilia Polian[2]

[1] Albert-Ludwigs-University Freiburg
Georges-Köhler-Allee 051, 79110 Freiburg, Germany
[2] University of Passau
Innstraße 33 & 43, 94032 Passau, Germany

Abstract. Cryptography is one of the key technologies ensuring security in the digital domain. As such, its primitives and implementations have been extensively analyzed both from a theoretical, cryptoanalytical perspective, as well as regarding their capabilities to remain secure in the face of various attacks.
One of the most common ciphers, the Advanced Encryption Standard (AES) (thus far) appears to be secure in the absence of an active attacker. To allow for the testing and development of new attacks or countermeasures a small scale version of the AES with a variable number of rounds, number of rows, number of columns and data word size, and a complexity ranging from trivial up to the original AES was developed.
In this paper we present a collection of various implementations of the relevant small scale AES versions based on hardware (VHDL and gate-level), algebraic representations (Sage and CoCoA) and their translations into propositional formulas (in CNF). Additionally, we present fault attack equations for each version.
Having all these resources available in a single and well structured package allows researchers to combine these different sources of information which might reveal new patterns or solving strategies. Additionally, the fine granularity of difficulty between the different small scale AES versions allows for the assessment of new attacks or the comparison of different attacks.

1 Introduction

The Small Scale AES cipher is a family of smaller variants of the AES [1] and was introduced in [2]. The basic cryptographic functions are similar those used by AES, consisting of the usual *SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey* operations. The reduction in complexity and size is achieved by lowering the block size from the original 16 bytes (usually arranged as a 4×4 matrix) to arbitrary smaller matrix sizes as well as a choice between 4 and 8-bit per entry. Unlike other small or lightweight ciphers, the small scale AES variants are not meant to be cryptographically secure. Instead, their main field of application is research: The original AES has, thus far, proven to be secure in the absence of an active attacker. Hence, developing and testing possible attacks requires a simplified cipher which reduces the required computation and helps gauge these new attacks.

To the authors knowledge, the concept of the small scale AES has not been transformed into real hardware until now. With this paper we introduce a hardware implementation in VHDL which can, for example, be used in combination with a FPGA. In addition, we also provide a collection of formulas which allow for a theoretical analysis of the cipher using algebraic or propositional solvers.

In detail we present the following contribution for each small scale AES variant:

- Hardware implementations in behavioral VHDL as well as on gate level (Section 2).
- Fault attack equations which can be used to calculate the secret key with only one or two fault injections (Section 3).
- Algebraic representations for the computer algebra software Sage [3] as well as CoCoa [4] (Section 4).
- Propositional formulas derived from the hardware implementations and the algebraic representations (Section 5).

All files are freely available for download (link at the end of the paper).

2 Hardware Implementations

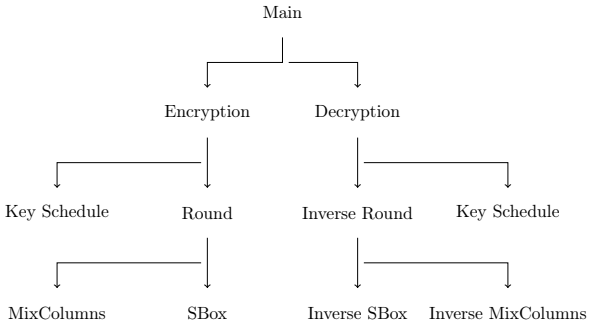
2.1 VHDL

The small scale AES is an entirely parameterizable cipher. In the definition of the small scale AES, it is possible to choose the size of words (4 or 8-bits), the number of rows (R) and columns (C) of the block matrix (1, 2 or 4), the number of rounds (1 to 10) and the presence or absence of a *MixColumns* in the last round. We identify the small scale AES versions by the tuple RC.

All different operations, such as *MixColumns* or *ShiftRows*, are different depending on these parameters. Hence, the choice was made to divide the implementation into 18 different implementations, related to those parameters.

The number of rounds and the presence of the last *MixColumns* can be specified directly in the implementation as generic values (set by default to 10 and 0, respectively). As inputs the circuits require the data to process, the key and the mode (encryption or decryption). The produced output is the processed data (cipher or plaintext).

