Quantum Analysis of AES

Kyungbae Jang[1], Anubhab Baksi[2], Gyeongju Song[1], Hyunji Kim[1], Hwajeong Seo[1], and Anupam Chattopadhyay[2]

[1] Division of IT Convergence Engineering, Hansung University, Seoul, South Korea

[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. Moreover, our qubit count - Toffoli depth product is improved from theirs by more than 75 percent. Moreover, 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 presents the currently best-known results, thereby improving 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[3] being organized by the US government's National Institute of Standards and Technology (NIST)[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,

We thank Da Lin (Hubei University, Wuhan, PR China) for the kind support.

[3] https://csrc.nist.gov/projects/post-quantum-cryptography.

[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 roadmap[5]. 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 $\times$ 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 $\times$ 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 isssues (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

[5] https://research.ibm.com/blog/ibm-quantum-roadmap.

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 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 1: Performance comparison of AES quantum implementations.

Cipher	Source	$\#TD^{\bullet\bullet}$	$\#M^{\circ\!\!\circ}$	$\#TD \times \#M^{\bullet\bullet\circ\!\!\circ}$	Full depth
AES-128	GLRS [30]	12,672 (99.76)	984 (−84.55)	12,469,248 (97.96)	110,799 (99.12)
	LPS [43]	1,880 (98.41)	864 (−86.43)	1,624,320 (84.32)	28,927 (96.62)
	ZWSLW [59]	2,016 (98.51)	512 (−91.96)	1,032,192 (75.32)	·
	JNRV [38] (fixed)	2,394 (98.33)	1,656 (−74.00)	3,964,464 (93.58)	33,320 (97.07)
	This work	$40^{⌓✠}$	$6{,}368^{⌓}$	$254{,}720^{⌓}$	$978^{⌓}$
AES-192	GLRS [30]	11,088 (99.68)	1,112 (−83.37)	12,329,856 (97.34)	96,956 (98.79)
	LPS [43]	1,640 (97.81)	896 (−86.60)	1,469,440 (78.15)	25,556 (95.41)
	ZWSLW [59]	2,022 (98.22)	640 (−90.43)	1,294,080 (75.19)	·
	JNRV [38] (fixed)	2,682 (98.21)	1,976 (−70.46)	5,299,632 (93.94)	37,328 (96.86)
	This work	$48^{⌓✠}$	$6{,}688^{⌓}$	$321{,}024^{⌓}$	$1{,}174^{⌓}$
AES-256	GLRS [30]	14,976 (99.72)	1,336 (−80.85)	20,007,936 (98.05)	130,929 (98.95)
	LPS [43]	2,160 (98.06)	1,232 (−82.34)	2,661,120 (85.32)	33,525 (95.89)
	ZWSLW [59]	2,292 (98.17)	768 (−88.99)	1,760,256 (77.81)	·
	JNRV [38] (fixed)	3,306 (98.31)	2,296 (−67.09)	7,590,576 (94.85)	46,012 (97.01)
	This work	$56^{⌓✠}$	$6{,}976^{⌓}$	$390{,}656^{⌓}$	$1{,}377^{⌓}$

Parenthesized numbers show % improvement reported in this work.
$\bullet\bullet$: $\#TD$ is Toffoli depth.
$\circ\!\!\circ$: $\#M$ 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[6],

[6]Homepage: https://projectq.ch/.

which is a Python-based open-source framework for quantum computing. All our relevant source codes can be accessed as an open-source project[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[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 $\lfloor \frac{\pi}{4}\sqrt{2^k} \rfloor$ 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:

$$|\psi\rangle = H^{\otimes k}\,|0\rangle^{\otimes k} = \left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right) = \frac{1}{2^{k/2}} \sum_{x=0}^{2^k-1} |x\rangle \tag{1}$$

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).

$$f(x) = \begin{cases} 1 \text{ if } Enc(k) = c \\ 0 \text{ if } Enc(k) \neq c \end{cases} \tag{2}$$

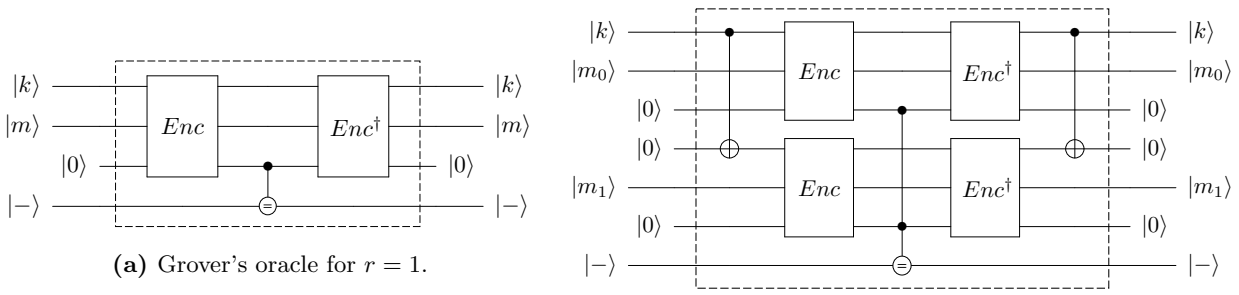
[7] https://github.com/starj1023/AES_QC.
[8] https://github.com/AES-quantum-circuit/AES-quantum-circuit.

$$U_f(|\psi\rangle\,|-\rangle) = \frac{1}{2^{k/2}} \sum_{x=0}^{2^k-1} (-1)^{f(x)}\,|x\rangle\,|-\rangle \tag{3}$$

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 $\lfloor \frac{\pi}{4}\sqrt{2^k} \rfloor$ [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† is defined as the reverse operation of encryption, which reverts to the state before encryption.



(a) Grover's oracle for $r = 1$.

(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 Sbox 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 J. Boyar and Peralta are frequently used. In [43], Langenberg et al. adopted the S-box implementation of [14] and converted it to suit their purpose of reducing qubits. In [58], the S-box implementation of [14] was adopted and improved. 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 2: Comparison of quantum resources required for S-box implementations.

Method	#CNOT	#1qCliff	#T	Toffoli depth	#qubits	Full depth
S-box [14]	358	68	224	8	123	104
S-box [15]	392	72	238	6	136	85
S-box [43]	628	98	367	40	32	514
S-box [59]	437	72	245	55	22	339
S-box [33]$^{\text{☼}}$	418	72	238	4	136	72
S-box [33]$^{\circledast}$	824	160	546	3	198	69

☼⊛: 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 reduceed 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.

