Quantum Analysis of AES

Kyungbae Jang\textsuperscript{1}, Anubhab Baksi\textsuperscript{2}, Gyeongju Song\textsuperscript{1}, Hyunji Kim\textsuperscript{1},
Hwajeong Seo\textsuperscript{1}, and Anupam Chattopadhyay\textsuperscript{2}

\textsuperscript{1} Division of IT Convergence Engineering, Hansung University, Seoul, South Korea
\textsuperscript{2} Temasek Laboratories, Nanyang Technological University, Singapore

starj1023@gmail.com, anubhab001@e.ntu.edu.sg, thdrudwn98@gmail.com, khj1594012@gmail.com,
hwajeong84@gmail.com, anupam@ntu.edu.sg

Abstract. Quantum computing is considered among the next big leaps in the computer science. While a fully functional quantum computer is still in the future, there is an ever-growing need to evaluate the security of the secret-key ciphers against a potent quantum adversary. Keeping this in mind, our work explores the key recovery attack using the Grover’s search on the three variants of AES (-128, -192, -256) with respect to the quantum implementation and the quantum key search using the Grover’s algorithm. We develop a pool of implementations, by mostly reducing the circuit depth metrics. We consider various strategies for optimization, as well as make use of the state-of-the-art advancements in the relevant fields. In a nutshell, we present the least Toffoli depth and full depth implementations of AES, thereby improving from Zou et al.’s Asiacrypt’20 paper by more than 98 percent for all variants of AES. Our qubit count - Toffoli depth product is improved from theirs by more than 75 percent. Furthermore, we analyze the Jaques et al.’s Eurocrypt’20 implementations in details, fix its bugs and report corrected benchmarks. To the best of our finding, our work improves from all the previous works (including the recent Eprint’22 paper by Huang and Sun).

Keywords: Quantum Implementation · Grover’s Search · AES

1 Introduction

In the current situation in the world of cryptography, quantum computers are considered an upcoming major threat. This is due to the innate nature of how the quantum computers can efficiently model and solve certain problems. There is an overlap between the problems efficiently solvable by a functional quantum computer and those act as the backbones to certain cryptographic systems. Those problems are hard to solve by a classical computer, hence considered secure as of now, but the security of those systems may be threatened if quantum computers become viable in the future. It is well-known that the public key cryptography will have severe consequence [32], still the secret-key counterpart will likely not be completely unscathed either. Depending on the structure, a secret-key cipher, too, can have severe security flaw against a quantum computer (refer to [22, 39]).

One serious way for this to manifest arises from the observation that, a lot of the post-quantum ciphers use some secret-key ciphers internally as a component in one way or the other (apart from the standalone usage of the secret-key ciphers). This is evident from the current portfolio of the Post-Quantum Cryptography (PQC) standardization\textsuperscript{3} being organized by the US government’s National Institute of Standards and Technology (NIST)\textsuperscript{4}. While the core components of ciphers are based on a problem presumed to be quantum-safe, due to the usage of secret-key ciphers,


\textsuperscript{4}For example, the Public Key Encryption & Key Encapsulation Mechanism (PKE & KEM) finalist CRYSTALS-KYBER [51] and the Digital Signature (DS) finalist CRYSTALS-DILITHIUM [25] use SHA-3 in some form.
it may be possible for the attacker to bypass the overall security claim (i.e., by exploiting only the secret-key component). In other words, it may just so happen that the secret-key component becomes the security bottleneck of the a post-quantum cipher (despite the core components being secure) against a potent quantum computer. Therefore, it is probably a commendable plan to consider the quantum security of the secret-key ciphers, to be on the safe side.

Ultimately, the NIST call for post-quantum ciphers specified five levels of security. Each of the levels are defined over secret-key ciphers (variants of AES for PKE & KEM, and variants of SHA-3 for DS). As noted in [38, Section 1], this essentially calls for a concrete and precise resource estimates that would be required by an attacker with a quantum computer at disposal.

Therefore, finding quantum vulnerabilities of a secret-key cipher is among the top research directions (see Section 1.3 for related works). One of the main way an attacker with a functional quantum computer can try to mitigate the security of the secret-key ciphers is by running the Grover’s search algorithm [31] (refer to Section 1.2 for an overview). As a rule of thumb, it reduces the search space to nearly square root complexity (with a high probability).

Our work makes a humble attempt to conduct a detailed and systematic quantum assisted exhaustive search on the AES family of block ciphers (AES-128, AES-192 and AES-256) [17]. Most recent papers about AES quantum implementations focus on reducing the number of qubits, but do not give much consideration to the depth of the circuit [1, 30, 43, 54, 55, 59]. That said, until a few years ago, quantum computers could not use enough qubits. However, it is hard to say that today’s quantum computers are small anymore. Quantum computers that will emerge in the near future are not small, and this can be observed in IBM’s quantum computer development roadmap⁵. In the Noisy Intermediate-Scale Quantum (NISQ) era, Toffoli depth is probably the most important metric for error-prone quantum computing [59] and full depth is related to the execution time of circuits [10]. The importance of depth is also observed in NIST’s post-quantum security requirements. In estimating the complexity of quantum attacks, NIST uses only the number of gates and depth as metrics, not the number of qubits [49].

We revisit recent research works to incorporate state-of-the-art improvements in various related areas, in a bid to reduce the cost (qubit count, gate count), circuit depth (Toffoli depth, full depth) and/or cost-depth trade-off (Toffoli depth × qubit count) of the quantum circuits. In the process, we carefully weigh and choose from a number of possible options.

1.1 Contribution and Organization

We discuss in detail about the considerations/choices that are made during design separately for AES in Section 3. In particular, we optimize AES for quantum computers, keeping the focus on the Toffoli depth and full depth (our AES quantum circuits attain the least Toffoli and full depths). Further, we also consider the Toffoli depth × qubit count as a trade-off (since the qubit count is also an important metric).

We observe that the implementation by [38] contains some programming related issue, which probably results in underestimating the resources for non-linear components (the same issue was reported by the Asiacrypt’20 authors [59], and they did not use those results either); although the linear components are not affected. We patch the issues (such as impossible parallelism and omitting initialization of ancilla qubits) and estimate the correct quantum gates and depth from the number of qubits they report in Section 4.

Results are consolidated in Section 5 (cost of the implemented quantum circuits), and Section 6 (cost for running the Grover’s search). Comparison of our implementations with respect to the previous works are shown in Table 4 for the three variants of AES. Table 1 shows the overall

---

performance gain of our work with respect to previous AES implementations. It can be seen that we make significant improvement over the Asiacrypt’20 paper [59] (such as our Toffoli depth is reduced by over 98% for AES-128) and also the bug-fixed version of the Eurocrypt’20 paper [38]. We also include the two improved implementations done in [33] for a quick comparison.

We develop multiple quantum implementations of the ciphers in the AES family (AES-128, AES-192 and AES-256), and report the least depth implementations so-far (with moderate number of qubits and quantum gates). Optimization is done at three levels, namely individual component level (S-box, MixColumn etc.), architecture level (16 S-boxes to make 1 SubBytes, 4 MixColumn to make 1 MixColumns etc.), and finally by sharing of resources among the modules. We present a pool of three implementations, each optimized for a specific objective (see Section 2 for related discussion):

- The regular version uses the least qubit count in our work and reduces Toffoli circuit depth compared to the previous works for all the 3 variants.
- The shallow version runs all parallel-executable parts of AES simultaneously, including reverse operations. The depth of one round only counts SubBytes + MixColumns, which is ideal. The shallow version takes the least qubit cost and Toffoli circuit depth product with an improved pipeline architecture. According to [59], this is an important a notion of circuit complexity.
- Further, the shallow/low depth version looks for reducing the circuit depth by opting for a low quantum depth implementation of MixColumn (which was found in [44]).

<table>
<thead>
<tr>
<th>Cipher</th>
<th>Source</th>
<th>#TD*</th>
<th>#M*</th>
<th>#TD \times #M** Full depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES-128</td>
<td>GLRS [30]</td>
<td>12,672 (99.76)</td>
<td>984 (−84.55)</td>
<td>12,469,248 (97.96)</td>
</tr>
<tr>
<td></td>
<td>LPS [43]</td>
<td>1,880 (98.41)</td>
<td>864 (−86.43)</td>
<td>1,624,320 (84.32)</td>
</tr>
<tr>
<td></td>
<td>ZWSLW [59]</td>
<td>2,016 (98.51)</td>
<td>512 (−91.96)</td>
<td>1,032,192 (75.32)</td>
</tr>
<tr>
<td></td>
<td>HS [33] (p = 18)</td>
<td>820 (95.12)</td>
<td>492 (−92.27)</td>
<td>403,440 (30.86)</td>
</tr>
<tr>
<td></td>
<td>HS [33] (p = 9)</td>
<td>1,558 (97.43)</td>
<td>374 (−94.13)</td>
<td>582,692 (56.29)</td>
</tr>
<tr>
<td></td>
<td>JNRV [38] (fixed)</td>
<td>2,394 (98.33)</td>
<td>1,656 (−74.00)</td>
<td>3,964,464 (93.58)</td>
</tr>
<tr>
<td>This work</td>
<td>40°*</td>
<td>6,368°</td>
<td>254,720°</td>
<td>978°</td>
</tr>
<tr>
<td>AES-192</td>
<td>GLRS [30]</td>
<td>11,088 (99.68)</td>
<td>1,112 (−83.37)</td>
<td>12,329,856 (97.34)</td>
</tr>
<tr>
<td></td>
<td>LPS [43]</td>
<td>1,640 (98.71)</td>
<td>896 (−86.60)</td>
<td>1,469,440 (78.15)</td>
</tr>
<tr>
<td></td>
<td>ZWSLW [59]</td>
<td>2,022 (98.22)</td>
<td>640 (−90.43)</td>
<td>1,294,080 (75.19)</td>
</tr>
<tr>
<td></td>
<td>JNRV [38] (fixed)</td>
<td>2,682 (98.21)</td>
<td>1,976 (−70.46)</td>
<td>5,299,632 (93.94)</td>
</tr>
<tr>
<td>This work</td>
<td>48°*</td>
<td>6,688°</td>
<td>321,024°</td>
<td>1,174°</td>
</tr>
<tr>
<td>AES-256</td>
<td>GLRS [30]</td>
<td>14,976 (99.72)</td>
<td>1,336 (−80.85)</td>
<td>20,007,936 (98.05)</td>
</tr>
<tr>
<td></td>
<td>LPS [43]</td>
<td>2,160 (98.06)</td>
<td>1,232 (−82.34)</td>
<td>2,661,120 (85.32)</td>
</tr>
<tr>
<td></td>
<td>ZWSLW [59]</td>
<td>2,292 (98.17)</td>
<td>638 (−88.99)</td>
<td>1,760,256 (77.81)</td>
</tr>
<tr>
<td></td>
<td>JNRV [38] (fixed)</td>
<td>3,306 (98.31)</td>
<td>2,296 (−67.09)</td>
<td>7,590,576 (94.85)</td>
</tr>
<tr>
<td>This work</td>
<td>56°*</td>
<td>6,976°</td>
<td>390,656°</td>
<td>1,377°</td>
</tr>
</tbody>
</table>

Parenthesized numbers show % improvement reported in this work.

- ⚫: #TD is Toffoli depth.
- ✡: #M is qubit count.
- ☪: Regular version (using MixColumn from [56]).
- ◯: Shallow version (using MixColumn from [56]).
- ✤: Shallow/low depth version (using MixColumn from [44]).

We conclude in Section 7. Some additional information/discussion can be found in Appendices A (description of the AES variants), B (details about implementation and result) and C (a brief
comparison of classical and quantum depths). Our source codes are written in IBM ProjectQ\textsuperscript{6}, which is a Python-based open-source framework for quantum computing. All our relevant source codes can be accessed as an open-source project\textsuperscript{7}.

As noted, we make use of the state-of-the-art progress in the relevant areas. For instance, coming to the implementation of AES MixColumn, a few have been proposed in a relatively short time [45, 46, 47, 56]. We have experimented with all these, and ultimately choose that of [56] (for regular and shallow versions) and [46] (for shallow/low depth version).

Reflection on Huang and Sun (Eprint’22) We are aware of the parallel development by Huang and Sun (Eprint’22) [33]. The content of this paper only revolves with AES-128, and can be summarized as:

- Improve from the Asiacrypt’20 paper’s [59] qubit count and performance.
- Choose an improved S-box implementation atop the Eurocrypt’20 implementation [38] with proposal for a quick fix for the qubit count.

In our humble opinion, this fix done by [33] on the Eurocrypt’20 implementation is not perfect (based on the Q# code\textsuperscript{8}). Also, the number of qubits was estimated manually.