The implementation was split according to the number of rows and columns but it is also divided into components corresponding to each operation as follows.



For each small scale AES variant we created a VHDL implementation with generic inputs for the number of rounds and whether to omit the last *MixColumns* operation (as in full-scale AES). Generally, each operation is implemented as a module. Hence, different realizations could easily be tested and compared. The exception from this rule are the *ShiftRows* and *ShiftRows* operations which are performed directly in the round function, as they are simple operations.

2.2 Gate Level

The gate level representation is obtained by mapping the VHDL implementation to the 45NM NanGate cell library [5]. The result is a combinational circuit which performs the encryption or decryption operation and could be implemented directly on any hardware.

3 Fault Attack Equations

A differential fault analysis of the AES using a single fault was introduced in [6]. We generalize these results to each instance of the Small Scale AES.

In the formulas we use the following notation: δ_i is the XOR difference of the words at the chosen spot. $S()$ and $S^{-1}()$ are respectively the SBox and the inverted SBox functions. x_i and x'_i stand for the correct cipher and the faulty cipher, k_i is the key.

The number of rows and columns affects the way the fault will propagate through the last rounds. This leads to some differences on the fault equations between each instance of the Small Scale AES which can be classified into three different groups: the number of rows is inferior to the number of columns, there are equally many rows and columns or the numbers of rows is superior to the number of columns. The next section provides an example for each group.

In addition, the fault equations also differ when the *MixColumns* operation in the last round is omitted.

3.1 Equations Without the Last Mix Columns

The encryption process is traced back until right after the last *MixColumns* in order to get the fault equations. As previously discussed, the equations differ depending on the relation between the number of rows and columns.

The obtained equations are processed by trying all possible δ_i and k_i, as shown in [6], until the secret key is obtained.

$\boldsymbol{R < C}$ The propagation of the fault can be seen in Figure 1.

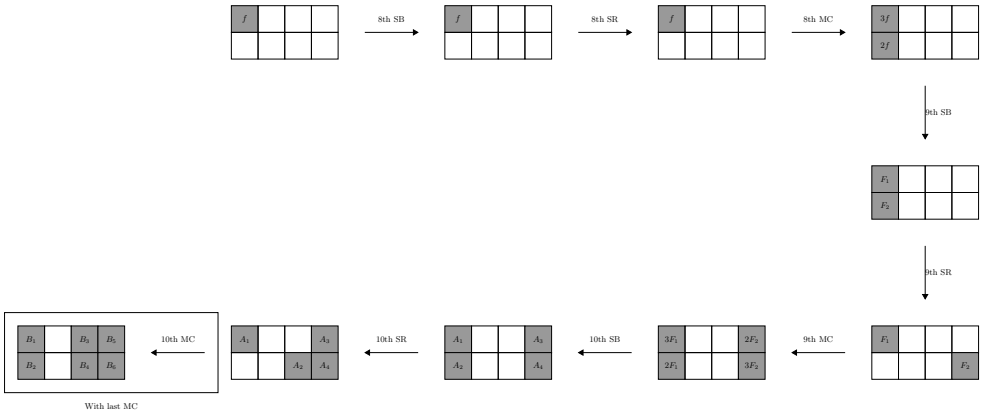


Fig. 1. Fault propagation when the number of rows is lower than the number of columns, exemplary for RC = 24. The fault does not propagate sufficiently.

The fault does not propagate sufficiently to recover the whole key in the end. Hence, a second fault injection on the fifth element of the state matrix is required. This can occur either at the same time or on a subsequent encryption. For $RC = 24$ the following equations are obtained:

$$
\begin{aligned}
3\delta_1 &= S^{-1}(x_1 \oplus k_1) \oplus S^{-1}(x'_1 \oplus k_1) \\
2\delta_1 &= S^{-1}(x_8 \oplus k_8) \oplus S^{-1}(x'_8 \oplus k_8) \\
2\delta_2 &= S^{-1}(x_7 \oplus k_7) \oplus S^{-1}(x'_7 \oplus k_7) \\
3\delta_2 &= S^{-1}(x_6 \oplus k_6) \oplus S^{-1}(x'_6 \oplus k_6)
\end{aligned}
\qquad
\begin{aligned}
3\delta_3 &= S^{-1}(x_5 \oplus k_5) \oplus S^{-1}(x'_5 \oplus k_5) \\
2\delta_3 &= S^{-1}(x_4 \oplus k_4) \oplus S^{-1}(x'_4 \oplus k_4) \\
2\delta_4 &= S^{-1}(x_3 \oplus k_3) \oplus S^{-1}(x'_3 \oplus k_3) \\
3\delta_4 &= S^{-1}(x_2 \oplus k_2) \oplus S^{-1}(x'_2 \oplus k_2)
\end{aligned}
$$

$\boldsymbol{R = C}$ The fault is fully propagated as shown in Figure 2.

In the example of $RC = 44$ the following equations are obtained:

$$
\begin{aligned}
2\delta_1 &= S^{-1}(x_1 \oplus k_1) \oplus S^{-1}(x'_1 \oplus k_1) \\
\delta_1 &= S^{-1}(x_{14} \oplus k_{14}) \oplus S^{-1}(x'_{14} \oplus k_{14}) \\
\delta_1 &= S^{-1}(x_{11} \oplus k_{11}) \oplus S^{-1}(x'_{11} \oplus k_{11}) \\
3\delta_1 &= S^{-1}(x_8 \oplus k_8) \oplus S^{-1}(x'_8 \oplus k_8)
\end{aligned}
$$

$\boldsymbol{R > C}$ For this case the fault propagation is shown in Figure 3.

There is too much fault propagation, which results in more complex equations. One way to address this issue is to inject the fault one round later (for example on the ninth round instead of the eighth). If there are only two columns, then two fault injections are needed (simultaneously or not), as in the following example, where $RC = 42$.

$$
\begin{aligned}
2\delta_1 &= S^{-1}(x_1 \oplus k_1) \oplus S^{-1}(x'_1 \oplus k_1) \\
\delta_1 &= S^{-1}(x_6 \oplus k_6) \oplus S^{-1}(x'_6 \oplus k_6) \\
\delta_1 &= S^{-1}(x_3 \oplus k_3) \oplus S^{-1}(x'_3 \oplus k_3) \\
3\delta_1 &= S^{-1}(x_8 \oplus k_8) \oplus S^{-1}(x'_8 \oplus k_8)
\end{aligned}
\qquad
\begin{aligned}
2\delta_2 &= S^{-1}(x_5 \oplus k_5) \oplus S^{-1}(x'_5 \oplus k_5) \\
\delta_2 &= S^{-1}(x_2 \oplus k_2) \oplus S^{-1}(x'_2 \oplus k_2) \\
\delta_2 &= S^{-1}(x_7 \oplus k_7) \oplus S^{-1}(x'_7 \oplus k_7) \\
3\delta_2 &= S^{-1}(x_4 \oplus k_4) \oplus S^{-1}(x'_4 \oplus k_4)
\end{aligned}
$$