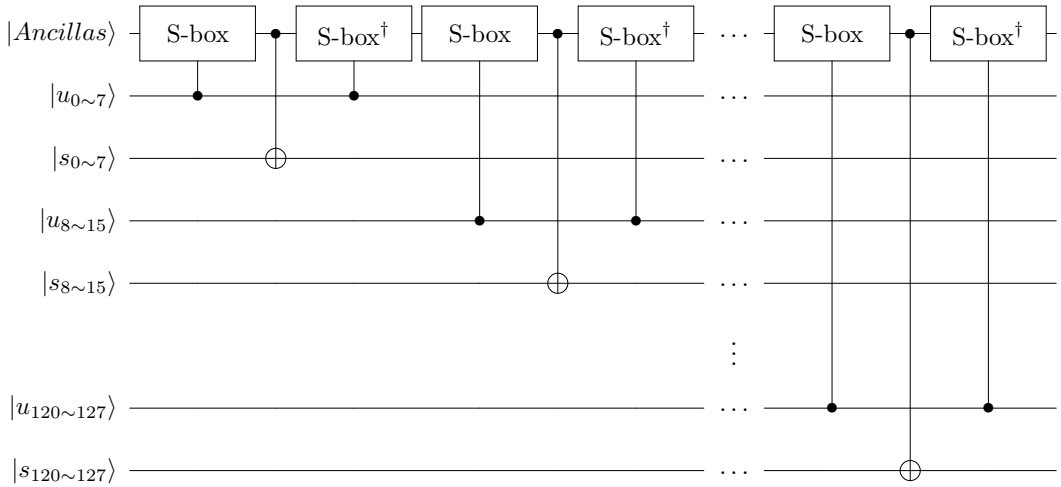
Considering the trade-off between the circuit depth and the number of qubits required for an S-box implementation, we treat two cases. The first case is when the ancilla qubits have to be allocated per SubBytes, which is indeed sensitive to the number of qubits. The second case is when the initially 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.

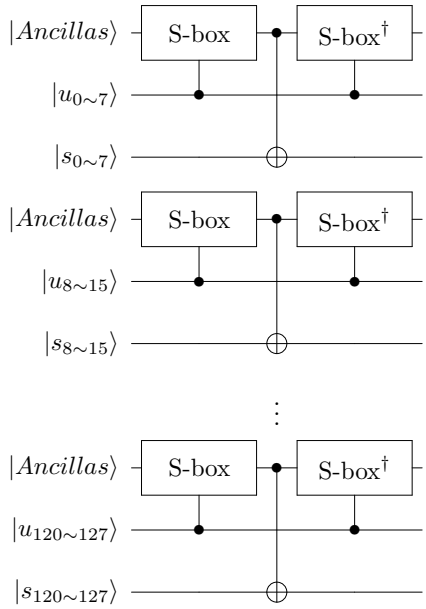
[10] https://github.com/dansarie/sboxgates.

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.



(a) Using 1 set of ancillas.



(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 $|k_{120\sim127}\rangle$ 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†).

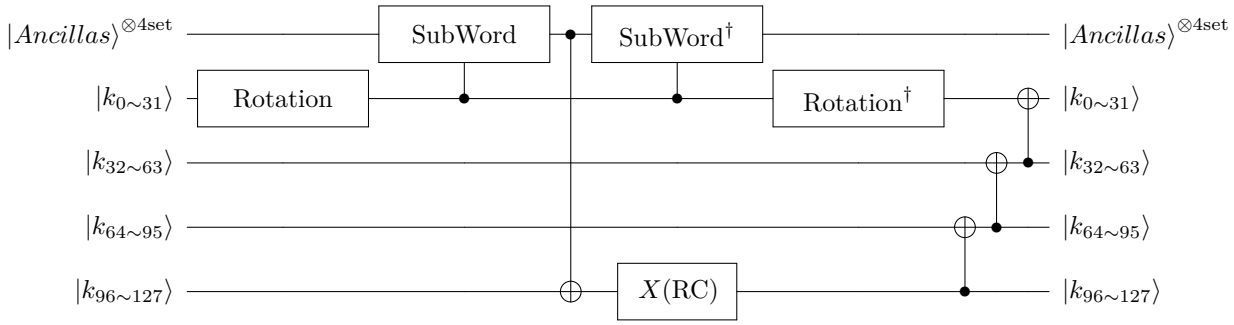


Fig. 3: AES-128 key schedule in quantum.

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.

Method	#CNOT	#qubits	Full depth
MixColumn [29, 59]	277	32	39
MixColumn [41]	194	129	15
MixColumn [1]	275	32	200
MixColumn [47]†	184	124	16
MixColumn [38]✤	277	32	111
MixColumn [53]	188	126	17
MixColumn [56]☆♢	92	32	30
MixColumn [45]⅄	182	123	16
MixColumn [46]ϒ	206	135	13
MixColumn [44]ϒ✠	210	137	11

†✤: Reused in this work to fix [38].
☆♢: Used in this work (regular, shallow versions), [33].
⅄: Least XOR count in classical circuit.
✠: Used in this work (shallow/low depth version).
ϒ: Least depth in classical circuit.

3.7 Architecture of Quantum Circuits

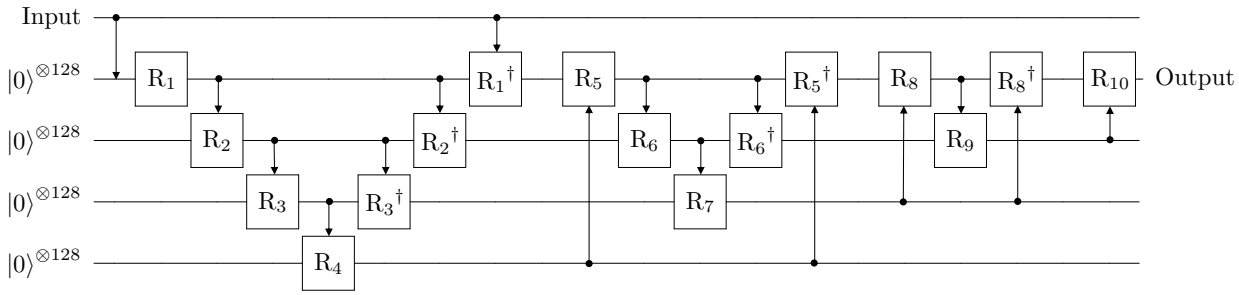
There are several architectures for designing quantum circuits of AES. The architectures differ in how they store the 128-qubit output generated from SubBytes in each round. In [1, 30, 43], the basic zig-zag architecture (Figure 4(a)) was adopted that uses 4 lines to save qubits by performing reverses 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.

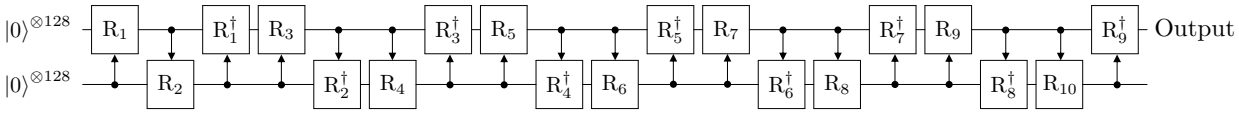
In Figure 5(a), SubBytes to generate 128-qubit output and SubBytes† to clean ancilla qubits operate serially 19 times in total (SubBytes is 10, SubBytes† is 9). Depending on the number of rounds, AES-192 operates 23 times and AES-256 operates 27 times. In our parallel design, the key schedule operates 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 20×120 ancilla qubits. In the shallow version, the first SubBytes in Figure 5(b) uses the first 20×120 ancilla qubits. The second SubBytes uses the second 20×120 ancilla qubits, and at the same time SubBytes† cleans the first 20×120 ancilla qubits. That is, SubBytes† operates simultaneously with the SubBytes of the next round. Conceptually, this can be thought as all SubBytes† in Figure 5(a) are pushed one space to the right. This is possible because SubBytes and SubBytes† 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† 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.