In our paper the main contributions are, low depth implementations of AES and a thorough bug-fixing of the Eurocrypt’20 implementations. Our approaches are mostly disjoint from that of [33]; and when their S-box implementation is used in our implementation, our result outperforms theirs (thus we have the best-known implementation so far). Parallelism is a major focus in their work, which we pursue through our shallow version. Further, we cover optimized quantum implementations of AES-192 and AES-256 as well.

1.2 Quantum Key Search using Grover’s Algorithm

For a secret-key cipher using an $k$-bit key, $2^k$ queries are required for the exhaustive key search. The Grover’s search [31] is a well-known quantum algorithm that recovers the key with a high probability in about $\floor{\frac{\pi}{4} \sqrt{\frac{2^k}{\theta}}}$ queries. The procedure can be briefly described as follows (some basic familiarity with the quantum notations/terminology is assumed, one may refer to, e.g., [21, 48] for a more detailed description):

1. A $k$-qubit key is prepared in superposition $|\psi\rangle$ by applying Hadamard gates. All states of qubits have the same amplitude:

$$|\psi\rangle = H^{\otimes k} |0\rangle^{\otimes k} = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = \frac{1}{2^{k/2}} \sum_{x=0}^{2^k-1} |x\rangle$$

2. The cipher is implemented as a quantum circuit and placed in oracle. In oracle $f(x)$, the plaintext is encrypted with the key in the superposition state. As a result, ciphertexts for all key values are generated. The sign of the solution key is changed to a negative by comparing it with the known ciphertext. The condition ($f(x) = 1$) changes the sign to negative and applies to all states. For this phase flip, an $n$-qubit controlled Z gate is utilized ($n$ is the length of the ciphertext).

\textsuperscript{6}Homepage: https://projectq.ch/.
\textsuperscript{7}https://github.com/starj1023/AES_QC.
\[ f(x) = \begin{cases} 1 & \text{if } Enc(k) = c \\ 0 & \text{if } Enc(k) \neq c \end{cases} \]  

(2)

\[ U_f(|\psi\rangle |\rangle) = \frac{1}{2^{k/2}} \sum_{x=0}^{2^k-1} (-1)^{f(x)} |x\rangle |\rangle \]  

(3)

3. Lastly, the diffusion operator\(^9\) amplifies the amplitude of the negative sign state. Diffusion operator is implemented with the following (H gates layer → X gates layer → k-qubit controlled Z gate → X gates layer → H gates layer). In [50], a simple technique was introduced by which a constant number of X gates are used for the diffusion operator. If a constant number of X gates are applied before the Hadamard gates in Step 1, the diffusion operator is implemented as (H gates layer → k-qubit controlled Z gate → H gates layer).

The Grover’s search executes Equations (2) and (3) in a series to sufficiently increase the amplitude of the solution and observes it at the end. For an k-bit key, the optimal number of iterations of the Grover’s search algorithm is roughly \([\frac{\pi}{4} \sqrt{2^k}]\) [16], which is about \(\sqrt{2^k}\). In the process, an exhaustive key search that requires \(2^k\) queries in a classic computer is reduced to roughly \(\sqrt{2^k}\) queries in a quantum computer (this works with a high probability).

In the exhaustive key search, \(r = \lceil k/n \rceil\) (plaintext, ciphertext) pairs are needed to recover a unique key that is not a spurious key (see Section 6 for details). Figure 1 shows the Grover’s oracle of exhaustive key search. Encryption\(^1\) is defined as the reverse operation of encryption, which reverts to the state before encryption.

![Grover’s oracle for r = 1](image)

(a) Grover’s oracle for \(r = 1\).

![Grover’s oracle for r = 2](image)

(b) Grover’s oracle for \(r = 2\).

**Fig. 1:** Schematic architecture for key search using Grover’s algorithm.

1.3 Related Works

Quantum analysis of secret-key ciphers with respect to the Grover’s search algorithm is one of the major research direction now-a-days. Some of the prominent examples include, but not limited to, AES [13,38,42,43,59], SIMON [6], SPECK [5,34], PRESENT and GIFT [36], SHA-2 and SHA-3 [2], FSR-based ciphers [4], ChaCha [9], SM3 [52,57], RECTANGLE and KNOT [7], DEFAULT [35].

However, this is not the only active direction of research; there are other avenues which try to find an efficient quantum attack for a secret-key cipher. One may, for instance, refer to classical attacks that are ported to the quantum realm [28,40], or specialized quantum attacks like [12,23,24]. These avenues, though important, are out-of-scope for our current work.

\(^9\)Since the diffusion operator is usually generic, it does not require any special techniques to implement.
2 Regular and Shallow Versions

Our quantum circuit implementations are divided into regular and shallow versions. The regular version offers high parallelism while taking into account the trade-off of qubit-depth. The shallow version also considers the trade-off of qubit-depth, but further reduces the depth by burdening the use of qubit. This achieves the ideal depth for quantum circuit implementations.

The regular version of AES focuses on the parallelism within the round. In the regular version, when the next round is continued, waiting occurs due to the reverse operations of the previous round. In other words, the next round cannot start until the reverse operation is complete. On the other hand, the shallow version of AES succeeds in parallelization while processing all rounds. In the shallow version, the reverse operation of the previous round is run simultaneously with the operations of this round in an alternate approach. The shallow version uses more qubits, but offers lower depth because all rounds of parallel-operable parts run completely simultaneously. Shallow version achieves an ideal circuit depth that counts as depth of SubBytes plus MixColumns in every round (except the last round, which only counts SubBytes).

3 AES in Quantum

Most papers implementing quantum circuits for AES focus on reducing the usage of qubits \[1, 30, 43, 55, 59\]. However, the serial circuit structure is forced to reduce the qubits, which significantly increases the circuit depth. Our quantum circuit implementations for AES considers the trade-off of using qubits in the best possible way to reduce circuit depth. As a result, our AES quantum circuit implementations provide the best trade-off in \#TD \times \#M, where \#TD is the Toffoli depth, and \#M is the number of qubits. This product is taken as the trade-off indicator for the quantum circuit presented in \[59\].

3.1 Regular Implementations of AES Quantum Circuits

The round function of AES is composed of SubBytes, ShiftRows, MixColumns, and AddRoundKey. For the key schedule, an on-the-fly approach is adopted, and our AES quantum circuit implementation executes the key schedule simultaneously with SubBytes in the round function. All the S-boxes in key schedule and round function are designed to operate in parallel. That is, the depth is the same as operating an 8-bit S-box once. Quantum implementation for S-box is required for key schedule and SubBytes, and S-box occupies the most resources in AES quantum circuit. In \[30\], Grassl et al. used Itoh–Tsujii inversion to implement S-box of AES, which requires a lot of quantum resources. Recently, the hardware design for AES has been adopted to implement an efficient S-box quantum circuit. In particular, S-box implementation techniques \[14, 15\] proposed by Boyar-Peralta were frequently used. In \[43\], Langenberg et al. adopted the S-box implementation of \[14\] and converted it to suit their purpose of reducing qubits. The S-box implementation of \[14\] was adopted and improved in \[58\]. Zou et al. \[59\] also used the S-box\(^{-1}\) implementation in designing a new architecture for AES that reduced number of qubits.

3.2 Implementation of S-box (SubByte)

Table 2 shows the resources required for the naïve implementations by Boyer-Peralta \[14, 15\] and the resources for the S-boxes used by the previous authors \[43, 59\]. Resource estimation is performed in IBM ProjectQ and according to the method of \[3\], one Toffoli gate is decomposed into 7 T gates + 8 Clifford gates, T-depth of 4 , and full depth of 8.
Apart from these, another method which is a courtesy of Dansarie [18, 19] exists. This is rather generic, as it can find implementation of an arbitrary 8-bit S-box (i.e., not specific to the AES S-box, which is the case for [14, 15]), with respect to a user-provided set of logic gates. With the publicly available source codes\(^{10}\) we checked the implementation of the AES S-box. However, the cost for the AES S-box seems to be more than 400 gates, therefore we kept the proper applicability of [18, 19] as a future work.

<table>
<thead>
<tr>
<th>Method</th>
<th>#CNOT</th>
<th>#1qCliff</th>
<th>#T</th>
<th>#TD</th>
<th>#qubits</th>
<th>Full depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-box [14]</td>
<td>358</td>
<td>68</td>
<td>224</td>
<td>8</td>
<td>123</td>
<td>104</td>
</tr>
<tr>
<td>S-box [15]</td>
<td>392</td>
<td>72</td>
<td>238</td>
<td>6</td>
<td>136</td>
<td>85</td>
</tr>
<tr>
<td>S-box [43]</td>
<td>628</td>
<td>98</td>
<td>367</td>
<td>40</td>
<td>32</td>
<td>514</td>
</tr>
<tr>
<td>S-box [59]</td>
<td>437</td>
<td>72</td>
<td>245</td>
<td>55</td>
<td>22</td>
<td>339</td>
</tr>
<tr>
<td>S-box [33]*</td>
<td>418</td>
<td>72</td>
<td>238</td>
<td>4</td>
<td>136</td>
<td>72</td>
</tr>
<tr>
<td>S-box [33]</td>
<td>824</td>
<td>160</td>
<td>546</td>
<td>3</td>
<td>198</td>
<td>69</td>
</tr>
</tbody>
</table>

\(^{\oplus}\): Used in this work.

If the Boyer-Peralta’s S-box implementations [14, 15] are naïvely ported to quantum, the quantum version of [15] requires more ancilla qubits (i.e., 120 ancilla qubits excluding input and output) than the quantum version of [14] (107 ancilla qubits), but provides lower depth. In [38], Jaques et al. adopted the naïve implementation of the S-box of [15] on a quantum circuit.

Very recently, Huang and Sun reported an improved quantum implementation for the S-box of [38] in their Eprint’22 paper [33]. They presented two quantum implementations of reduced Toffoli depth with new observations of the classical implementation of the AES S-box as given in [15]. The first version reduced the Toffoli depth without increasing the number of qubits, while the second version used more qubits to further reduce the Toffoli depth.

In [43, 59], the authors extended the first S-box implementation by Boyar-Peralta [14] and presented the S-box quantum circuit with a reduced number of qubits. Consequently, it leaves us with a few of ways to choose from.

Considering the trade-off between the circuit depth and the number of qubits required for an S-box implementation, we treat two cases. The first case is when the ancilla qubits have to be allocated per SubBytes, which is indeed sensitive to the number of qubits. The second case is when the initially allocated ancilla qubits can be reused. In this case, there is no need to allocate additional ancilla qubits for the next SubBytes. Therefore, the number of ancilla qubits is maintained, but the depth increases accordingly. We choose the second S-box implementation for our case, since we believe the benefit of depth reduction outweighs the price for initial allocation of additional qubits. Alternatively, it is also possible to implement a method of allocating new ancilla qubits to each S-box. In this case, since there is no need for additional operations to reuse ancilla qubits, the fewest quantum gates and the lowest circuit depth are obtained. However, we think that this trade-off of the number of qubits-depth (gates) places an excessive burden on the number of qubits. Thus, we use Huang and Sun’s [33] S-box implementations with relatively high qubit count but low depth.

One may note that the AES implementation in [59] required the implementation of the inverse S-box. In our case, however, we do not use the inverse S-box.

\(^{10}\)https://github.com/dansarie/sboxgates.
3.3 Implementation of SubBytes

After we decide upon the implementation of one S-box (SubByte, Section 3.2), this can be used to implement 16 S-boxes (SubBytes). Regarding the implementation of SubBytes in AES, Figure 2(a) is the method that uses the fewest qubits. In this case, all S-boxes are executed sequentially, which causes a significant increase in depth, as shown in Figure 2(a). On the other hand, we reduce the depth by allocating more ancillas set initially. The notation S-box† is described in Appendix A.

![Diagram](https://example.com/diagram.png)

(a) Using 1 set of ancillas.

![Diagram](https://example.com/diagram.png)

(b) Using multiple sets of ancillas.

Fig. 2: SubBytes implementation.

In one round, 16 S-boxes in SubBytes and 4 S-boxes in key schedule, a total of 20 S-boxes are operated, simultaneously. Therefore, we allocate 20 × 120 ancilla qubits for S-box with Toffoli depth 4 and 20 × 182 ancilla qubits for S-box with Toffoli depth 3 to run all S-boxes simultaneously. After
S-box operations, ancilla qubits are not in a clean state (i.e., not all ancilla is 0). Initialization (i.e., returning to 0) is performed in parallel for the next round. Figure 2(b) shows 16 S-boxes operation in parallel using multiple ancillas sets. In [59], 16 S-boxes of SubBytes were implemented in parallel using residual ancillas, but key schedule was not implemented in parallel with SubBytes.