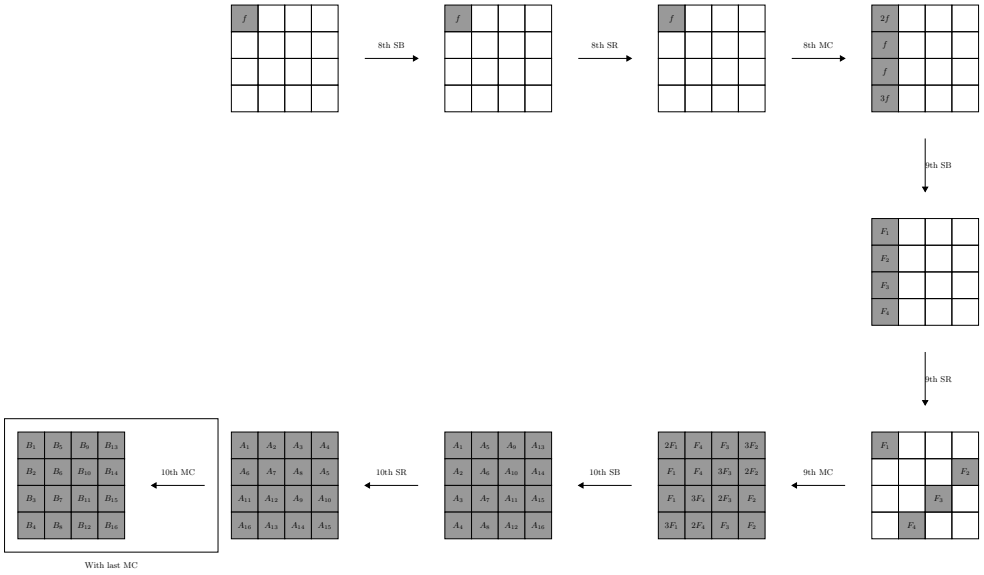


Fig. 2. Fault propagation with an equal number of rows and columns, exemplary for RC = 44. The fault propagates just enough to change the entire matrix.

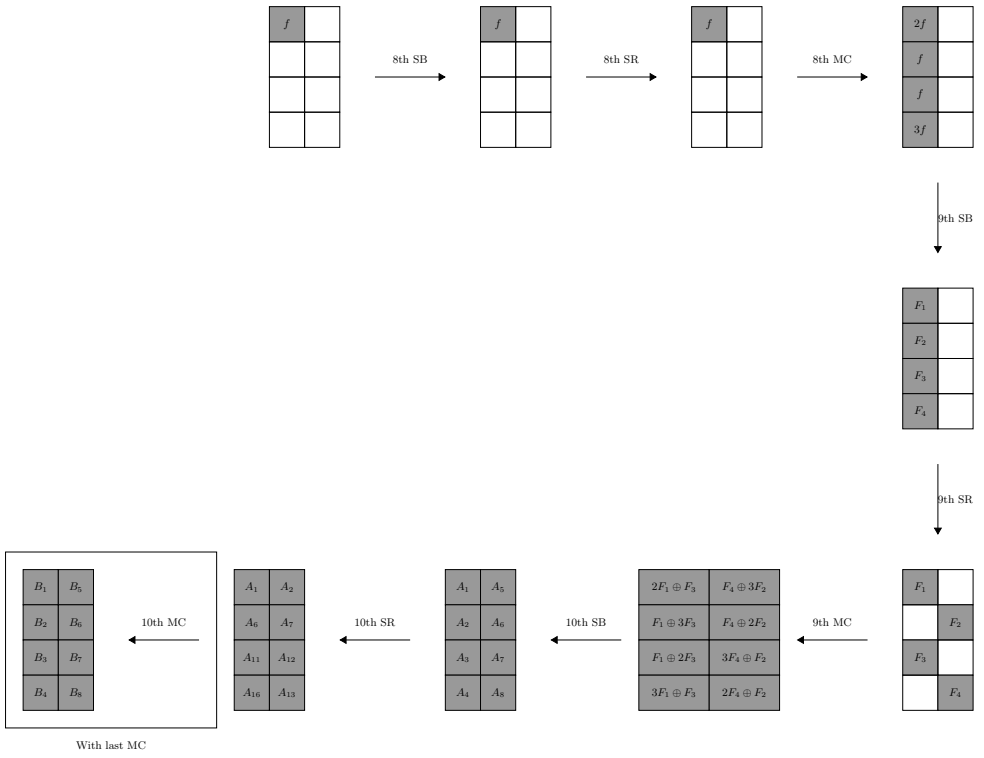


Fig. 3. Fault propagation when the number of rows is larger than the number of columns, exemplary for RC = 42. The fault propagates through the entire matrix but introduces too many changes resulting in more complex equations.