(a) Basic implementation.



(b) Improved implementation.

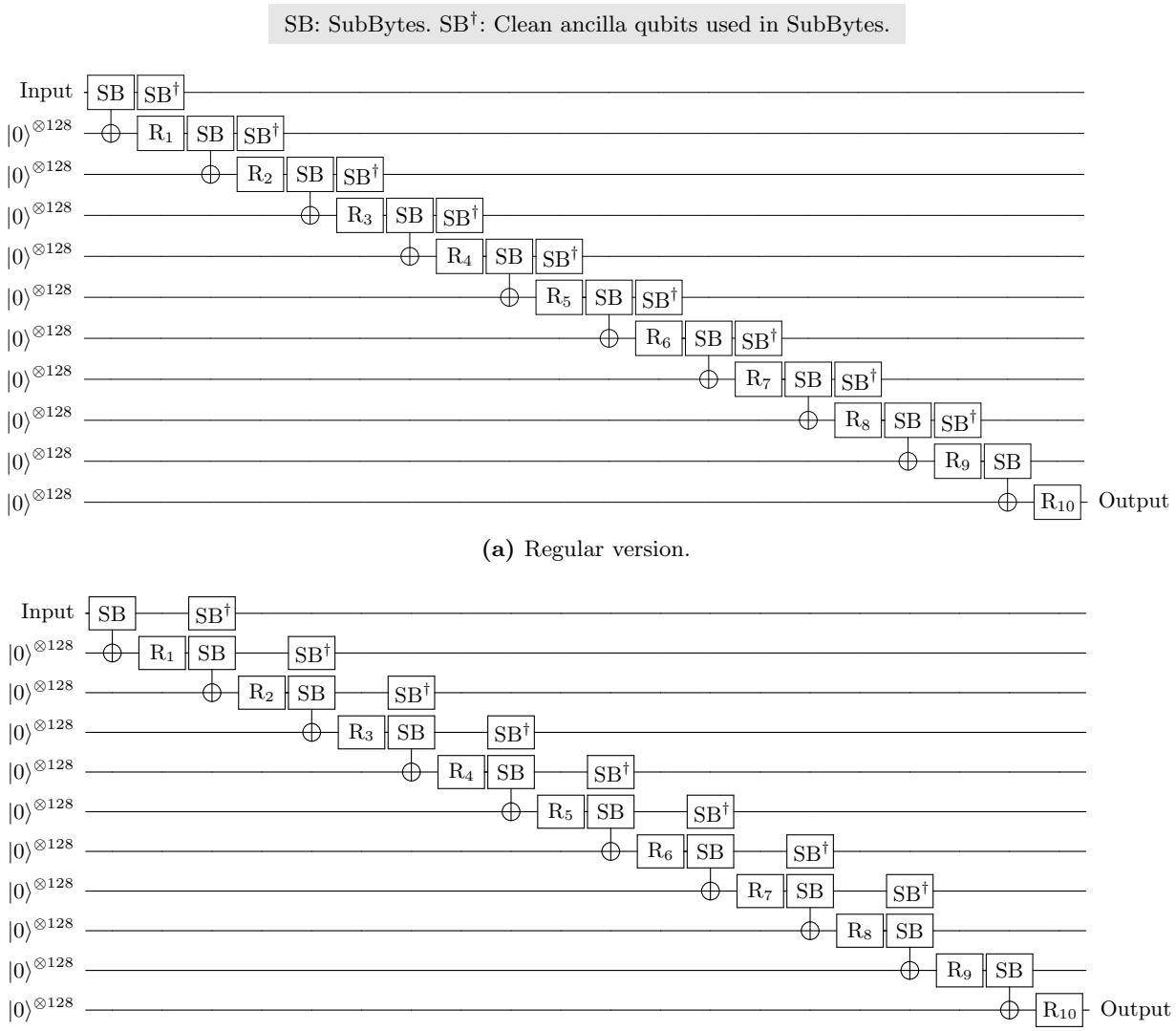
Fig. 4: Zig-zag architecture for AES quantum circuit.

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



(a) Regular version.

(b) Shallow version (and shallow/low depth version).

Fig. 5: Pipeline architecture of AES.

paper [38]. This was discovered when we tried to cross-check their publicly available source codes[11]. 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# (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

[11] https://github.com/microsoft/grover-blocks.

Table 4: Comparison of quantum resources required for variants of AES.

Cipher	Source	#CNOT	#NOT	#Toffoli	Toffoli depth	#qubits (M)	$T \times M$	Full depth
AES-128	GLRS [30]	166,548	1,456	151,552	12,672	984	12,469,248	110,799
	ASAM [1]	192,832	1,370	150,528	·	976	·	·
	LPS [43]	107,960	1,570	16,940	1,880	864	1,624,320	28,927
	ZWSLW [30]	128,517	4,528	19,788	2,016	512	1,032,192	·
	Regular☼	84,120	800	12,920	76	3,936	299,136	1,364
	Shallow☼	81,312	800	12,240	40	6,368	254,720	978
	Shallow/low☼	90,816	800	12,240	40	7,520	300,800	799
	Regular⊛	138,080	800	29,640	57	5,176	295,032	1,307
	Shallow⊛	132,432	800	28,080	30	8,848	265,440	948
	Shallow/low⊛	141,936	800	28,080	30	10,000	300,000	769
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	321,024	1,174
	Shallow/low☼	104,472	896	14,008	48	8,096	388,608	955
	Regular⊛	157,456	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	1,336	20,007,936	130,929
	LPS [43]	151,011	1,992	23,760	2,160	1,232	2,661,120	33,525
	ZWSLW [30]	177,645	6,103	26,774	2,292	768	1,760,256	·
	Regular☼	117,704	1,103	18,088	108	4,576	494,208	1,907
	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	5,816	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[12].

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.

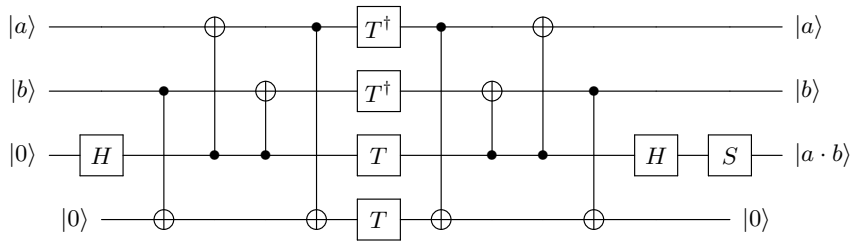
[12] https://github.com/microsoft/qsharp-runtime/pull/404.

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.