3.4 Implementation of Key Schedule

In the key schedule of AES, SubWord operates on rearranged 32-qubit. Out of the $20 \times (120$ or 182) ancilla qubits previously decided to use (refer to Section 3.2), $4 \times (120$ or 182) ancilla qubits are used to simultaneously operate S-boxes for 32-qubit in the key schedule (16 × 120 or 16 × 182 ancilla qubits are used in SubBytes of round). For rearranging the 32 qubits, quantum resources are not used by using logical swap that only changes the index of the qubits.

In SubBytes, the outputs of S-boxes are stored in new qubits. On the other hand, in the key schedule, no additional qubits are allocated because the outputs of the S-boxes are XORed (using CNOT gates) inside the key. Since SubWord for 32-qubit operates in parallel with SubBytes of round, there is no depth overhead in our AES quantum circuit implementation. This approach is already utilized in [38]. XORing the 8-bit round constant (RC) is implemented by performing X gates to $|k_{120-127}\rangle$ according to the positions where the bit value of the round constant is 1. Lastly, the CNOT gates inside the key are performed. Figure 3 shows the quantum circuit for the AES-128 key schedule (see Appendix A for description of Rotation† and SubWord†).

![Fig. 3: AES-128 key schedule in quantum.](image)

In most implementations of AES quantum circuits, the full depth and Toffoli depth of AES-128 are higher [30, 38, 43] or similar [59] to those of AES-192. Although AES-128 has fewer rounds, this is due to differences in key schedule. AES-128 requires 16 S-boxes for SubBytes and 4 S-boxes for key schedule in every round. On the other hand, some rounds of AES-192 require only 16 S-boxes for SubBytes, since SubWord in the key schedule are not required. As a result, AES-128 has a higher depth than AES-192.

Another interpretation of this is that there is an overhead for key schedules in implementing AES quantum circuits. However, in our AES quantum circuits there is no overhead for key schedule (except for gates). Our AES quantum circuit runs the key schedule in complete parallel, so we achieve the same depth as if the key schedule was omitted. As a result, unlike other implementations, the quantum resources required for our AES-128, 192, and 256 quantum circuits are strictly dependent on the number of rounds.
3.5 Implementation of AddRoundKey and ShiftRows

The AddRoundKey operation, which XORs a 128-qubit round key, can be implemented simply by using 128 CNOT gates. In the case of ShiftRows, it can be implemented using swap gates, but quantum resources are not used through logical swap that changes the index of qubits. Since no special implementation technique is applied for AddRoundKey and ShiftRows, this approach is mostly used in quantum circuit implementations.

3.6 Implementation of MixColumn (and MixColumns)

In [56], Xiang et al. presented a novel heuristic search algorithm to optimize the implementation of linear layers based on factorization of binary matrices. When applied to the MixColumn of AES, their algorithm resulted in an implementation using 92 XOR gates (with classical depth 6) in a classical circuit. A different implementation costing 92 XOR gates (with classical depth 6) was reported previously by [47]. These two were the least cost implementations in classical circuits, until another implementation with 91 XOR gates (with classical depth 7) was found by [45]. Recently, a new implementation of AES MixColumn was found thanks to [46], which managed to reduce the classical depth to 3 with 103 XOR gates (cf. 103 XOR/3 classical depth implementation from [8]). However, this work came as a tie with another implementation from [44], albeit the latter requires 105 XOR gates.

While dealing with quantum circuit, the best option (in terms of gate cost) seems to be that one reported by [56], as can be seen from Table 3. This implementation is relatively easier to port to a quantum circuit, since the operations are done in-place (i.e., of the form: \( a \leftarrow a \oplus b \)). In contrast, implementations like that of [45, 46, 47], are not compatible per-se, as those require usage of temporary variables. These temporary variables would incur additional cost (due to extra ancilla qubits in parallel implementation) and/or depth (due to cleaning up qubits) when these are converted to quantum circuits. On a different direction, the implementation from [44] appears to have lower depth than that of [46] when converted to quantum circuits.

In addition to that, a direct comparison with the quantum MixColumn implementations used by the previous papers [1, 29, 38, 59], this representation is the most efficient in terms of number of qubits (32, a tie with [1, 29, 38, 59]), CNOT gates (92, which is the same as the number of XOR gates in classical, and by far and large the best).

To the best of our knowledge, the result by [56] has never been applied to the implementation of quantum circuits for AES (except for the very recent [33]), and so is the case for [44]. We port the implementation of MixColumn in [56] to quantum and use it in our AES quantum circuit. This implementation is referred to as, the low-resource implementation. Additionally, in order to minimize the circuit depth, we also use the MixColumn implementation from [44] in our shallow/low depth version. In our AES quantum circuit, if we adopt an implementation that requires temporary variables for MixColumns (i.e., not in-place), the overhead of ancilla qubits for temporary variables is offset. This is because some of the ancilla qubits allocated from SubBytes can be reused in MixColumns. In SubBytes, ancilla qubits are initialized to 0 for reuse in the next round. We reuse these ancilla qubits in MixColumns as a concept to borrow for a while. If it is a stand-alone MixColumn, ancilla qubits for temporary variables are required, but since we reuse ancilla qubits, only 64-qubits are used for the input and output of the MixColumn.

In other words, for our implementation, it does not matter how many ancilla qubits are required in MixColumn. That is, unless it is an in-place MixColumn, there is no qubit count-depth trade-off, so the MixColumn with the lowest depth is the best regardless of the number of qubits.

For the 128-bit MixColumns operation (i.e., 4 MixColumn operations), the MixColumn implementation can be scaled up directly (with proper adjustment for quantum depth).
Table 3: Comparison of quantum resources required for AES MixColumn implementations.

<table>
<thead>
<tr>
<th>Method</th>
<th>#CNOT</th>
<th>#qubits</th>
<th>Full depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>MixColumn [29, 59]</td>
<td>277</td>
<td>32</td>
<td>39</td>
</tr>
<tr>
<td>MixColumn [41]</td>
<td>194</td>
<td>129</td>
<td>15</td>
</tr>
<tr>
<td>MixColumn [1]</td>
<td>275</td>
<td>32</td>
<td>200</td>
</tr>
<tr>
<td>MixColumn [47]</td>
<td>184</td>
<td>124</td>
<td>16</td>
</tr>
<tr>
<td>MixColumn [38]</td>
<td>277</td>
<td>32</td>
<td>111</td>
</tr>
<tr>
<td>MixColumn [53]</td>
<td>188</td>
<td>126</td>
<td>17</td>
</tr>
<tr>
<td>MixColumn [56]</td>
<td>92</td>
<td>32</td>
<td>30</td>
</tr>
<tr>
<td>MixColumn [45]</td>
<td>182</td>
<td>123</td>
<td>16</td>
</tr>
<tr>
<td>MixColumn [46]</td>
<td>206</td>
<td>135</td>
<td>13</td>
</tr>
<tr>
<td>MixColumn [44]</td>
<td>210</td>
<td>137</td>
<td>11</td>
</tr>
</tbody>
</table>

✝ ✜: Reused in this work to fix [38].
# ✜: Used in this work (regular, shallow versions), [33].
✝: Least XOR count in classical circuit.
ⓘ: Least depth in classical circuit.
*: Used in this work (shallow/low depth version).

3.7 Architecture of Quantum Circuits

There are several architectures for designing quantum circuits of AES. The architectures differ in how they store the 128-qubit output generated from SubBytes in each round. In [1, 30, 43], the basic zig-zag architecture (Figure 4(a)) was adopted that uses 4 lines to save qubits by performing reverses on rounds. In [59], an improved zig-zag architecture that requires only 2 lines of qubits (Figure 4(b)) was presented. By using a quantum circuit of S-box⁻¹, they were able to achieve an improved architecture using fewer qubits. The basic pipeline architecture allocates 128-qubits every round and does not need reverses of rounds. Simply put, the zig-zag architecture requires reverse operations on rounds to save qubits, significantly increasing depth and gates. The pipeline architecture allocates new qubits per round, but does not require reverse operations, reducing depth and gates. It’s a trade-off issue, but in a sense, a generic pipeline is probably the most efficient architecture for implementing AES quantum circuits. We believe that it is much more efficient to allocate a new 128-qubits per round than doubling the gates, depth by performing reverse operations on the rounds to save qubits.

In our approach, which has already allocated many ancilla qubits, the overhead of increasing the number of qubits according to the architecture is relatively low. Therefore, for our implementation, rather than reducing the number of qubits with the zig-zag method, a pipeline architecture that can reduce the depth by omitting the reverses is more suitable. Figure 5(a) shows the pipeline architecture of our AES-128 quantum circuit in more detail for the regular version, and Figure 5(b) shows the same for the shallow version.

In Figure 5(a), SubBytes to generate 128-qubit output and SubBytes† to clean ancilla qubits operate serially 19 times in total (SubBytes is 10, SubBytes† is 9). Depending on the number of rounds, AES-192 operates 23 times and AES-256 operates 27 times. In our parallel design, the key schedule operates simultaneously with SubBytes and MixColumn operates simultaneously with SubBytes†. Therefore, the circuit depth is determined by the number of serial operations of SubBytes and SubBytes†.

In SubBytes, S-boxes operate simultaneously. The depth of SubBytes is 72 equal to the depth of S-box (with Toffoli depth 4) once. Finally, when S-box with Toffoli depth 4 is used, our AES quantum circuits provide a depth of 1,364 (about 72 × 19) for AES-128, 1,627 (about 72 × 23) for AES-256, and 1,907 (about 72 × 27) for AES-256.
Further, we propose a shallow version in which all possible parts of AES quantum circuits operate, simultaneously. When S-box with Toffoli depth 4 is used, this can be achieved by using 2 sets of $20 \times 120$ ancilla qubits. In the shallow version, the first SubBytes in Figure 5(b) uses the first $20 \times 120$ ancilla qubits. The second SubBytes uses the second $20 \times 120$ ancilla qubits, and at the same time SubBytes$^\dagger$ cleans the first $20 \times 120$ ancilla qubits. That is, SubBytes$^\dagger$ operates simultaneously with the SubBytes of the next round. Conceptually, this can be thought as all SubBytes$^\dagger$ in Figure 5(a) are pushed one space to the right. This is possible because SubBytes and SubBytes$^\dagger$ do not share any ancilla qubit. Finally, the shallow version counts the depth for one round as SubBytes (72) + MixColumns (30), which is the ideal depth. The circuit depth of AES-128 is 978 (about 9 rounds $\times 102 + 72$), AES-192 is 1,174 (about 11 rounds $\times 102 + 72$), and AES-256 is 1,377 (about 13 rounds $\times 102 + 72$). The low depth version changes only the MixColumn from the shallow version to a MixColumn based on [44]. The low depth version counts the depth for one round as SubBytes (72) + MixColumns (11). In the shallow version, up to SubBytes$^\dagger$ operates concurrently within one round, providing maximum parallelism. Finally, the shallow version of AES offers the best Toffoli depth of Sbox’s Toffoli depth $\times$ rounds.

![Diagram of AES quantum circuit](image)

(a) Basic implementation.

(b) Improved implementation.

Fig. 4: Zig-zag architecture for AES quantum circuit.

4 Bug-fixing JNRV (Eurocrypt’20) AES Implementations

In this part, we report errors from the AES implementation and resource estimation by Jaques, Naehrig, Roetteler and Virdia in Eurocrypt’20 [38]. To this end, we analyze the Q# code of their AES implementation and cross-compare it with the quantum resources reported in the Eurocrypt’20 paper. Furthermore, we fix bugs in the AES implementation of Eurocrypt’20 and reports the corrected resources for the lower-bound resources in their work.

4.1 Issues with Q#

For a clearer context, we give a brief description of the cases where Q#’s ResourcesEstimator issues arise and how those issues affect the quantum benchmarks given in the Eurocrypt’20
In order to derive the correct result, the number of qubits or depth using the implementation of AES S-box are allocated with the count, qubit count and depth reported in [38] for the non-linear components (namely the S-box and S-box\(^{-1}\) of AES). Indeed, this was also noted in [59] as a bug; and this apparently led to underestimation of gate count, qubit count and depth reported in [38] for the non-linear components (namely the S-box and S-box\(^{-1}\) of AES).

To our understanding, some problems arise if the qubits are allocated by the using command in Q\(_n#\) (and it affects the non-linear components). However more experiments are to be carried out in order to be completely certain about it.