3.2 Reduction of the Key Space

It is possible to add a second step to reduce the key space by using the key schedule equations, similar to the method presented in [6]. This greatly decreases the computational effort required to

acquire the correct key. The key schedule being slightly different for the small scale AES, here is one example of equations for $RC = 44$ that can be used to reduce the key space.

$$
\begin{array}{llll}
k_1^{-1} = S^(k_{16} \oplus k_{12}) \oplus k_1 \oplus \kappa_i & k_5^{-1} = k_5 \oplus k_1 & k_9^{-1} = k_9 \oplus k_5 & k_{13}^{-1} = k_{13} \oplus k_9 \\
k_2^{-1} = S^(k_{15} \oplus k_{11}) \oplus k_2 & k_6^{-1} = k_6 \oplus k_2 & k_{10}^{-1} = k_{10} \oplus k_6 & k_{14}^{-1} = k_{14} \oplus k_{10} \\
k_3^{-1} = S^(k_{14} \oplus k_{10}) \oplus k_3 & k_7^{-1} = k_7 \oplus k_3 & k_{11}^{-1} = k_{11} \oplus k_7 & k_{15}^{-1} = k_{15} \oplus k_{11} \\
k_4^{-1} = S^(k_{13} \oplus k_9) \oplus k_4 & k_8^{-1} = k_8 \oplus k_4 & k_{12}^{-1} = k_{12} \oplus k_8 & k_{16}^{-1} = k_{16} \oplus k_{12}
\end{array}
$$

3.3 Equations With the Last Mix Columns

If the *MixColumns* operation is performed in the last round one equation becomes dependent on a full column instead of a simple key part k_i.

This increases the size of the exhaustive search that we have to go through, instead of going through δ_i and k_i, we have to go through a tuple of k_i. In the worst case scenario, when $RC = 44$, this is equivalent to a 2^{32} exhaustive search. Fortunately, with one fault injection, we also get several equations that involve the same key-tuple, thus already leading to a key space reduction.

Another noticeable thing is that, in the case $RC = 42$, it is not needed to inject two faults to get the full key. Thanks to the dependence on one column, it is possible to get the whole key with only one fault injection.

4 Algebraic Representations

In the section we will show how to rewrite the small scale AES cipher and fault equations given above into the polynomial form. Let us start with the fault equations.

If we are dealing with 4-bit resp. 8-bit version, we introduce 4 resp. 8 new polynomial equations over the finite field $\mathbb{F}_2$ per one "old" fault equation and 4 resp. 8 new variables per one "old" variable in the fault equations. The main idea is as follows: we split word variables into bit variables and on this level we find polynomial relations between input and output bits of the transformations.

We represent $\mathbb{F}_{2^4} \simeq \mathbb{F}_2[\alpha]/\langle \alpha^4 + \alpha + 1 \rangle$ and $\mathbb{F}_{2^8} \simeq \mathbb{F}_2[\alpha]/\langle \alpha^8 + \alpha^4 + \alpha^3 + \alpha^2 + 1 \rangle$. Then we can perform the field multiplication or addition in this representation. E.g. the bits (a_3, a_2, a_1, a_0) corresponds to $a = a_3\alpha^3 + a_2\alpha^2 + a_1\alpha + a_0 \in \mathbb{F}_{2^4}$ in the 4-bit version. Thus we can easily find polynomial relations for the field multiplication and the addition (i.e. XOR in this situation) on the bit level.

The inversion of the S-boxes (both 4-bit and 8-bit version) can be rewritten as a system of polynomials mapping the input bits to the output bits, because every map over finite field is in fact a polynomial map. This might be done with the Buchberger Möller algorithm ([7], Thm. 6.3.10). For example for the 4-bit version we obtain:

$$
\begin{aligned}
y_0 &= x_0x_1x_2 + x_0x_2x_3 + x_1x_2x_3 + x_1x_2 + x_0x_1 + x_2x_3 + x_2 + x_1 + 1 \\
y_1 &= x_0x_1x_2 + x_0x_1x_3 + x_1x_2x_3 + x_0x_3 + x_1 + 1 \\
y_2 &= x_0x_1x_2 + x_0x_1x_3 + x_0x_2x_3 + x_0x_2 + x_1x_3 + x_2x_3 + x_2 + x_3 + 1 \\
y_3 &= x_0x_1x_2 + x_0x_1x_3 + x_0x_1 + x_2x_3 + x_1x_2 + x_0 + x_1 + x_3,
\end{aligned}
$$

where x_i are the input variables and y_i are the output variables of the inverse S-box.