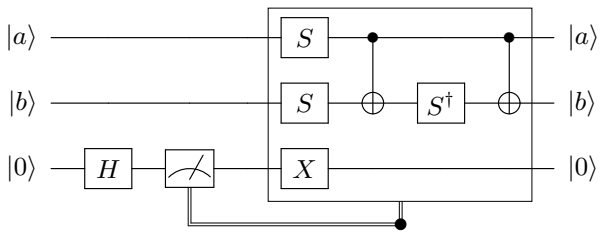
According to the reported number of required qubits, only one ancilla set is used in their SubBytes implementation. In other words, 16 S-boxes share one ancilla set. Thus, the arrangement of qubits in their SubBytes quantum circuit is the serial structure of Figure 2(a). Since 16 S-boxes generate each 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.



(a) AND gate.



(b) AND† gate.

Fig. 6: Quantum AND and AND† gates.

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

Method	S-box	MixColumn	
		In-place [38]	Maximov [47]
#CNOT	654	1,108	1,248
#1qCliff	184	0	0
#T	136	0	0
#Measure	34	0	0
#T-depth	6	0	0
#qubits	137	128	318
Full depth	101	111	22

(b) Oracles

Method	In-place MixColumn [38]			Maximov's MixColumn [47]		
	AES-128	AES-192	AES-256	AES-128	AES-192	AES-256
#CNOT	292,313	329,697	404,139	294,863	332,665	407,667
#1qCliff	84,428	94316	116,286	84,488	94,092	116,062
#T	54,908	61,436	75,580	54,908	61,436	75,580
#Measure	13,727	15,359	18,895	13,727	15,359	18,895
#T-depth	121	120	126	121	120	126
#qubits	1,665	1,985	2,305	2,817	3,393	3,969
Full depth	2,816	2,978	3,353	2,086	1,879	1,951

(c) AES-128 Modules

Method	#CNOT	#1qCliff	#T	Toffoli depth	#qubits	Full depth
SubBytes	12,000	1,220	7,328	192	376	2,672
Key schedule	3,096	355	1,832	48	248	669
MixColumns (Maximov [47])	1,248	0	0	0	318	88
One round$^{\Upsilon}$	16,472	1,507	9,160	240	632	3,417

$^{\Upsilon}$: One typical round (that includes MixColumn).

(d) Summary

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

$\clubsuit$: Using in-place MixColumn [38].
$\dagger$: 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(b) shows the quantum resources reported in the Eurocrypt'20 paper. Quantum resources for S-box and MixColumns include cleaning up ancilla qubits. 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`[13] 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].

[13] https://github.com/microsoft/qsharp-runtime/issues/192.

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

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

☆: Regular version (using MixColumn from [56]).
⌔: Shallow version (using MixColumn from [56]).
✠: Shallow/low depth version (using MixColumn from [44]).
☼: S-box with Toffoli depth 4.
⊛: S-box with Toffoli depth 3.

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

Cipher	r	#qubits	Total gates	Total depth	Search complexity	NIST security
AES-128☆	1	3,937	$1.597 \cdot 2^{82}$	$1.046 \cdot 2^{75}$	$1.671 \cdot 2^{157}$	Not achieved ($< 2^{170}$)
AES-128⌔		6,369	$1.527 \cdot 2^{82}$	$1.501 \cdot 2^{74}$	$1.146 \cdot 2^{157}$	
AES-128✠		7,521	$1.599 \cdot 2^{82}$	$1.226 \cdot 2^{74}$	$1.96 \cdot 2^{156}$	
AES-192☆	2	8,321	$1.536 \cdot 2^{115}$	$1.248 \cdot 2^{107}$	$1.917 \cdot 2^{222}$	Level-1 ($< 2^{233}$)
AES-192⌔		13,185	$1.464 \cdot 2^{115}$	$1.801 \cdot 2^{106}$	$1.318 \cdot 2^{222}$	
AES-192✠		16,001	$1.551 \cdot 2^{115}$	$1.465 \cdot 2^{106}$	$1.136 \cdot 2^{222}$	
AES-256☆	2	8,897	$1.797 \cdot 2^{147}$	$1.463 \cdot 2^{139}$	$1.314 \cdot 2^{287}$	Level-3 ($< 2^{298}$)
AES-256⌔		13,697	$1.720 \cdot 2^{147}$	$1.056 \cdot 2^{139}$	$1.816 \cdot 2^{286}$	
AES-256✠		17,025	$1.824 \cdot 2^{147}$	$1.715 \cdot 2^{138}$	$1.564 \cdot 2^{286}$	

☆: Regular version (using MixColumn from [56]).
⌔: Shallow version (using MixColumn from [56]).
✠: Shallow/low depth version (using MixColumn from [44]).

In the oracle, the target cipher's quantum circuit encrypts a known plaintext with the key in the superposition state. 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 sholud be estimated only once. Finally, the cost of quantum exhaustive key search for the target block cipher is roughly the cost of oracle $\times \lfloor \frac{\pi}{4}\sqrt{2^k} \rfloor$ (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 8×8 S-boxes. The tools described in [18, 19, 20] can possibly be considered as starting points.

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); however

Table 8: Comparison of NIST security levels for variants of AES.

NIST security	LPS [43]	This work
Not achieved ($< 2^{170}$)	AES-128	
	2^{162}	2^{157}
Level-1 ($\geq 2^{170}$)	AES-192	
	2^{227}	2^{222}
Level-3 ($\geq 2^{233}$)	AES-256	
	2^{291}	2^{286}