The using command automatically disposes when the function ends. If ancilla qubits to implement AES S-box are allocated with the using command, the consistency between depth and qubits is lost. When 16 S-boxes are executed in SubBytes, the ancilla qubits allocated by the using are counted only for the first S-box and not after. Also counts the depth for executing 16 S-boxes simultaneously. In order to derive the correct result, the number of qubits or depth

\[\text{SB: SubBytes. } \text{SB}^1: \text{Clean ancilla qubits used in SubBytes.}\]

---

\[\text{Fig. 5: Pipeline architecture of AES.}\]
Table 4: Comparison of quantum resources required for variants of AES.

<table>
<thead>
<tr>
<th>Cipher</th>
<th>Source</th>
<th>#CNOT</th>
<th>#NOT</th>
<th>#Toffoli</th>
<th>Toffoli depth</th>
<th>#qubits (M)</th>
<th>T × M</th>
<th>Full depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES-128</td>
<td>GLRS [30]</td>
<td>166,548</td>
<td>1,456</td>
<td>151,552</td>
<td>12,672</td>
<td>984</td>
<td>12,469,248</td>
<td>110,799</td>
</tr>
<tr>
<td></td>
<td>ASAM [1]</td>
<td>192,832</td>
<td>1,370</td>
<td>150,528</td>
<td>·</td>
<td>976</td>
<td>·</td>
<td>·</td>
</tr>
<tr>
<td></td>
<td>LPS [43]</td>
<td>107,960</td>
<td>1,570</td>
<td>16,940</td>
<td>1,880</td>
<td>864</td>
<td>1,624,320</td>
<td>28,927</td>
</tr>
<tr>
<td></td>
<td>ZWSLW [30]</td>
<td>128,517</td>
<td>4,528</td>
<td>19,788</td>
<td>2,016</td>
<td>512</td>
<td>1,032,192</td>
<td>·</td>
</tr>
<tr>
<td>Regular</td>
<td>84,120</td>
<td>800</td>
<td>12,920</td>
<td>76</td>
<td>3,936</td>
<td>295,136</td>
<td>1,364</td>
<td></td>
</tr>
<tr>
<td>Shallow</td>
<td>81,312</td>
<td>800</td>
<td>12,240</td>
<td>40</td>
<td>6,368</td>
<td>254,720</td>
<td>978</td>
<td></td>
</tr>
<tr>
<td>Shallow/low</td>
<td>90,816</td>
<td>800</td>
<td>12,240</td>
<td>40</td>
<td>7,520</td>
<td>300,800</td>
<td>799</td>
<td></td>
</tr>
<tr>
<td>Regular</td>
<td>138,080</td>
<td>5128</td>
<td>22,380</td>
<td>2,022</td>
<td>640</td>
<td>1,294,080</td>
<td>·</td>
<td></td>
</tr>
<tr>
<td>Shallow</td>
<td>132,432</td>
<td>5128</td>
<td>28,080</td>
<td>30</td>
<td>8,096</td>
<td>388,608</td>
<td>955</td>
<td></td>
</tr>
<tr>
<td>Shallow/low</td>
<td>141,936</td>
<td>5128</td>
<td>28,080</td>
<td>30</td>
<td>10,000</td>
<td>380,000</td>
<td>769</td>
<td></td>
</tr>
</tbody>
</table>

| AES-192 | GLRS [30] | 189,432 | 1,608 | 172,032 | 11,088 | 1,112 | 12,329,856 | 96,956 |
|         | LPS [43] | 125,580 | 1,692 | 19,580 | 1,640 | 896 | 1,469,440 | 25,556 |
|         | ZWSLW [30] | 152,378 | 5,128 | 22,380 | 2,022 | 640 | 1,294,080 | ·          |
| Regular | 96,112 | 896 | 14,688 | 92 | 4,256 | 391,552 | 1,627 |
| Shallow | 92,856 | 896 | 14,008 | 48 | 6,688 | 320,124 | 1,174 |
| Shallow/low | 104,472 | 896 | 14,008 | 48 | 8,096 | 388,608 | 955 |
| Regular | 135,456 | 896 | 33,696 | 69 | 5,496 | 379,224 | 1,558 |
| Shallow | 151,360 | 896 | 32,136 | 36 | 9,168 | 330,048 | 1,138 |
| Shallow/low | 162,976 | 896 | 32,136 | 36 | 10,576 | 380,736 | 919 |

| AES-256 | GLRS [30] | 233,836 | 1,943 | 215,040 | 14,976 | 1,112 | 20,007,936 | 130,929 |
|         | LPS [43] | 151,011 | 1,992 | 23,760 | 2,160 | 896 | 2,661,120 | 33,525 |
|         | ZWSLW [30] | 177,645 | 5,128 | 26,774 | 2,022 | 640 | 1,760,256 | ·          |
| Regular | 117,704 | 1,103 | 18,088 | 108 | 4,576 | 1,760,256 | ·          |
| Shallow | 113,744 | 1,103 | 17,408 | 56 | 6,976 | 390,656 | 1,377 |
| Shallow/low | 127,472 | 1,103 | 17,408 | 56 | 8,640 | 483,840 | 1,118 |
| Regular | 193,248 | 1,103 | 41,496 | 81 | 5,176 | 471,096 | 1,826 |
| Shallow | 186,448 | 1,103 | 39,936 | 42 | 9,456 | 397,152 | 1,335 |
| Shallow/low | 200,176 | 1,103 | 39,936 | 42 | 11,120 | 467,040 | 1,076 |

☼: Using S-box with Toffoli depth 4.
⊛: Using S-box with Toffoli depth 3.

must be increased. To be modified, the number of qubits must be increased or the depth must be increased. Q#'s ResourcesEstimator tries to find its own lower bound for depth and qubit. That is, to achieve the qubits of the lower bound, the depth may have to be increased, and to achieve the depth of the lower bound, the qubits may have to be increased.

Another problem is that ancilla qubits allocated by using command are always prepared in a clean state. After S-box operation, ancilla qubits are not in a clean state (i.e. not all zeros), so they cannot be used in the next S-box as it is. However, the qubits allocated by the using command are always set to 0, the impossible S-box operation becomes possible. This is possible if new ancilla qubits are allocated for every S-box, but the qubits do not increase in resource estimation. This issue leads to returning a lower bound on the number of gates because the reverse operation is ignored. These issues are not problematic for single estimates (e.g., cost for one S-box). For this reason we only use a few of the results reported in [38] with caution. However, in the current version, these issues seem to be resolved.

These issues allow designing quantum circuit structures that are impossible. For example, encryption can be performed without a cleaning up operation in S-boxes. It seems difficult to estimate after solving the issues in [38], since apparently there is no quick fix and hence a considerable effort to change the design structure is to be made.

---

12https://github.com/microsoft/qsharp-runtime/pull/404.
Non-parallelizable SubBytes In their implementation, the S-box of [14] is adopted and ported to the quantum domain. The quantum resources required for the S-box quantum circuit reported in the Eurocrypt’20 paper [38, Table 1] are only correct for the stand-alone S-box (except for $T$-depth, this will be described in Section 4.1). However, in the case of SubBytes operating with 16 S-boxes, incorrect quantum resources are reported. This is a major cause of their resource estimation issues.

According to the reported number of required qubits, only one ancilla set is used in their SubBytes implementation. In other words, 16 S-boxes share one ancilla set. Thus, the arrangement of qubits in their SubBytes quantum circuit is the serial structure of Figure 2(a). Since 16 S-boxes generate each output using one ancilla set, all S-boxes in a limited space (one ancilla set) must be operated sequentially. However, in their report, the depth of the SubBytes is the same as the depth for a stand-alone S-box (meaning all S-boxes operate in parallel). That is, it is an impossible quantum circuit structure and the lower-bound depth is reported. The same error applies to the SubWord implementation of Key schedule.

Issue With AND Gate This issue is also found in their use of AND gates. Suppose that 5 Toffoli gates are operated in parallel during the Sbox process. Toffoli gates (method of [3]) operate in parallel without any additional work, providing one Toffoli depth and full depth for one Toffoli gate. On the other hand, in the AND gate of Figure 6(a), one ancilla qubit (bottom line) is used. Thus, if replaced with AND gates, 5 ancilla qubits for 5 AND gates must be allocated for parallel operation. Clearly, the ancilla qubit of the AND gate is initialized to 0 after operation and can be reused in the next AND gate, but a sequential operation is forced.

In a nutshell, in their S-box (out of 137 qubits, 136 qubits for the S-box and 1 qubit for the AND gate application), only one ancilla qubit is used for AND gates. However, quantum resources for parallel operations are reported.

Uninitialized Ancilla Qubits in SubBytes To generate the quantum S-box output for [15], an ancilla set consisting of 120 qubits is required. Ancilla qubits are responsible for storing the temp
values to compute the correct output for the input. After S-box is operated, these ancilla qubits still have their temp values stored. Thus, in order to reuse these ancilla qubits in the subsequent S-boxes, it is essential to initialize the stored temp values to 0. For this, reverse operation must be performed as shown in Figure 2(a) or 2(b).

However, in their quantum circuit implementation, initialization of qubits of ancilla set used in S-box is omitted. The authors say that they do not clean up ancilla qubits by not performing a reverse operation until the ciphertext is generated. This implementation is possible only when new clean ancilla qubits are allocated for every S-box used in encryption. Their intention is to reduce circuit depth, but we believe that omitting the reverse operation will not produce the correct ciphertext. Since the unclean ancilla qubits are reused in subsequent S-boxes, the correct output cannot be generated, so their implementation cannot be verified. Also, reduced quantum gates and depth due to the omitted reverse operations are estimated.

When we tested the SubWord quantum circuit for a 32-qubit input 0xffffffff by omitting the reverse operation and reusing the ancilla set, the output was 0x6a4e6216. Only the first S-box generates the correct output (0x16), and the subsequent S-boxes generate incorrect outputs. By analyzing their Q# source code, we think that this problem occurs probably because ancilla qubits are allocated by C#’s using command.

4.2 Corrected Report

To our understanding, some problems arise if the qubits are allocated by the using command in Q# (and it affects the non-linear components). However more experiments are to be carried out in order to be completely certain about it.

The using command automatically disposes when the function ends. If ancilla qubits to implement AES S-box are allocated with the using command, the consistency between depth and qubits is lost. When 16 S-boxes are executed in SubBytes, the ancilla qubits allocated by the using are counted only for the first S-box and not after. Also counts the depth for executing 16 S-boxes simultaneously. In order to derive the correct result, the number of qubits or depth must be increased. Q#’s ResourcesEstimator tries to find its own lower bound for depth and qubit. That is, to achieve the qubits of the lower bound, the depth may have to be increased, and to achieve the depth of the lower bound, the qubits may have to be increased.

Another problem is that ancilla qubits allocated by using command are always prepared in a clean state. After S-box operation, ancilla qubits are not in a clean state (i.e., not all zero), so they cannot be used in the next S-box as it is. However, the qubits allocated by the using command are always set to 0, the impossible S-box operation becomes possible. This is possible if new ancilla qubits are allocated for every S-box, but the qubits do not increase in resource estimation.

These issues allow designing quantum circuit structures that are impossible. In these issues, we estimate the corrected results from the results reported as lower-bound in their AES quantum circuit architecture. We contribute to three major modifications:

1. We reflect on the increasing depth in their number of qubits using only one ancilla set. As shown in Figure 2(a), since the ancilla set is shared, not only SubBytes but also S-boxes of SubWord of the key schedule are operated sequentially.
2. We modify the omitted reverse operations to be performed so that they produce the correct output, which further increases the depth.
3. We correct the implementation of MixColumns where the same issue occurs. In Eurocrypt’20 paper [38], two MixColumn implementations were presented. The in-place method of MixColumn implementation (which uses PLU decomposition, and derived by the authors themselves [38]) does not cause this issue. On the other hand, similar to S-box, the same issue applies to the
MixColumn implementation by Maximov [47], which requires ancilla qubits, so this is also solved in the same way as the S-box.

Table 5: Corrected benchmarks of JNRV (Eurocrypt’20) implementation of AES.