As previously mentioned, for each variable that can be assigned from $\mathbb{F}_{2^4}$ resp. $\mathbb{F}_{2^8}$ in the "old" fault equation we introduce 4 resp. 8 new variables that encode the bits of the word. We start rewriting the innermost expression (usually of the type $x_i \oplus k_i$), then the inverse S-box and finally the outer XOR that connects the inverse S-boxes. The left-hand side of the equations is trivial (i.e. only field multiplication). At the end, we have to simplify the expressions, because we work in the Boolean ring (e.g. $x_i + x_i = 0$ holds here).

In the similar manner, we can rewrite whole small scale AES cipher into the polynomial form.

5 Propositional Representations

Similar to the transformation discussed in the previous section, we converted the hardware implementation as into propositional formulas in conjunctive normal form (CNF). This can be achieved

efficiently through the Tseitin transformation [8]. In addition, we also converted the algebraic representation into formulas in CNF. This yields two formulas for each small scale AES variant.

By assigning the variables corresponding to the input and key and solving the formula, the variables representing the outputs will take on encrypted or decrypted value.

Solving such formulas is the well known SAT problem, for which many highly optimized algorithms exist, e.g. [9,10].

Even though the formulas in Sections 4 and 5 have different origins they both represent the exact same problem. However, due to the very different original encodings (circuit and polynomial) combining these two versions might yield additional insights or help the solver by providing additional structural information.

6 Conclusion

In this paper we presented the first hardware implementation of the small scale AES cipher which is used for cryptographic research. The implementation is available in VHDL as well as on gate level and performs the encryption and decryption for a variable number of rounds and all the different variants of the cipher.

In addition, we also provide fault attack equations which can be used to calculate the secret key if a fault is injected into the operations. Depending on the small scale variant, this can be achieved with only one or two fault injections.

Finally, we converted the cipher description into algebraic and propositional formulas. Solving these formulas is similar to performing the calculation steps in the description of the AES algorithm or setting the inputs of the circuits to the corresponding values. In all cases the en- or decryption is performed and the result will always be the same.

Having all these resources available in a single and well structured package allows researchers to combine the different sources of information which might reveal new patterns or solving strategies. Additionally, the fine granularity of difficulty between the different small scale AES versions allows for the assessment of new attacks or the comparison of different attacks.

All files discussed here are available at: `http://afa.fim.uni-passau.de/en/benchmarks/`

Acknowledgements

This work was partially supported by the DFG project "Algebraic Fault Attacks" (PO 1220/7-1, BE 1176 20/1, KR 1907/6-1) and by the COST Action TRUDEVICE.

References

1. National Institute of Standards, Technology (NIST): Advanced Encryption Standard (FIPS PUB 197) (2001)
2. Cid, C., Murphy, S., Robshaw, M.J.B.: Small scale variants of the aes. Fast Software Encryption: 12th International Workshop (2005) 145–162
3. Stein, W., Joyner, D.: SAGE: System for Algebra and Geometry Experimentation. ACM SIGSAM Bulletin **39**(2) (2005) 61–64. Available at `http://www.sagemath.org`
4. The ApCoCoA Team: ApCoCoA: Approximate Computations in Commutative Algebra. (Available at `http://www.apcocoa.org`)
5. Si2: NanGate FreePDK45 generic open cell library, v1.3. (`http://www.si2.org/openeda.si2.org/projects/nangatelib`)
6. Tunstall, M., Mukhopadhyay, D., Ali, S. In: Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault. Springer Berlin Heidelberg, Berlin, Heidelberg (2011) 224–233
7. Kreuzer, M., Robbiano, L.: Computational commutative algebra 2. Springer Berlin (2005)
8. Tseitin, G.: On the Complexity of Derivation in Propositional Calculus. Studies in Constructive Mathematics and Mathematical Logic (1968)
9. Schubert, T., Reimer, S.: antom. In: `https://projects.informatik.uni-freiburg.de/projects/antom`. (2016)
10. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: SAT 2005: 8th International Conference on Theory and Applications of Satisfiability Testing. SAT'05, Berlin, Heidelberg, Springer-Verlag (2005) 61–75