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

1. Almazrooie, M., Samsudin, A., Abdullah, R., Mutter, K.N.: Quantum reversible circuit of AES-128. Quantum Information Processing **17**(5) (may 2018) 1–30 2, 6, 10, 11, 14, 18
2. Amy, M., Di Matteo, O., Gheorghiu, V., Mosca, M., Parent, A., Schanck, J.: Estimating the cost of generic quantum pre-image attacks on SHA-2 and SHA-3. In Avanzi, R., Heys, H., eds.: Selected Areas in Cryptography – SAC 2016, Cham, Springer International Publishing (2017) 317–337 5
3. Amy, M., Maslov, D., Mosca, M., Roetteler, M., Roetteler, M.: A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **32**(6) (Jun 2013) 818–830 6, 15, 18
4. Anand, R., Maitra, A., Maitra, S., Mukherjee, C.S., Mukhopadhyay, S.: Quantum resource estimation for FSR based symmetric ciphers and related Grover's attacks. In Adhikari, A., Küsters, R., Preneel, B., eds.: Progress in Cryptology - INDOCRYPT 2021 - 22nd International Conference on Cryptology in India, Jaipur, India, December 12-15, 2021, Proceedings. Volume 13143 of Lecture Notes in Computer Science., Springer (2021) 179–198 5
5. Anand, R., Maitra, A., Mukhopadhyay, S.: Evaluation of quantum cryptanalysis on SPECK. In Bhargavan, K., Oswald, E., Prabhakaran, M., eds.: Progress in Cryptology – INDOCRYPT 2020, Cham, Springer International Publishing (2020) 395–413 5, 18
6. Anand, R., Maitra, A., Mukhopadhyay, S.: Grover on SIMON. Quantum Information Processing **19**(9) (Sep 2020) http://dx.doi.org/10.1007/s11128-020-02844-w. 5
7. Baksi, A., Jang, K., Song, G., Seo, H., Xiang, Z.: Quantum implementation and resource estimates for rectangle and knot. Quantum Information Processing **20**(12) (dec 2021) 5, 20
8. Banik, S., Funabiki, Y., Isobe, T.: Further results on efficient implementations of block cipher linear layers. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **104-A**(1) (2021) 213–225 10, 20
9. Bathe, B.N., Anand, R., Dutta, S.: Evaluation of Grover's algorithm toward quantum cryptanalysis on ChaCha. Quantum Inf. Process. **20**(12) (2021) 394 5
10. Bhattacharjee, D., Chattopadhyay, A.: Depth-optimal quantum circuit placement for arbitrary topologies. arXiv preprint arXiv:1703.08540 (2017) 2
11. Bijwe, S., Chauhan, A.K., Sanadhya, S.K.: Quantum search for lightweight block ciphers: Gift, skinny, saturnin. Cryptology ePrint Archive (2020) 20
12. Bonnetain, X., Leurent, G., Naya-Plasencia, M., Schrottenloher, A.: Quantum linearization attacks. In Tibouchi, M., Wang, H., eds.: Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part I. Volume 13090 of Lecture Notes in Computer Science., Springer (2021) 422–452 5
13. Bonnetain, X., Naya-Plasencia, M., Schrottenloher, A.: Quantum security analysis of AES. IACR Transactions on Symmetric Cryptology **2019**(2) (Jun. 2019) 55–93 5
14. Boyar, J., Peralta, R.: A new combinational logic minimization technique with applications to cryptology. In Festa, P., ed.: Experimental Algorithms, Berlin, Heidelberg, Springer Berlin Heidelberg (2010) 178–189 6, 7, 15, 18
15. Boyar, J., Peralta, R.: A depth-16 circuit for the AES S-box. Cryptology ePrint Archive, Report 2011/332 (2011) https://eprint.iacr.org/2011/332. 6, 7, 16
16. Boyer, M., Brassard, G., Høyer, P., Tapp, A.: Tight bounds on quantum searching. Fortschritte der Physik **46**(4-5) (Jun 1998) 493–505 5

17. Daemen, J., Rijmen, V.: The Design of Rijndael: AES - The Advanced Encryption Standard. Information Security and Cryptography. Springer (2002) 2, 23
18. Dansarie, M.: Cryptanalysis of the SoDark family of cipher algorithms. PhD thesis, Naval Postgraduate School, Dudley Knox Library (2017) https://calhoun.nps.edu/handle/10945/56118. 7, 20
19. Dansarie, M.: sboxgates: A program for finding low gate count implementations of S-boxes. Journal of Open Source Software **6**(62) (2021) 2946 7, 20
20. Dasu, V.A., Baksi, A., Sarkar, S., Chattopadhyay, A.: LIGHTER-R: optimized reversible circuit implementation for sboxes. In: 32nd IEEE International System-on-Chip Conference, SOCC 2019, Singapore, September 3-6, 2019. (2019) 260–265 20
21. de Wolf, R.: Quantum Computing: Lecture Notes. (2019) https://arxiv.org/pdf/1907.09415v1.pdf. 4
22. Dong, X., Dong, B., Wang, X.: Quantum attacks on some feistel block ciphers. Des. Codes Cryptogr. **88**(6) (2020) 1179–1203 1
23. Dong, X., Sun, S., Shi, D., Gao, F., Wang, X., Hu, L.: Quantum collision attacks on AES-like hashing with low quantum random access memories. In Moriai, S., Wang, H., eds.: Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part II. Volume 12492 of Lecture Notes in Computer Science., Springer (2020) 727–757 5
24. Dong, X., Zhang, Z., Sun, S., Wei, C., Wang, X., Hu, L.: Automatic classical and quantum rebound attacks on AES-like hashing by exploiting related-key differentials. In Tibouchi, M., Wang, H., eds.: Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part I. Volume 13090 of Lecture Notes in Computer Science., Springer (2021) 241–271 5
25. Ducas, L., Lepoint, E.K.T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-Dilithium – algorithm specifications and supporting documentation (version 3.1). Technical report, 2021 (2021) https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf. 1
26. Ekdahl, P., Johansson, T., Maximov, A., Yang, J.: A new snow stream cipher called snow-v. IACR Transactions on Symmetric Cryptology **2019**(3) (Sep. 2019) 1–42 21
27. Feng, J., Chen, H., Gao, S., Fan, L., Feng, D.: Improved fault analysis on the block cipher speck by injecting faults in the same round. In Hong, S., Park, J.H., eds.: Information Security and Cryptology – ICISC 2016: Seoul, South Korea, Revised Selected Papers, Cham, Springer International Publishing (2017) 317–332 20
28. Frixons, P., Naya-Plasencia, M., Schrottenloher, A.: Quantum boomerang attacks and some applications. IACR Cryptol. ePrint Arch. (2022) 60 5
29. Grassi, L., Rechberger, C., Rønjom, S.: Subspace trail cryptanalysis and its applications to AES. Volume 2016. (2016) 192–225 10, 11
30. Grassl, M., Langenberg, B., Roetteler, M., Steinwandt, R.: Applying Grover's algorithm to AES: Quantum resource estimates. In Takagi, T., ed.: Post-Quantum Cryptography, Cham, Springer International Publishing (2016) 29–43 2, 3, 6, 9, 11, 14, 18, 19, 20
31. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing. (1996) 212–219 2, 4
32. Grumbling, E., Horowitz, M.: Quantum Computing: Progress and Prospects. The National Academies Press, Washington DC (2019) https://www.nap.edu/catalog/25196/quantum-computing-progress-and-prospects. 1
33. Huang, Z., Sun, S.: Synthesizing quantum circuits of aes with lower t-depth and less qubits. Cryptology ePrint Archive, Report 2022/620 (2022) https://eprint.iacr.org/2022/620. 4, 7, 10, 11
34. Jang, K., Choi, S., Kwon, H., Kim, H., Park, J., Seo, H.: Grover on Korean block ciphers. Applied Sciences **10**(18) (2020) 5
35. Jang, K., Baksi, A., Breier, J., Seo, H., Chattopadhyay, A.: Quantum implementation and analysis of default. Cryptology ePrint Archive, Paper 2022/647 (2022) https://eprint.iacr.org/2022/647. 5
36. Jang, K., Song, G., Kim, H., Kwon, H., Kim, H., Seo, H.: Efficient implementation of PRESENT and GIFT on quantum computers. Applied Sciences **11**(11) (2021) 5, 20
37. Jang, K., Song, G., Kim, H., Kwon, H., Kim, H., Seo, H.: Parallel quantum addition for Korean block cipher. IACR Cryptol. ePrint Arch. (2021) 1507 20
38. Jaques, S., Naehrig, M., Roetteler, M., Virdia, F.: Implementing grover oracles for quantum key search on AES and lowmc. In Canteaut, A., Ishai, Y., eds.: Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part II. Volume 12106 of Lecture Notes in Computer Science., Springer (2020) 280–310 2, 3, 4, 5, 7, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
39. Kaplan, M., Leurent, G., Leverrier, A., Naya-Plasencia, M.: Breaking symmetric cryptosystems using quantum period finding. In: CRYPTO, Springer (2016) 207–237 1