(a) AES-128 gate costs

<table>
<thead>
<tr>
<th>Method</th>
<th>S-box</th>
<th>MixColumns</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>In-place</td>
</tr>
<tr>
<td>#CNOT</td>
<td>654</td>
<td>1,108</td>
</tr>
<tr>
<td>#1qCliff</td>
<td>184</td>
<td>0</td>
</tr>
<tr>
<td>#T</td>
<td>136</td>
<td>0</td>
</tr>
<tr>
<td>#Measure</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>#T-depth</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>#qubits</td>
<td>137</td>
<td>128</td>
</tr>
<tr>
<td>Full depth</td>
<td>101</td>
<td>111</td>
</tr>
</tbody>
</table>

(b) Oracles

<table>
<thead>
<tr>
<th>Method</th>
<th>In-place MixColumn [38]</th>
<th>Maximov’s MixColumn [47]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AES-128</td>
<td>AES-192</td>
</tr>
<tr>
<td>#CNOT</td>
<td>292,313</td>
<td>329,697</td>
</tr>
<tr>
<td>#1qCliff</td>
<td>84,428</td>
<td>94,316</td>
</tr>
<tr>
<td>#T</td>
<td>54,908</td>
<td>61,436</td>
</tr>
<tr>
<td>#Measure</td>
<td>13,727</td>
<td>15,359</td>
</tr>
<tr>
<td>#T-depth</td>
<td>121</td>
<td>120</td>
</tr>
<tr>
<td>#qubits</td>
<td>1,665</td>
<td>1,985</td>
</tr>
<tr>
<td>Full depth</td>
<td>2,816</td>
<td>2,978</td>
</tr>
</tbody>
</table>

(c) AES-128 Modules

<table>
<thead>
<tr>
<th>Method</th>
<th>#CNOT</th>
<th>#1qCliff</th>
<th>#T</th>
<th>Toffoli depth</th>
<th>#qubits</th>
<th>Full depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>SubBytes</td>
<td>12,000</td>
<td>1,220</td>
<td>7,328</td>
<td>192</td>
<td>376</td>
<td>2,672</td>
</tr>
<tr>
<td>Key schedule</td>
<td>3,096</td>
<td>355</td>
<td>1,832</td>
<td>48</td>
<td>248</td>
<td>669</td>
</tr>
<tr>
<td>MixColumns (Maximov [47])</td>
<td>1,248</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>318</td>
<td>88</td>
</tr>
<tr>
<td>One round$^*$</td>
<td>16,472</td>
<td>1,507</td>
<td>9,160</td>
<td>240</td>
<td>632</td>
<td>3,417</td>
</tr>
</tbody>
</table>

$^*$: One typical round (that includes MixColumn).

(d) Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>#CNOT</th>
<th>#1qCliff</th>
<th>#T</th>
<th>Toffoli depth</th>
<th>#qubits</th>
<th>Full depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES-128$^*$</td>
<td>161,982</td>
<td>14,400</td>
<td>91,380</td>
<td>2,394</td>
<td>1,656</td>
<td>33,320</td>
</tr>
<tr>
<td>AES-192$^*$</td>
<td>182,774</td>
<td>16,128</td>
<td>102,372</td>
<td>2,682</td>
<td>1,976</td>
<td>37,328</td>
</tr>
<tr>
<td>AES-256$^*$</td>
<td>224,214</td>
<td>19,871</td>
<td>126,188</td>
<td>3,306</td>
<td>2,296</td>
<td>46,012</td>
</tr>
<tr>
<td>AES-128$^+$</td>
<td>163,242</td>
<td>14,994</td>
<td>91,380</td>
<td>2,394</td>
<td>2,808</td>
<td>33,914</td>
</tr>
<tr>
<td>AES-192$^+$</td>
<td>184,314</td>
<td>16,854</td>
<td>102,372</td>
<td>2,682</td>
<td>3,384</td>
<td>38,054</td>
</tr>
<tr>
<td>AES-256$^+$</td>
<td>226,034</td>
<td>20,729</td>
<td>126,188</td>
<td>3,306</td>
<td>3,960</td>
<td>46,870</td>
</tr>
</tbody>
</table>

$^+$: Using in-place MixColumn [38].

$^*$: Using Maximov’s MixColumn [47].

We modified from the quantum circuit base of [38] and implemented it on ProjectQ. Our source code generates the correct ciphertext and the correct resources are estimated. To avoid confusion, we estimate quantum resources using Toffoli gates (using the method from [3]), rather than applying AND gates with issues.
Results with bug-fixed Eurocrypt’20 implementation can be found in Table 5. Table 5(a) shows quantum resources for S-box and MixColumns reported in the Eurocrypt’20 paper. Quantum resources in Table 5(a) include cleaning up of used ancilla qubits. Table 5(b) shows the quantum resources for AES oracles reported in the Eurocrypt’20 paper. Quantum resources are reported for an oracle rather than a single AES quantum circuit. In oracle, since the AES quantum circuit operates twice, the estimation of quantum resources for a single AES quantum circuit can be counted in half except for the number of qubits in Table 5(b). Table 5(c) shows the estimated resources for SubBytes, key schedule, MixColumns, and one round where the issue occurs. The difference for the corrected MixColumns is relatively small, but the depth estimated as lower-bound for SubBytes is corrected high. The resources estimated in Table 5(c) include a reverse operation to clean ancilla qubits. Table 5(d) shows the corrected quantum resources for AES quantum circuits, and it is confirmed that the depth increases significantly when maintaining the number of qubits.

5 Performance of AES Quantum Circuits

In this part, we present the performance of our implementations of AES quantum circuits. We use the open-source quantum programming tool IBM ProjectQ to implement and simulate the quantum circuits. An internal library, ClassicalSimulator, simulates quantum circuits and verifies test vectors. Quantum resources required to implement quantum circuits are estimated using another library, ResourceCounter.

Table 4 shows the quantum resources required to implement our AES quantum circuits and previous AES quantum circuits. Although various decompositions exist in the Toffoli gate, Table 4 enables consistent comparison with NCT (NOT, CNOT, Toffoli) level analysis. In [1,30], Itoh–Tsujii-based inversion is implemented on a quantum circuit, so many resources are used for SubBytes. In [43,59], more efficient quantum circuits are implemented by extending the S-box of [14], but the circuit depth is increased due to the serial execution of S-boxes by concentrating on saving qubits. On the other hand, our implementation focuses on minimizing circuit depth while considering the trade-offs for using qubits. In [59], \#TD \times \#M (where \#TD is the Toffoli depth and \#M is the number of qubits ) is used to measure the trade-off of quantum circuits. In this work, all AES quantum circuits with reduced depth and quantum gates using a reasonable number of qubits offer the best trade-off. In [38], the quantum resources required to implement quantum circuits for AES were also estimated. However, there seem to be some issues with Q#'s ResourcesEstimator used in their work, especially in implementing quantum circuits for SubBytes. Therefore, the results of [38] are not used here. Following [59, Table 10], Table 6 shows detailed quantum resources by decomposing Toffoli gates for the AES quantum circuits implemented in this work. The Toffoli gate is decomposed into 7 T gates + 8 Clifford gates, a T depth of 4, and a full depth of 8 according to one of the methods (described in Section 3.2) in [3].

6 Quantum Key Search

In this part, the corresponding costs for applying Grover’s search algorithm to exhaustive key search are estimated based on the proposed quantum circuits for the three variants of AES. We estimate the cost of oracle, which accounts for the largest portion of Grover’s search algorithm. The overhead for diffusion operator is negligible compared to oracle and is not difficult to implement. For this reason, it is common to estimate the cost for oracle excluding the diffusion operator [5,30,43]. In the oracle, the target cipher’s quantum circuit encrypts a known plaintext with the key in the

Table 6: Quantum resources required for variants of AES (this work).

<table>
<thead>
<tr>
<th>Cipher</th>
<th>#CNOT</th>
<th>#1qCliff</th>
<th>#T</th>
<th>T-depth</th>
<th>#qubits</th>
<th>Full depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES-128(^{\circledast})</td>
<td>161,640</td>
<td>14,400</td>
<td>90,440</td>
<td>304</td>
<td>3,936</td>
<td>1,364</td>
</tr>
<tr>
<td>AES-128(^{\circledast\circledast})</td>
<td>154,752</td>
<td>14,400</td>
<td>85,680</td>
<td>160</td>
<td>6,368</td>
<td>978</td>
</tr>
<tr>
<td>AES-128(^{*})</td>
<td>164,256</td>
<td>16,832</td>
<td>85,680</td>
<td>160</td>
<td>7,520</td>
<td>799</td>
</tr>
<tr>
<td>AES-128(^{\circledast\circledast})</td>
<td>315,920</td>
<td>32,000</td>
<td>207,480</td>
<td>228</td>
<td>5,176</td>
<td>1,307</td>
</tr>
<tr>
<td>AES-128(^{\circledast\circledast})</td>
<td>300,912</td>
<td>32,000</td>
<td>196,560</td>
<td>120</td>
<td>8,848</td>
<td>948</td>
</tr>
<tr>
<td>AES-128(^{*})</td>
<td>310,416</td>
<td>33,248</td>
<td>196,560</td>
<td>120</td>
<td>10,000</td>
<td>769</td>
</tr>
<tr>
<td>AES-192(^{\circledast})</td>
<td>184,240</td>
<td>16,400</td>
<td>102,816</td>
<td>368</td>
<td>4,256</td>
<td>1,627</td>
</tr>
<tr>
<td>AES-192(^{\circledast\circledast})</td>
<td>176,904</td>
<td>16,400</td>
<td>98,056</td>
<td>192</td>
<td>6,688</td>
<td>1,174</td>
</tr>
<tr>
<td>AES-192(^{*})</td>
<td>188,520</td>
<td>19,440</td>
<td>98,056</td>
<td>192</td>
<td>8,096</td>
<td>955</td>
</tr>
<tr>
<td>AES-192(^{\circledast\circledast})</td>
<td>359,632</td>
<td>36,464</td>
<td>235,872</td>
<td>276</td>
<td>5,496</td>
<td>1,558</td>
</tr>
<tr>
<td>AES-192(^{\circledast\circledast})</td>
<td>344,176</td>
<td>36,464</td>
<td>224,952</td>
<td>144</td>
<td>9,168</td>
<td>1,138</td>
</tr>
<tr>
<td>AES-192(^{*})</td>
<td>355,792</td>
<td>38,024</td>
<td>224,952</td>
<td>144</td>
<td>10,576</td>
<td>919</td>
</tr>
<tr>
<td>AES-256(^{\circledast})</td>
<td>226,232</td>
<td>19,871</td>
<td>126,616</td>
<td>432</td>
<td>4,576</td>
<td>1,907</td>
</tr>
<tr>
<td>AES-256(^{\circledast\circledast})</td>
<td>218,192</td>
<td>19,871</td>
<td>121,856</td>
<td>224</td>
<td>6,976</td>
<td>1,377</td>
</tr>
<tr>
<td>AES-256(^{*})</td>
<td>231,920</td>
<td>23,519</td>
<td>121,856</td>
<td>224</td>
<td>8,640</td>
<td>1,118</td>
</tr>
<tr>
<td>AES-256(^{\circledast\circledast})</td>
<td>442,224</td>
<td>44,159</td>
<td>290,472</td>
<td>324</td>
<td>5,816</td>
<td>1,826</td>
</tr>
<tr>
<td>AES-256(^{\circledast\circledast})</td>
<td>426,064</td>
<td>44,159</td>
<td>279,552</td>
<td>168</td>
<td>9,456</td>
<td>1,335</td>
</tr>
<tr>
<td>AES-256(^{*})</td>
<td>439,792</td>
<td>46,031</td>
<td>279,552</td>
<td>168</td>
<td>11,120</td>
<td>1,076</td>
</tr>
</tbody>
</table>

\(\circledast\): Regular version (using MixColumn from [56]).
\(\circledast\circledast\): Shallow version (using MixColumn from [56]).
\(*\): Shallow/low depth version (using MixColumn from [44]).

Table 7: Quantum resources required for exhaustive key search for AES (this work).

<table>
<thead>
<tr>
<th>Cipher</th>
<th>(r)</th>
<th>#qubits</th>
<th>Total gates</th>
<th>Total depth</th>
<th>Search complexity</th>
<th>NIST security</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES-128(^{\circledast})</td>
<td>1</td>
<td>3,937</td>
<td>1.597 \cdot 2^{32}</td>
<td>1.046 \cdot 2^{35}</td>
<td>1.671 \cdot 2^{157}</td>
<td>Not achieved (&lt; (2^{176}))</td>
</tr>
<tr>
<td>AES-128(^{\circledast\circledast})</td>
<td>6,369</td>
<td>1.527 \cdot 2^{32}</td>
<td>1.501 \cdot 2^{34}</td>
<td>1.146 \cdot 2^{157}</td>
<td>1.96 \cdot 2^{156}</td>
<td></td>
</tr>
<tr>
<td>AES-128(^{*})</td>
<td>7,521</td>
<td>1.599 \cdot 2^{32}</td>
<td>1.226 \cdot 2^{34}</td>
<td>1.917 \cdot 2^{222}</td>
<td>Level-1 (&lt; (2^{233}))</td>
<td></td>
</tr>
<tr>
<td>AES-192(^{\circledast})</td>
<td>2</td>
<td>8,321</td>
<td>1.536 \cdot 2^{415}</td>
<td>1.248 \cdot 2^{107}</td>
<td>1.318 \cdot 2^{222}</td>
<td>Level-1 (&lt; (2^{233}))</td>
</tr>
<tr>
<td>AES-192(^{\circledast\circledast})</td>
<td>13,185</td>
<td>1.464 \cdot 2^{415}</td>
<td>1.801 \cdot 2^{406}</td>
<td>1.136 \cdot 2^{222}</td>
<td>Level-3 (&lt; (2^{298}))</td>
<td></td>
</tr>
<tr>
<td>AES-192(^{*})</td>
<td>16,001</td>
<td>1.551 \cdot 2^{415}</td>
<td>1.465 \cdot 2^{406}</td>
<td>1.314 \cdot 2^{286}</td>
<td>1.816 \cdot 2^{286}</td>
<td></td>
</tr>
<tr>
<td>AES-256(^{\circledast})</td>
<td>8,897</td>
<td>1.797 \cdot 2^{417}</td>
<td>1.463 \cdot 2^{139}</td>
<td>Level-1 (&lt; (2^{233}))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AES-256(^{\circledast\circledast})</td>
<td>13,697</td>
<td>1.720 \cdot 2^{417}</td>
<td>1.056 \cdot 2^{139}</td>
<td>Level-3 (&lt; (2^{298}))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AES-256(^{*})</td>
<td>17,025</td>
<td>1.824 \cdot 2^{417}</td>
<td>1.715 \cdot 2^{138}</td>
<td>1.564 \cdot 2^{286}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(\circledast\): Regular version (using MixColumn from [56]).
\(\circledast\circledast\): Shallow version (using MixColumn from [56]).
\(*\): Shallow/low depth version (using MixColumn from [44]).

The generated ciphertext in the superposition state is compared with the known ciphertext and a reverse operation is performed for Grover’s iterations. For comparison, an \(n\)-multi controlled NOT gate is used to check that the generated ciphertext (\(n\)-qubit) is a known ciphertext. This occupies a small part in the oracle, and since the main part is a block cipher’s quantum circuit, the cost for an \(n\)-multi controlled NOT gate is omitted for simplicity of analysis.

In quantum exhaustive key search, to recover a unique key, not a spurious key, Grassl et al. in [30] estimated the attack cost for \(r\) known (plaintext, ciphertext) pairs (\(r = 3\), \(r = 4\) and \(r = 5\), respectively). Later in [43], Langenberg et al. explained that \(r = \lceil k/n \rceil\) (key size/block size) is sufficient to successfully recover a unique key. The authors in [38] also estimated the cost for the same \(r\) (plaintext, ciphertext) pairs in [43] through detailed computations. Following this approach,
we also estimate the cost of recovering a unique key for \( r = \lceil k/n \rceil \) (plaintext, ciphertext) pairs. When \( r = 1 \), the target block cipher quantum circuit is serially executed twice in oracle. Thus, the cost of oracle is twice that required to implement a quantum circuit, excluding qubits. When \( r \geq 2 \), \( r \) target block quantum circuits are executed twice in parallel, and the following should be considered in cost estimation. Although \( r \geq 2 \) plaintexts are used, only one input key is used, so the cost for key schedule should be estimated only once. Finally, the cost of quantum exhaustive key search for the target block cipher is roughly the cost of oracle \( \times \lfloor \frac{k}{4} \sqrt{2^k} \rfloor \) (where \( k \) is the key size). Costs are estimated at the \( T + \) Clifford level and computed as total gates \( \times \) total depth. We show the cost of quantum exhaustive key search for AES (AES-128, AES-192, AES-256 using S-box with Toffoli depth of 4) in Table 7, respectively. Additionally, we evaluate the post-quantum security strength for symmetric key cryptography presented by NIST based on the cost of Grover’s key search [49]. NIST estimated the security strength based on the cost of Grover’s key search for AES estimated by Grassl et al. [30]. Level-1 is the cost of Grover’s key search for AES-128 (\( 2^{170} \)), and Level-3 and Level-5 are costs for AES-192 (\( 2^{233} \)) and AES-256 (\( 2^{298} \)), respectively. The number of qubits is not included in NIST’s estimation. NIST is focusing more on gates and depths that increase dramatically with Grover iterations. The quantum exhaustive search cost for the AES family estimated in this paper is much reduced from the cost previously estimated by NIST based on Grassl et al.’s work [30].

NIST recommends that submitters meet the requirements for Levels 1, 2 and/or 3 and states that this will provide sufficient security for the foreseeable future post-quantum era [49]. Now, we compare the cost of quantum exhaustive key search of AES and LowMC in this work with NIST security. In the case of 128-bit key, security is not achieved in all ciphers, and in the case of 192-bit and 256-bit key, security is achieved one level lower. However, it should be pointed out that the estimated costs (from the work of Grassl et al. [30]) for the levels (level 1: \( 2^{170} \), level 3: \( 2^{233} \), and level 5: \( 2^{298} \)) are considerably high. This is evident from the significantly reduced cost of most recent quantum implementations of ciphers [7,11,27,36,37,38,43,59]. By current NIST estimates (too conservative), most ciphers do not achieve their post-quantum security corresponding to their key size. NIST noted that these preliminary classifications should be evaluated conservatively if the cost of best known attacks is significantly reduced. Also, the specific cost is defined based on Grassl’s AES quantum circuit, but the security level is defined according to the relative attack cost for AES. Based on our validated AES quantum circuit, the estimated cost of the attack is the lowest and is much lower than the NIST estimate.

7 Conclusion

In this work, we collate multiple contributions reported in the last couple of years, together with up-to-date improvements in the quantum technology as well as optimizations on the building blocks of the ciphers. Among other results, we show the least Toffoli depth and full depth implementations of all variants of AES (more than 96% improvement from [59]) as well as least qubit count-Toffoli depth product (more than 70% from the same paper). A quick comparison of NIST’s security level (under Grover’s search) of our work together with [43] is given in Table 8.

Finding optimizations for the cipher building blocks can be considered among the top priorities for the future research works. As far as we can tell, there is a vacant niche for a tool that can efficiently find such implementation for \( 8 \times 8 \) S-boxes. The tools described in [18,19,20] can possibly be considered as starting points.
Table 8: Comparison of NIST security levels for variants of AES.

<table>
<thead>
<tr>
<th>NIST security</th>
<th>LPS [43]</th>
<th>This work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not achieved</td>
<td>AES-128</td>
<td>$2^{162}$</td>
</tr>
<tr>
<td>(≤ $2^{170}$)</td>
<td></td>
<td>$2^{157}$</td>
</tr>
<tr>
<td>Level-1</td>
<td>AES-192</td>
<td>$2^{217}$</td>
</tr>
<tr>
<td>(≥ $2^{170}$)</td>
<td></td>
<td>$2^{222}$</td>
</tr>
<tr>
<td>Level-3</td>
<td>AES-256</td>
<td>$2^{266}$</td>
</tr>
<tr>
<td>(≥ $2^{233}$)</td>
<td></td>
<td>$2^{286}$</td>
</tr>
</tbody>
</table>

We could find two other papers with new MixColumn implementations. First, the authors of [8] presented two implementations (103 XOR/3 classical depth, and 95 XOR/6 classical depth)\textsuperscript{14}; however we could not verify the results (probably due to an encoding issue). Second, an implementation of 108 XOR count is mentioned in [26, Footnote 3/Page 42], but it is not clear to us so far. These can be considered in a future work.

References


\textsuperscript{14}If taken as-is, the 103 XOR/3 classical depth implementation yields 206 #CNOT gates, 135 #qubits, with 11 quantum depth when ported. Thus, it is in theory possible to (marginally) improve our shallow/low depth version by switching to this implementation.


A Concise Description of AES Variants

The Advanced Encryption Standard (AES) [17] is an SPN block cipher family with a block of 128 bits. The state of AES is arranged as a $4 \times 4$ matrix of bytes. AES contains three specific variants denoted as AES-128, AES-192 and AES-256 according to the key size. Schematic diagrams of AES-128 round function and key schedule can be found in Figure 7.
**Round Function** The round function of AES consists of AddRoundKey \(\circ\) MixColumns \(\circ\) ShiftRows \(\circ\) SubBytes, except for the last round which misses the MixColumns operation.

*SubBytes.* This operation substitutes each element by a predefined \(8 \times 8\) S-box.

*ShiftRows.* This operation cyclically rotates the \(r^{th}\) row of state to the left by \(i\) places, for \(i = 0, 1, 2, 3\).

*MixColumns.* The MixColumn operation pre-multiplies each of the state column with the right circulant matrix \((0x02, 0x03, 0x01, 0x01)\), over \(GF(2^8)[x]\) with modulus \(x^4 + 1\).

*AddRoundKey.* The sub-key of each round is generated by the Key Expansion algorithm. Each call of AddRoundKey XORs the 128-bit sub-key to the state.

The encryption procedure for different instances of AES family are somewhat similar, except the number of round varies. For AES-128, AES-192 and AES-256, the round numbers are 10, 12, 14 respectively and all round functions are identical except that there is no MixColumns operation in the last round. Note that there is an extra key addition before the first round (also known as whitening).

**Key Schedule** Similar to the state, the master key of AES is allocated to a \(4 \times l\) grid of byte in order, where \(l = 4, 6\) or 8 for AES-128, AES-192 and AES-256, respectively. Generally, the generation of the round sub-keys are based on *word* (the entire column in the grid) with the operations RotWord (cyclically rotating the bytes in a word to the left by one byte), SubWord (operating the SubBytes of round function on each bytes in a word) and the XOR of Rcon\(\left[\frac{r}{4}\right]\) (the \(r^{th}\) 32-bit round constant).

The master key is loaded to the grid \(W_0, W_1, \cdots, W_i\); where \(i\) is 3, 5 and 7 for AES-128, AES-192 and AES-256 respectively. In order to guarantee the encryption, 40, 46 and 52 words need to be provided by key expansion for those three AES instances, respectively.

For AES-128, the word \(W_i\) is generated by

\[
W_i = \begin{cases} 
W_{i-4} \oplus \text{SubWord}(\text{RotWord}(W_{i-1})) \oplus \text{Rcon}[i/4], & \text{if } i \equiv 0 \mod 4, \\
W_{i-4} \oplus W_{i-1}, & \text{otherwise},
\end{cases}
\]

where \(i = 4, 5, \cdots, 43\).

For AES-192, the word \(W_i\) is generated by

\[
W_i = \begin{cases} 
W_{i-6} \oplus \text{SubWord}(\text{RotWord}(W_{i-1})) \oplus \text{Rcon}[i/6], & \text{if } i \equiv 0 \mod 6, \\
W_{i-6} \oplus W_{i-1}, & \text{otherwise},
\end{cases}
\]

where \(i = 6, 7, \cdots, 51\).

For AES-256, the word \(W_i\) is generated by

\[
W_i = \begin{cases} 
W_{i-8} \oplus \text{SubWord}(\text{RotWord}(W_{i-1})) \oplus \text{Rcon}[i/8], & \text{if } i \equiv 0 \mod 8, \\
W_{i-8} \oplus \text{SubWord}(W_{i-1}), & \text{if } i \equiv 4 \mod 8, \\
W_{i-8} \oplus W_{i-1}, & \text{otherwise},
\end{cases}
\]

where \(i = 8, 9, \cdots, 59\).
(a) Round function of encryption (except last round which skips MixColumns).

(b) Key schedule for encryption.

Fig. 7: Schematic of AES construction.

Notes

**Singular and Plural Forms** The AES state is represented as a $4 \times 4$ matrix and the operation on one column of the matrix is denoted here as MixColumn. As described earlier, MixColumn corresponds to a matrix multiplication over $\text{GF}(2^8)$, which can equivalently be expressed as multiplication by a matrix of dimension $32 \times 32$ over $\mathbb{F}_2$. In the AES round function, the MixColumns operates on the whole block by applying MixColumn to every four bytes in the state (i.e., one column in the $4 \times 4$ matrix). Thus, one MixColumns operation is equivalent to $4 \times \text{MixColumn}$ operations on different columns in the matrix. Denoting the binary matrix corresponding to MixColumn as $M$ with size $32 \times 32$, MixColumns can be represented as the diagonal matrix $(M, M, M, M)$ of dimension $128 \times 128$ over $\mathbb{F}_2$.