40. Kaplan, M., Leurent, G., Leverrier, A., Naya-Plasencia, M.: Quantum differential and linear cryptanalysis. IACR Transactions on Symmetric Cryptology **2016**(1) (Dec. 2016) 71–94 5
41. Kranz, T., Leander, G., Stoffelen, K., Wiemer, F.: Shorter linear straight-line programs for mds matrices. IACR Transactions on Symmetric Cryptology **2017**(4) (Dec. 2017) 188–211 11
42. Langenberg, B., Pham, H., Steinwandt, R.: Reducing the cost of implementing AES as a quantum circuit. Cryptology ePrint Archive, Report 2019/854 (2019) https://eprint.iacr.org/2019/854. 5
43. Langenberg, B., Pham, H., Steinwandt, R.: Reducing the cost of implementing the advanced encryption standard as a quantum circuit. IEEE Transactions on Quantum Engineering **1** (01 2020) 1–12 2, 3, 5, 6, 7, 9, 11, 14, 18, 19, 20, 21
44. Li, S., Sun, S., Li, C., Wei, Z., Hu, L.: Constructing low-latency involutory mds matrices with lightweight circuits. IACR Transactions on Symmetric Cryptology (2019) 84–117 3, 10, 11, 12, 19
45. Lin, D., Xiang, Z., Zeng, X., Zhang, S.: A framework to optimize implementations of matrices. In Paterson, K.G., ed.: Topics in Cryptology - CT-RSA 2021 - Cryptographers' Track at the RSA Conference 2021, Virtual Event, May 17-20, 2021, Proceedings. Volume 12704 of Lecture Notes in Computer Science., Springer (2021) 609–632 4, 10, 11
46. Liu, Q., Wang, W., Fan, Y., Wu, L., Sun, L., Wang, M.: Towards low-latency implementation of linear layers. IACR Transactions on Symmetric Cryptology **2022**(1) (Mar. 2022) 158–182 4, 10, 11, 26, 34
47. Maximov, A.: AES MixColumn with 92 XOR gates. Cryptology ePrint Archive, Report 2019/833 (2019) https://eprint.iacr.org/2019/833. 4, 10, 11, 17
48. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information (10th Anniversary Edition). Cambridge University Press (2010) 4
49. NIST.: Submission requirements and evaluation criteria for the post-quantum cryptography standardization process (2016) https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf. 2, 20
50. Perriello, S.: Design and development of a quantum circuit to solve the information set decoding problem. (2019) 5
51. Schwabe, P., Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Seiler, G., Stehle, D.: CRYSTALS-KYBER. Submission to PQC third round (2021) https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf. 1
52. Song, G., Jang, K., Kim, H., Lee, W., Hu, Z., Seo, H.: Grover on SM3. IACR Cryptol. ePrint Arch. (2021) https://eprint.iacr.org/2021/668. 5
53. Tan, Q.Q., Peyrin, T.: Improved heuristics for short linear programs. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2020**(1) (2020) 203–230 11
54. Wang, Z.G., Wei, S.J., Long, G.L.: A quantum circuit design of aes requiring fewer quantum qubits and gate operations. Frontiers of Physics **17**(4) (2022) 1–7 2
55. Wang, Z., Wei, S., Long, G.: A quantum circuit design of AES (2021) https://arxiv.org/abs/2109.12354. 2, 6
56. Xiang, Z., Zeng, X., Lin, D., Bao, Z., Zhang, S.: Optimizing implementations of linear layers. IACR Trans. Symmetric Cryptol. **2020**(2) (2020) 120–145 3, 4, 10, 11, 19, 26, 34
57. Zou, J., Li, L., Wei, Z., Luo, Y., Liu, Q., Wu, W.: New quantum circuit implementations of sm4 and sm3. Quantum Information Processing **21**(5) (2022) 1–38 5
58. Zou, J., Liu, Y., Dong, C., Wu, W., Dong, L.: Observations on the quantum circuit of the SBox of AES. Cryptology ePrint Archive, Report 2019/1245 (2019) https://eprint.iacr.org/2019/1245. 6
59. Zou, J., Wei, Z., Sun, S., Liu, X., Wu, W.: Quantum circuit implementations of AES with fewer qubits. In Moriai, S., Wang, H., eds.: Advances in Cryptology – ASIACRYPT 2020, Cham, Springer International Publishing (2020) 697–726 2, 3, 4, 5, 6, 7, 9, 10, 11, 13, 18, 20, 26

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 4×4 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 8×8 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 $(\texttt{0x02}, \texttt{0x03}, \texttt{0x01}, \texttt{0x01})$, over $\mathrm{GF}(2^8)[x]$ 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[r] (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

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

where $i = 4, 5, \cdots, 43$.

For AES-192, the word W_i is generated by

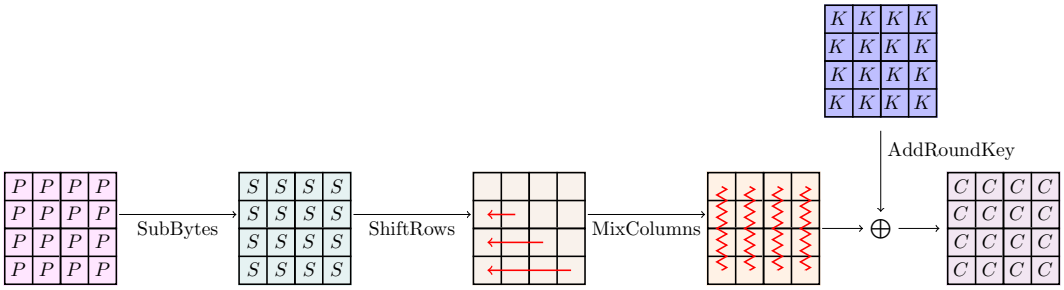
$$W_i = \begin{cases} W_{i-6} \oplus \mathrm{SubWord}(\mathrm{RotWord}(W_{i-1})) \oplus \mathrm{Rcon}[i/6], & \text{if } i \equiv 0 \bmod 6, \\ W_{i-6} \oplus W_{i-1}, & \text{otherwise,} \end{cases}$$

where $i = 6, 7, \cdots, 51$.