The bytes in each row of the matrix will be cyclically shifted to the left in each round and the shift operation on the bytes in one row is denoted here as ShiftRow, in the step of ShiftRows, the ShiftRow will be operated on all the rows in the matrix and shift the bytes in the $i$th row to the left by $i$ bytes, where $i = 1, 2, 3$. Thus, one ShiftRows operation is equivalent to $4 \times \text{ShiftRow}$ operations on different rows in the $4 \times 4$ matrix with the shift parameter varies from 0 to 3.
Different from MixColumns and ShiftRows, the SubBytes in the round function updates every byte in the $4 \times 4$ matrix in the same way. The process of applying the S-box to one byte in the AES state is denoted here as SubByte. In each round, the SubBytes updates all the bytes in the $4 \times 4$ matrix by replacing each byte by another one according to the predefined nonlinear map. Thus, one SubBytes operation is equivalent to 16 SubByte operations on the bytes of the $4 \times 4$ matrix.

**S-box and S-box$^\dagger$ in Quantum** S-box in quantum denotes before storing values from ancilla qubits to output qubits. Denote the reverse operation of S-box as S-box$^\dagger$ and uses input qubits to clean up ancilla qubits.

**SubBytes and SubBytes$^\dagger$ in Quantum** SubBytes (of AES) in quantum denotes parallel operation for 16 S-boxes. Denote the reverse operation of SubBytes as SubBytes$^\dagger$ and cleans up all used ancilla qubits in 16 S-boxes.

**Rotation and Rotation$^\dagger$ in Quantum** Rotation (of AES) in quantum denotes the same RotWord. The reverse operation of Rotation is denoted as Rotation$^\dagger$.

**SubWord and SubWord$^\dagger$ in Quantum** SubWord (of AES) in quantum denotes parallel operation for 4 S-boxes. We denote the reverse operation of SubWord as SubWord$^\dagger$ (and clean up all used ancilla qubits in 4 S-boxes).

### B Further Details on Implementation and Result

Codes 1.1, 1.2 and 1.3 show our implementation of the AES S-box and two implementations the AESMixColumn in ProjectQ, respectively. The variable `resource_check` in Code 1.1 is set to 0 when using ClassicalSimulator to verify implementation suitability, and when using ResourceCounter, it is set to 1 to decompose Toffoli gates to estimate detailed quantum resources. Code 1.2 and 1.3 respectively correspond to the implementations reported in [56] and [46]. Similar to [59, Table 6], we show the per-round benchmark for our implementations of the AES family in Table 9.

### C Depth of Sequential XOR: Classical vs. Quantum

One may note from Table 3 that the depth for quantum circuit corresponding to the implementation by [56] is 30, whereas the same for the classical circuit is 6. Although this implementation operates in-place, it still reuses one variable multiple times. In other words, the same variable appears multiple times in the right hand side. For example, one may check that $x_{31}$ appears more than once: $x_{16} \leftarrow x_{16} \oplus x_{31}$ (Line 15), $x_4 \leftarrow x_4 \oplus x_{31}$ (Line 29), $x_0 \leftarrow x_0 \oplus x_{31}$ (Line 56), and so on. This does not account for extra depth in a classical circuit (as multiple fan-outs are allowed). However, in a quantum circuit where there is exactly one fan-out, this situation causes increase of depth.
Code 1.1: Quantum circuit for AES S-box

def Sbox(eng, u, t, m, l, s, round_number, resource_check):
    with Compute(eng):
        CNOT2(eng, u[7], u[4], t[0]); CNOT2(eng, u[7], u[2], t[1])
        CNOT2(eng, u[7], u[1], t[2]); CNOT2(eng, u[4], u[2], t[3])
        CNOT2(eng, u[3], u[1], t[4]); CNOT2(eng, t[0], t[4], t[5])
        CNOT2(eng, u[6], u[5], t[6]); CNOT2(eng, u[0], t[6], t[7])
        CNOT2(eng, u[0], t[6], t[8]); CNOT2(eng, t[5], t[6], t[9])
        CNOT2(eng, u[6], u[2], t[10]); CNOT2(eng, u[5], u[2], t[11])
        CNOT2(eng, t[0], t[12], t[13]); CNOT2(eng, u[5], t[10], t[14])
        CNOT2(eng, t[1], t[10], t[14]); CNOT2(eng, t[4], t[11], t[15])
        CNOT2(eng, t[0], t[15], t[16]); CNOT2(eng, t[5], t[10], t[16])
        CNOT2(eng, u[6], u[2], t[19], t[20]); CNOT2(eng, t[5], t[18], t[20])
        CNOT2(eng, t[1], t[21], t[22]); CNOT2(eng, t[1], t[9], t[23])
        CNOT2(eng, t[19], t[16], t[24]); CNOT2(eng, t[2], t[15], t[25])
        CNOT2(eng, t[0], t[11], t[26])
        Toffoli_gate(eng, t[12], t[5], m[0], resource_check)
        Toffoli_gate(eng, t[22], t[7], m[1], resource_check)
        CNOT2(eng, t[13], m[0], m[2])
        Toffoli_gate(eng, t[24], u[0], m[3], resource_check)
        CNOT2(eng, m[3], m[0], m[4])
        Toffoli_gate(eng, t[2], t[16], m[5], resource_check)
        Toffoli_gate(eng, t[21], t[8], m[6], resource_check)
        CNOT2(eng, t[26], m[5], m[7])
        Toffoli_gate(eng, t[19], t[16], m[8], resource_check)
        Toffoli_gate(eng, m[5], m[6], m[5])
        Toffoli_gate(eng, t[0], t[14], m[10], resource_check)
        Toffoli_gate(eng, t[3], t[26], m[11], resource_check)
        CNOT2(eng, m[11], m[10], m[12])
        Toffoli_gate(eng, t[1], t[9], m[13], resource_check)
        CNOT2(eng, m[13], m[10], m[14]); CNOT2(eng, m[3], m[1], m[15])
        CNOT2(eng, m[13], t[23], m[16]); CNOT2(eng, m[7], m[5], m[17])
        CNOT2(eng, m[9], m[14], m[18]); CNOT2(eng, m[16], m[12], m[19])
        CNOT2(eng, m[16], m[14], m[20]); CNOT2(eng, m[17], m[12], m[21])
        CNOT2(eng, m[18], t[24], m[22]); CNOT2(eng, m[21], m[22], m[23])
        Toffoli_gate(eng, m[21], m[19], m[24], resource_check)
        CNOT2(eng, m[20], m[24], m[26]); CNOT2(eng, m[19], m[20], m[26])
        CNOT2(eng, m[22], m[24], m[27])
        Toffoli_gate(eng, m[27], m[28], m[26], resource_check)
        Toffoli_gate(eng, m[28], m[23], m[29], resource_check)
        Toffoli_gate(eng, m[19], m[22], m[30], resource_check)
        Toffoli_gate(eng, m[20], m[30], m[31], resource_check)
        CNOT2(eng, m[26], m[24], m[29])
        Toffoli_gate(eng, m[20], m[21], m[33], resource_check)
        Toffoli_gate(eng, m[29], m[33], m[34], resource_check)
        CNOT2(eng, m[23], m[24], m[35]); CNOT2(eng, m[20], m[28], m[36])
        CNOT2(eng, m[31], m[32], m[37]); CNOT2(eng, m[22], m[29], m[38])
        CNOT2(eng, m[34], m[39], m[39]); CNOT2(eng, m[37], m[39], m[40])
        CNOT2(eng, m[36], m[39], m[43]); CNOT2(eng, m[36], m[37], m[42])
        CNOT2(eng, m[38], m[39], m[43]); CNOT2(eng, m[41], m[40], m[44])
        Toffoli_gate(eng, m[43], t[6], m[48], resource_check)
        Toffoli_gate(eng, m[39], t[7], m[46], resource_check)
        Toffoli_gate(eng, m[38], u[0], m[47], resource_check)
        Toffoli_gate(eng, m[43], t[6], m[48], resource_check)
Toffoli_gate(eng, m[42], t[15], m[48], resource_check)
Toffoli_gate(eng, m[37], t[8], m[49], resource_check)
Toffoli_gate(eng, m[36], t[16], m[50], resource_check)
Toffoli_gate(eng, m[41], t[14], m[51], resource_check)
Toffoli_gate(eng, m[44], t[26], m[52], resource_check)
Toffoli_gate(eng, m[40], t[9], m[53], resource_check)
Toffoli_gate(eng, m[43], t[12], m[54], resource_check)
Toffoli_gate(eng, m[39], t[22], m[55], resource_check)
Toffoli_gate(eng, m[38], t[18], m[56], resource_check)
Toffoli_gate(eng, m[42], t[2], m[57], resource_check)
Toffoli_gate(eng, m[37], t[21], m[58], resource_check)
Toffoli_gate(eng, m[36], t[19], m[59], resource_check)
Toffoli_gate(eng, m[41], t[0], m[60], resource_check)
Toffoli_gate(eng, m[44], t[3], m[61], resource_check)
Toffoli_gate(eng, m[40], t[1], m[62], resource_check)

CNOT2(eng, m[60], m[61], l[0]); CNOT2(eng, m[49], m[55], l[1])
CNOT2(eng, m[45], m[47], l[2]); CNOT2(eng, m[46], m[54], l[3])
CNOT2(eng, m[53], m[57], l[4]); CNOT2(eng, m[48], m[60], l[5])
CNOT2(eng, m[61], l[5], l[6]); CNOT2(eng, m[45], l[3], l[7])
CNOT2(eng, m[50], m[58], l[8]); CNOT2(eng, m[51], m[52], l[9])
CNOT2(eng, m[52], m[58], l[10]); CNOT2(eng, m[59], l[12], l[11])
CNOT2(eng, m[47], m[50], l[11]); CNOT2(eng, m[49], l[0], l[13])
CNOT2(eng, m[61], m[60], l[14]); CNOT2(eng, m[54], l[3], l[15])
CNOT2(eng, m[53], m[57], l[16]); CNOT2(eng, m[56], l[1], l[17])
CNOT2(eng, m[57], l[18], l[19]); CNOT2(eng, m[52], l[4], l[19])
CNOT2(eng, l[0], l[1], l[20]); CNOT2(eng, l[1], l[7], l[21])
CNOT2(eng, l[1], l[12], l[22]); CNOT2(eng, l[18], l[2], l[23])
CNOT2(eng, l[18], l[19], l[24]); CNOT2(eng, l[6], l[10], l[25])
CNOT2(eng, l[1], l[6], l[26]); CNOT2(eng, l[6], l[10], l[27])
CNOT2(eng, l[13], l[14], l[29]); CNOT2(eng, l[11], l[17], l[29])

# Reverse (cleaning)
if (round_number != last_round):
  Uncompute(eng)
return s

def CNOT2(eng, a, b, c):
  CNOT | (a, c); CNOT | (b, c)

def Toffoli_gate(eng, a, b, c, resource_check):
  if (resource_check):
    Tdag | a
    Tdag | b
    H | c
    CNOT | (c, a)
    T | a
    CNOT | (b, c)
    CNOT | (b, a)
    T | c
    Tdag | a
    CNOT | (b, c)
    CNOT | (c, a)
    T | a
    Tdag | c
    CNOT | (b, a)
    H | c
  else:
    Toffoli | (a, b, c)
def Mixcolumn(eng, x_in):
    
    # Changing the index of qubits
    x = []
    for i in range(8):
        x.append(x_in[24 + i])
    for i in range(8):
        x.append(x_in[16 + i])
    for i in range(8):
        x.append(x_in[8 + i])
    for i in range(8):
        x.append(x_in[i])

    CNOT | (x[31], x[23]); CNOT | (x[21], x[31])
    CNOT | (x[10], x[12]); CNOT | (x[21], x[10])
    CNOT | (x[4], x[12]); CNOT | (x[21], x[4])
    CNOT | (x[8], x[9]); CNOT | (x[15], x[24])
    CNOT | (x[23], x[15]); CNOT | (x[16], x[17])
    CNOT | (x[9], x[10]); CNOT | (x[31], x[9])
    CNOT | (x[17], x[18]); CNOT | (x[26], x[17])
    CNOT | (x[17], x[11]); CNOT | (x[24], x[25])
    CNOT | (x[8], x[9]); CNOT | (x[18], x[24])
    CNOT | (x[15], x[16]); CNOT | (x[0], x[18])
    CNOT | (x[23], x[16]); CNOT | (x[16], x[17])
    CNOT | (x[0], x[16]); CNOT | (x[31], x[0])
    CNOT | (x[23], x[16]); CNOT | (x[0], x[23])
    CNOT | (x[7], x[31]); CNOT | (x[22], x[31])
    CNOT | (x[6], x[30]); CNOT | (x[14], x[7])
    CNOT | (x[21], x[14]); CNOT | (x[5], x[6])
    CNOT | (x[21], x[22]); CNOT | (x[20], x[5])
    CNOT | (x[28], x[21]); CNOT | (x[21], x[28])
    CNOT | (x[13], x[21]); CNOT | (x[27], x[12])
    CNOT | (x[26], x[27]); CNOT | (x[20], x[28])
    CNOT | (x[4], x[20]); CNOT | (x[1], x[26])
    CNOT | (x[30], x[14]); CNOT | (x[12], x[5])
    CNOT | (x[19], x[13]); CNOT | (x[28], x[19])
    CNOT | (x[25], x[1]); CNOT | (x[24], x[0])
    CNOT | (x[0], x[1]); CNOT | (x[26], x[2])
    CNOT | (x[9], x[25]); CNOT | (x[7], x[15])
CNOT | (x[23], x[7]); CNOT | (x[14], x[6])
CNOT | (x[17], x[0]); CNOT | (x[31], x[23])
CNOT | (x[18], x[26]); CNOT | (x[8], x[22])
CNOT | (x[0], x[17]); CNOT | (x[11], x[27])

# Index setting of output qubits
x_out = []
x_out.append(x[0]); x_out.append(x[1])
x_out.append(x[26]); x_out.append(x[27])
x_out.append(x[12]); x_out.append(x[5])
x_out.append(x[22]); x_out.append(x[23])
x_out.append(x[8]); x_out.append(x[25])
x_out.append(x[2]); x_out.append(x[3])
x_out.append(x[28]); x_out.append(x[21])
x_out.append(x[20]); x_out.append(x[30])
x_out.append(x[29]); x_out.append(x[7])
x_out.append(x[24]); x_out.append(x[17])
x_out.append(x[10]); x_out.append(x[11])
x_out.append(x[4]); x_out.append(x[13])
x_out.append(x[14]); x_out.append(x[15])

return x_out
Code 1.3: Quantum circuit for AES MixColumn (shallow/low depth version).

```python
def MixColumn_low_depth(eng, t_ancillas, u_in):
    s = eng.allocate_qureg(32)  # qubits for output
    x = []
    for i in range(8):
        x.append(u_in[24 + i])
    for i in range(8):
        x.append(u_in[16 + i])
    for i in range(8):
        x.append(u_in[8 + i])
    for i in range(8):
        x.append(u_in[i])
    t = []
    for p in range(32):
        t.append(s[p])
    for p in range(71):
        t.append(t_ancillas[p])
    with Compute(eng):
        CNOT2(eng, x[5], x[13], t[32]); CNOT2(eng, x[21], x[29], t[43])
        CNOT2(eng, x[15], x[30], t[44]); CNOT2(eng, x[7], x[16], t[46])
        CNOT2(eng, x[23], x[24], t[47]); CNOT2(eng, x[1], x[18], t[56])
        CNOT2(eng, x[17], x[26], t[57]); CNOT2(eng, x[6], x[22], t[70])
        CNOT2(eng, t[70], t[43], t[44]); CNOT2(eng, x[14], x[31], t[71])
        CNOT2(eng, t[71], t[70], t[42]); CNOT2(eng, x[71], x[15], t[72])
        CNOT2(eng, t[72], t[71], t[44]); CNOT2(eng, x[1], x[18], t[73])
        CNOT2(eng, x[7], t[74], t[39])
        CNOT2(eng, x[7], t[73], t[38]); CNOT2(eng, x[6], x[23], t[74])
        CNOT2(eng, x[7], t[72], t[37]); CNOT2(eng, x[5], x[22], t[75])
    with Compute(eng):
        CNOT2(eng, x[12], x[28], t[75]); CNOT2(eng, x[3], x[7], t[76])
        CNOT2(eng, t[75], t[76], t[63]); CNOT2(eng, x[13], x[29], t[77])
        CNOT2(eng, t[75], t[77], t[36]); CNOT2(eng, x[30], t[77], t[38])
        CNOT2(eng, t[38], t[35], t[44])
    with Compute(eng):
        CNOT2(eng, x[14], x[22], t[78]); CNOT2(eng, t[78], x[30], t[34])
        CNOT2(eng, t[32], t[34], t[61]); CNOT2(eng, t[35], t[34], t[22])
        CNOT2(eng, t[37], t[34], t[23])
    with Compute(eng):
        CNOT2(eng, x[16], x[20], t[79]); CNOT2(eng, t[79], t[76], t[64])
        CNOT2(eng, x[12], x[20], t[80]); CNOT2(eng, x[5], t[80], t[41])
        CNOT2(eng, t[43], t[41], t[40]); CNOT2(eng, t[44], t[36], t[21])
    with Compute(eng):
        CNOT2(eng, x[14], x[21], t[81]); CNOT2(eng, x[5], t[81], t[40])
        CNOT2(eng, t[40], t[35], t[30])
    with Compute(eng):
        CNOT2(eng, x[18], x[23], t[82]); CNOT2(eng, x[11], x[27], t[83])
        CNOT2(eng, t[82], t[83], t[69]); CNOT2(eng, x[3], x[19], t[84])
        CNOT2(eng, x[8], t[69], t[68]); CNOT2(eng, x[16], x[23], t[85])
        CNOT2(eng, t[85], x[28], t[62]); CNOT2(eng, x[6], x[8], t[86])
        CNOT2(eng, x[31], t[86], t[45])
        CNOT2(eng, t[45], t[47], t[16]); CNOT2(eng, t[46], t[46], t[24])
```
```
with Compute(eng):
    CNOT2(eng, t[86], t[49]); CNOT2(eng, x[2], x[10], t[87])
    CNOT2(eng, t[87], x[25], t[55])
    CNOT2(eng, t[56], t[57], t[49]); CNOT2(eng, t[58], t[56], t[26])

with Compute(eng):
    CNOT2(eng, x[3], x[26], t[88]); CNOT2(eng, t[88], x[31], t[86])
    CNOT2(eng, t[69], t[66], t[19])

with Compute(eng):
    CNOT2(eng, x[12], x[27], t[89]); CNOT2(eng, t[89], x[31], t[61])
    CNOT2(eng, t[64], t[61], t[26])

with Compute(eng):
    CNOT2(eng, x[8], x[15], t[90]); CNOT2(eng, x[24], t[90], t[48])
    CNOT2(eng, t[48], t[46], t[0]); CNOT2(eng, t[49], t[48], t[8])

with Compute(eng):
    CNOT2(eng, x[9], x[25], t[91]); CNOT2(eng, t[91], t[53], t[1])
    CNOT2(eng, t[54], t[52], t[9])

with Compute(eng):
    CNOT2(eng, x[11], x[19], t[92]); CNOT2(eng, x[15], t[92], t[54])
    CNOT2(eng, t[58], t[67], t[10])

with Compute(eng):
    CNOT2(eng, x[10], x[27], t[93]); CNOT2(eng, x[21], t[93], t[67])
    CNOT2(eng, t[36], t[33], t[5]); CNOT2(eng, t[32], t[33], t[29])

with Compute(eng):
    CNOT2(eng, x[10], x[26], t[94]); CNOT2(eng, x[9], t[94], t[59])
    CNOT2(eng, t[59], t[56], t[2])

with Compute(eng):
    CNOT2(eng, x[2], x[18], t[96]); CNOT2(eng, x[9], t[96], t[58])
    CNOT2(eng, t[58], t[57], t[10])

with Compute(eng):
    CNOT2(eng, x[8], x[15], t[97]); CNOT2(eng, t[97], t[67], t[11])

with Compute(eng):
    CNOT2(eng, x[11], x[19], t[98]); CNOT2(eng, x[2], x[18], t[100])
    CNOT2(eng, t[99], t[100], t[65])

CNOT2(eng, t[65], t[67], t[3]); CNOT2(eng, t[65], t[66], t[27])

with Compute(eng):
    CNOT2(eng, x[10], x[101]); CNOT2(eng, x[1], x[24], t[102])
    CNOT2(eng, t[101], t[102], t[50])

CNOT2(eng, t[50], t[52], t[17]); CNOT2(eng, t[50], t[51], t[26])

#Computation end
Index setting of output qubits

```python
#Index setting of output qubits

    t_out = []
    Uncompute(eng); Uncompute(eng); Uncompute(eng); Uncompute(eng)  
    Uncompute(eng); Uncompute(eng); Uncompute(eng); Uncompute(eng)  
    Uncompute(eng); Uncompute(eng); Uncompute(eng); Uncompute(eng)  
    Uncompute(eng); Uncompute(eng); Uncompute(eng); Uncompute(eng)  
    Uncompute(eng); Uncompute(eng); Uncompute(eng); Uncompute(eng)  
    Uncompute(eng); Uncompute(eng); Uncompute(eng); Uncompute(eng)  

    for i in range(8):
        t_out.append(t[24 + i])
    for i in range(8):
        t_out.append(t[16 + i])
    for i in range(8):
        t_out.append(t[8 + i])
    for i in range(8):
        t_out.append(t[0 + i])

    return t_out
```
<table>
<thead>
<tr>
<th>Cipher</th>
<th>Round</th>
<th>#CNOT</th>
<th>#NOT</th>
<th>#Toffoli</th>
<th>Toffoli depth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>†</td>
<td>✠</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>AES-128</td>
<td>1</td>
<td>7,920</td>
<td>4,544</td>
<td>5,568</td>
<td>79 (✠), 6 (✠)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>7,792</td>
<td>7,920</td>
<td>8,944</td>
<td>79 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7,792</td>
<td>7,920</td>
<td>8,944</td>
<td>81 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7,792</td>
<td>7,920</td>
<td>8,944</td>
<td>81 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>7,792</td>
<td>7,920</td>
<td>8,944</td>
<td>81 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7,792</td>
<td>7,920</td>
<td>8,944</td>
<td>79 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>7,792</td>
<td>7,920</td>
<td>8,944</td>
<td>79 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>7,792</td>
<td>7,920</td>
<td>8,944</td>
<td>81 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>7,792</td>
<td>7,920</td>
<td>8,944</td>
<td>80 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>3,984</td>
<td>4,048</td>
<td>4,048</td>
<td>80 680, 680</td>
</tr>
<tr>
<td>AES-192</td>
<td>1</td>
<td>7,984</td>
<td>4,576</td>
<td>5,600</td>
<td>79 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>7,856</td>
<td>7,952</td>
<td>8,976</td>
<td>79 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>6,256</td>
<td>7,008</td>
<td>8,032</td>
<td>64 1,088, 1,224</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7,856</td>
<td>7,200</td>
<td>8,224</td>
<td>81 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>7,856</td>
<td>7,952</td>
<td>8,976</td>
<td>81 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>6,256</td>
<td>7,008</td>
<td>8,032</td>
<td>64 1,088, 1,224</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>7,856</td>
<td>7,200</td>
<td>8,224</td>
<td>81 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>7,856</td>
<td>7,952</td>
<td>8,976</td>
<td>79 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>6,256</td>
<td>7,008</td>
<td>8,032</td>
<td>64 1,088, 1,224</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>7,856</td>
<td>7,200</td>
<td>8,224</td>
<td>79 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>7,168</td>
<td>7,952</td>
<td>8,976</td>
<td>81 1,224, 1,360</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>3,136</td>
<td>3,136</td>
<td>3,136</td>
<td>64 544, 544</td>
</tr>
<tr>
<td>AES-256</td>
<td>1</td>
<td>6,384</td>
<td>3,632</td>
<td>4,656</td>
<td>64 1,088, 544</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>7,792</td>
<td>7,104</td>
<td>8,128</td>
<td>79 1,360, 1,224</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7,792</td>
<td>7,792</td>
<td>8,816</td>
<td>80 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7,792</td>
<td>7,792</td>
<td>8,816</td>
<td>79 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>7,792</td>
<td>7,792</td>
<td>8,816</td>
<td>80 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7,792</td>
<td>7,792</td>
<td>8,816</td>
<td>81 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>7,792</td>
<td>7,792</td>
<td>8,816</td>
<td>80 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>7,792</td>
<td>7,792</td>
<td>8,816</td>
<td>81 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>7,792</td>
<td>7,792</td>
<td>8,816</td>
<td>80 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>7,792</td>
<td>7,792</td>
<td>8,816</td>
<td>81 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>7,792</td>
<td>7,792</td>
<td>8,816</td>
<td>80 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>7,792</td>
<td>7,792</td>
<td>8,816</td>
<td>79 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>7,792</td>
<td>7,792</td>
<td>8,816</td>
<td>80 1,360, 1,360</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>3,984</td>
<td>3,984</td>
<td>3,984</td>
<td>79 680, 680</td>
</tr>
</tbody>
</table>

- †: Regular version (using MixColumn from [56]).
- ✠: Shallow version (using MixColumn from [56]).
- ○: Shallow/low depth version (using MixColumn from [46]).
- ∘: Including initial key XOR.