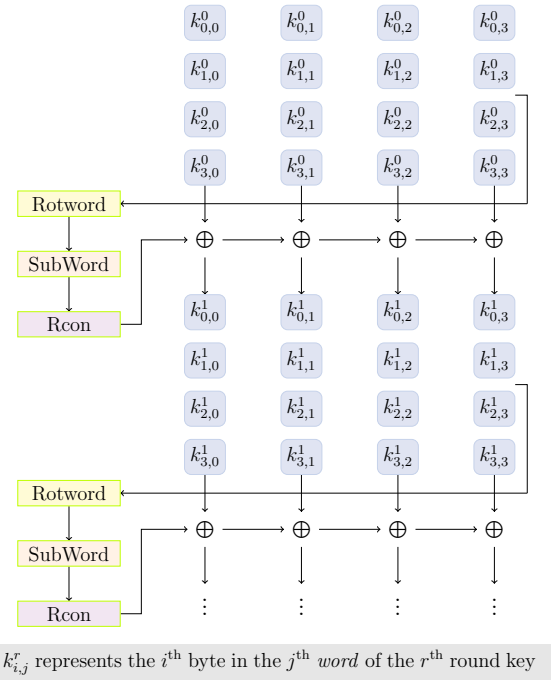
For AES-256, the word W_i is generated by

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

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 4×4 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 $\mathrm{GF}(2^8)$, which can equivalently be expressed as multiplication by a matrix of dimension 32×32 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 4×4 matrix). Thus, one MixColumns operation is equivalent to $4\times$ MixColumn operations on different columns in the matrix. Denoting the binary matrix corresponding to MixColumn as M with size 32×32, MixColumns can be represented as the diagonal matrix (M, M, M, M) of dimension 128×128 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 ith row to the left by i bytes, where $i = 1, 2, 3$. Thus, one ShiftRows operation is equivalent to $4\times$ ShiftRow operations on different rows in the 4×4 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 4×4 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 4×4 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 4×4 matrix.

S-box and S-box† 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† and uses input qubits to clean up ancilla qubits.

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

Rotation and Rotation† in Quantum Rotation (of AES) in quantum denotes the same RotWord. The reverse operation of Rotation is denoted as Rotation†.

SubWord and SubWord† in Quantum SubWord (of AES) in quantum denotes parallel operation for 4 S-boxes. We denote the reverse operation of SubWord as SubWord† (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[5], 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[2], t[3], t[12]); CNOT2(eng, t[5], t[10], t[13])
        CNOT2(eng, t[4], t[10], t[14]); CNOT2(eng, t[4], t[11], t[15])
        CNOT2(eng, t[8], t[15], t[16]); CNOT2(eng, u[4], u[0], t[17])
        CNOT2(eng, t[6], t[17], t[18]); CNOT2(eng, t[0], t[18], t[19])
        CNOT2(eng, u[1], u[0], t[20]); CNOT2(eng, t[6], t[20], t[21])
        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[18], u[0], m[3], resource_check)
        CNOT2(eng, m[3], m[0], m[4])
        Toffoli_gate(eng, t[2], t[15], m[5], resource_check)
        Toffoli_gate(eng, t[21], t[8], m[6], resource_check)
        CNOT2(eng, t[25], m[5], m[7])
        Toffoli_gate(eng, t[19], t[16], m[8], resource_check)
        CNOT2(eng, m[8], m[5], m[9])
        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[2], m[1], m[15])
        CNOT2(eng, m[4], t[23], m[16]); CNOT2(eng, m[7], m[6], m[17])
        CNOT2(eng, m[9], m[14], m[18]); CNOT2(eng, m[15], 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[25]); CNOT2(eng, m[19], m[20], m[26])
        CNOT2(eng, m[22], m[24], m[27])
        Toffoli_gate(eng, m[27], m[26], m[28], resource_check)
        Toffoli_gate(eng, m[25], m[23], m[29], resource_check)
        Toffoli_gate(eng, m[19], m[22], m[30], resource_check)
        Toffoli_gate(eng, m[26], m[30], m[31], resource_check)
        CNOT2(eng, m[26], m[24], m[32])
        Toffoli_gate(eng, m[20], m[21], m[33], resource_check)
        Toffoli_gate(eng, m[23], 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[35], m[39]); CNOT2(eng, m[37], m[39], m[40])
        CNOT2(eng, m[36], m[38], m[41]); 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[5], m[45], 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[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], l[4], l[10]); CNOT2(eng, m[59], l[2], l[11])
        CNOT2(eng, m[47], m[50], l[12]); CNOT2(eng, m[49], l[0], l[13])
        CNOT2(eng, m[51], m[60], l[14]); CNOT2(eng, m[54], l[1], l[15])
        CNOT2(eng, m[55], l[0], l[16]); CNOT2(eng, m[56], l[1], l[17])
        CNOT2(eng, m[57], l[8], l[18]); CNOT2(eng, m[62], l[4], l[19])
        CNOT2(eng, l[0], l[1], l[20]); CNOT2(eng, l[1], l[7], l[21])
        CNOT2(eng, l[3], l[12], l[22]); CNOT2(eng, l[18], l[2], l[23])
        CNOT2(eng, l[15], l[9], l[24]); CNOT2(eng, l[6], l[10], l[25])
        CNOT2(eng, l[7], l[9], l[26]); CNOT2(eng, l[8], l[10], l[27])
        CNOT2(eng, l[11], l[14], l[28]); CNOT2(eng, l[11], l[17], l[29])

    CNOT2(eng, l[6], l[24], s[7]); CNOT2(eng, l[16], l[26], s[6])
    CNOT2(eng, l[19], l[28], s[5]); X | s[6]; X | s[5]
    CNOT2(eng, l[6], l[21], s[4]); CNOT2(eng, l[20], l[22], s[3])
    CNOT2(eng, l[25], l[29], s[2]); CNOT2(eng, l[13], l[27], s[1])
    CNOT2(eng, l[6], l[23], s[0]); X | s[1]; X | s[0]

    # 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)
```

Code 1.2: Quantum circuit for AES MixColumn (regular and shallow versions).

```
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[15], x[31])
    CNOT | (x[4], x[12]); CNOT | (x[21], x[13])
    CNOT | (x[9], x[17]); CNOT | (x[27], x[11])
    CNOT | (x[28], x[4]); CNOT | (x[5], x[21])
    CNOT | (x[24], x[0]); CNOT | (x[7], x[15])
    CNOT | (x[1], x[9]); CNOT | (x[6], x[14])
    CNOT | (x[16], x[24]); CNOT | (x[22], x[6])
    CNOT | (x[31], x[16]); CNOT | (x[8], x[24])
    CNOT | (x[26], x[18]); CNOT | (x[30], x[22])
    CNOT | (x[10], x[26]); CNOT | (x[23], x[8])
    CNOT | (x[13], x[30]); CNOT | (x[29], x[13])
    CNOT | (x[13], x[5]); CNOT | (x[4], x[29])
    CNOT | (x[11], x[4]); CNOT | (x[19], x[11])
    CNOT | (x[12], x[13]); CNOT | (x[23], x[19])
    CNOT | (x[31], x[4]); CNOT | (x[20], x[12])
    CNOT | (x[12], x[28]); CNOT | (x[27], x[20])
    CNOT | (x[19], x[20]); CNOT | (x[31], x[27])
    CNOT | (x[15], x[12]); CNOT | (x[3], x[27])
    CNOT | (x[11], x[3]); CNOT | (x[2], x[11])
    CNOT | (x[18], x[19]); CNOT | (x[10], x[11])
    CNOT | (x[18], x[10]); CNOT | (x[2], x[18])
    CNOT | (x[9], x[10]); CNOT | (x[9], x[2])
    CNOT | (x[17], x[18]); CNOT | (x[25], x[17])
    CNOT | (x[17], x[1]); CNOT | (x[24], x[25])
    CNOT | (x[8], x[9]); CNOT | (x[15], x[24])
    CNOT | (x[15], x[11]); CNOT | (x[0], x[8])
    CNOT | (x[23], x[15]); CNOT | (x[16], x[17])
    CNOT | (x[0], x[16]); CNOT | (x[31], x[0])
    CNOT | (x[23], x[16]); CNOT | (x[6], 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[29], x[5])
    CNOT | (x[28], x[21]); CNOT | (x[21], x[29])
    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[4])
    CNOT | (x[19], x[3]); CNOT | (x[27], 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[9]); CNOT | (x[31], x[23])
    CNOT | (x[18], x[26]); CNOT | (x[6], 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[6]); x_out.append(x[31])

    x_out.append(x[16]); x_out.append(x[9])
    x_out.append(x[18]); x_out.append(x[19])
    x_out.append(x[20]); x_out.append(x[29])
    x_out.append(x[30]); 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).

```
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[35]); CNOT2(eng, x[14], x[31], t[71])
        CNOT2(eng, t[71], t[70], t[42]); CNOT2(eng, x[7], x[15], t[72])
        CNOT2(eng, t[72], t[71], t[37]); CNOT2(eng, x[0], x[17], t[73])
        CNOT2(eng, x[7], t[73], t[51]); CNOT2(eng, x[6], x[23], t[74])
        CNOT2(eng, x[7], t[74], t[39])
    CNOT2(eng, t[39], t[37], t[7]); CNOT2(eng, t[42], t[39], t[15])
    CNOT2(eng, t[44], t[39], t[31])

    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[14])

    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[6]); CNOT2(eng, t[35], t[34], t[22])
    CNOT2(eng, t[37], t[34], t[23])

    with Compute(eng):
        CNOT2(eng, x[4], 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[13]); CNOT2(eng, t[41], 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, t[82], t[84], t[68]); CNOT2(eng, x[16], x[23], t[85])
        CNOT2(eng, t[85], x[25], t[52]); CNOT2(eng, x[0], x[8], t[86])
        CNOT2(eng, x[31], t[86], t[45])
    CNOT2(eng, t[45], t[47], t[16]); CNOT2(eng, t[45], t[46], t[24])
```

```
    with Compute(eng):
        CNOT2(eng, t[85], t[86], t[49]); CNOT2(eng, x[2], x[10], t[87])
        CNOT2(eng, t[87], x[25], t[55])
    CNOT2(eng, t[55], t[57], t[18]); CNOT2(eng, t[55], t[56], t[26])

    with Compute(eng):
        CNOT2(eng, x[3], x[26], t[88]); CNOT2(eng, t[88], x[31], t[66])
    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[28])

    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[90], t[91], t[53])
    CNOT2(eng, t[53], t[51], t[1])

    with Compute(eng):
        CNOT2(eng, x[1], x[17], t[92]); CNOT2(eng, t[90], t[92], t[54])
    CNOT2(eng, t[54], t[52], t[9])

    with Compute(eng):
        CNOT2(eng, x[4], x[28], t[93]); CNOT2(eng, t[93], x[21], t[33])
    CNOT2(eng, t[36], t[33], t[5]); CNOT2(eng, t[32], t[33], t[29])

    with Compute(eng):
        CNOT2(eng, x[19], x[23], t[94]); CNOT2(eng, t[93], t[94], t[60])

    CNOT2(eng, t[60], t[61], t[20])

    with Compute(eng):
        CNOT2(eng, x[10], x[26], t[95]); CNOT2(eng, x[9], t[95], 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[10], x[27], t[97]); CNOT2(eng, x[15], t[97], t[67])
    CNOT2(eng, t[68], t[67], t[11])

    with Compute(eng):
        CNOT2(eng, x[11], x[20], t[98]); CNOT2(eng, x[15], t[98], t[62])

    CNOT2(eng, t[63], t[62], t[4]); CNOT2(eng, t[60], t[62], t[12])

    with Compute(eng):
        CNOT2(eng, x[11], x[19], t[99]); CNOT2(eng, x[2], x[7], 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[9], x[31], t[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[25])

#Computation end
```

```
#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)

    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 9: Quantum resources required per round for variants of AES (this work).

Cipher	Round	#CNOT			#NOT	#Toffoli		Toffoli depth
		☆	⌓	✠	☆⌓✠	☆	⌓✠	☆⌓✠
AES-128	1≀	7,920	4,544	5,568	79	1,360	680	12 (☆), 6 (⌓✠)
	2	7,792	7,920	8,944	79	1,360	1,360	12
	3	7,792	7,920	8,944	81	1,360	1,360	12
	4	7,792	7,920	8,944	81	1,360	1,360	12
	5	7,792	7,920	8,944	81	1,360	1,360	12
	6	7,792	7,920	8,944	79	1,360	1,360	12
	7	7,792	7,920	8,944	79	1,360	1,360	12
	8	7,792	7,920	8,944	81	1,360	1,360	12
	9	7,792	7,920	8,944	80	1,360	1360	12
	10	3,984	4,048	4,048	80	680	680	6
AES-192	1≀	7,984	4,576	5,600	79	1,360	680	12 (☆), 6 (⌓✠)
	2	7,856	7,952	8,976	79	1,360	1,360	12
	3	6,256	7,008	8,032	64	1,088	1,224	12
	4	7,856	7,200	8,224	81	1,360	1,224	12
	5	7,856	7,952	8,976	81	1,360	1,360	12
	6	6,256	7,008	8,032	64	1,088	1,224	12
	7	7,856	7,200	8,224	81	1,360	1,224	12
	8	7,856	7,952	8,976	79	1,360	1,360	12
	9	6,256	7,008	8,032	64	1,088	1,224	12
	10	7,856	7,200	8,224	79	1,360	1,224	12
	11	7,168	7,952	8,976	81	1,224	1,360	12
	12	3,136	3,136	3,136	64	544	544	6
AES-256	1≀	6,384	3,632	4,656	64	1,088	544	12 (☆), 6 (⌓✠)
	2	7,792	7,104	8,128	79	1,360	1,224	12
	3	7,792	7,792	8,816	80	1,360	1,360	12
	4	7,792	7,792	8,816	79	1,360	1,360	12
	5	7,792	7,792	8,816	80	1,360	1,360	12
	6	7,792	7,792	8,816	81	1,360	1,360	12
	7	7,792	7,792	8,816	80	1,360	1,360	12
	8	7,792	7,792	8,816	81	1,360	1,360	12
	9	7,792	7,792	8,816	80	1,360	1,360	12
	10	7,792	7,792	8,816	81	1,360	1,360	12
	11	7,792	7,792	8,816	80	1,360	1,360	12
	12	7,792	7,792	8,816	79	1,360	1,360	12
	13	7,792	7,792	8,816	80	1,360	1,360	12
	14	3,984	3,984	3,984	79	680	680	6

☆: Regular version (using MixColumn from [56]).
⌓: Shallow version (using MixColumn from [56]).
✠: Shallow/low depth version (using MixColumn from [46]).
≀: Including initial key XOR.