# Side-Channel Analysis of Privacy Amplification in Postprocessing Software for a Quantum Key Distribution System

## Technical Report TUD-CS-2018-0024

**January 2018**

Oleg Nikiforov[1], Alexander Sauer[2], Johannes Schickel[3], Alexandra Weber[3], Gernot Alber[2], Heiko Mantel[3], Thomas Walther[1]

Technische Universität Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Laser and Quantum Optics[1],
Theoretical Quantum Physics[2],
and Modeling and Analysis
of Information Systems[3]

This work has been funded by the DFG as part of the projects P4 (“Quantum Key Hubs”) and E3 (“Secure Refinement of Cryptographic Algorithms”) within the CRC 1119 CROSSING.

CROSSING

# Side-Channel Analysis of Privacy Amplification in Postprocessing Software for a Quantum Key Distribution System

Oleg Nikiforov[1], Alexander Sauer[1], Johannes Schickel[2], Alexandra Weber[2], Gernot Alber[1], Heiko Mantel[2], Thomas Walther[1]

[1] Physics Department, TU Darmstadt, Germany
{gernot.alber, oleg.nikiforov, alexander.sauer, thomas.walther}@physik.tu-darmstadt.de

[2] Computer Science Department, TU Darmstadt, Germany
mantel@cs.tu-darmstadt.de, {schickel,weber}@mais.informatik.tu-darmstadt.de

**Abstract.** Quantum key distribution is an alternative to the classical way of distributing secret cryptographic keys. Due to imperfections of existing hardware setups for quantum key distribution, postprocessing in software is needed to correct errors in the exchanged key and to amplify its privacy. We analyze an implementation of privacy amplification for quantum key distribution with respect to cache side channels, using program analysis. Our main result is that no information about the secret key is leaked through cache side channels in this implementation.

## 1 Introduction

The development of quantum computers endangers the security of most modern asymmetric cryptographic schemes. The reason are quantum algorithms that provide a huge speed-up of the calculation of classical hard problems. Examples are prime-number factorization and computation of discrete logarithms (Shor's algorithm [26]).

One important application of cryptography is the secure exchange of cryptographic keys. In this scenario, quantum physics provides a possible solution to the threat of improved computational power. Quantum key distribution (QKD) constitutes an approach for the distribution of cryptographic keys, requiring specialized hardware. In contrast to classical key distribution schemes, which base their security on the computational hardness of mathematical problems, its security is based on the laws of quantum physics. Specifically, an adversary reveals himself in the attempt to eavesdrop on a communication session.

By combining QKD with the one-time pad [28], information-theoretically secure exchange of data can be ensured [25].

When implemented in hardware, QKD systems may suffer from a large number of weaknesses due to imperfect setup devices, e.g., imperfect detectors and quantum bit sources or channel noise. In order to guarantee the security of such QKD systems, researchers actively search for weaknesses in existing experimental setups, using contemporary hardware, as well as in potential future setups.

The analysis for weaknesses in a setup has to be completed on each part of the implementation. Not only on the physical level, but also on the software level responsible for the last part of key distribution - error correction, privacy amplification and secure key storage and processing.

In the present report, we analyze a simplified version of an existing software implementation of privacy amplification for a QKD system using the BB84 protocol [2]. We focus on the security of this implementation with respect to cache-side-channel attacks. Cache-side-channel attacks allow an attacker to deduce secret information, e.g., about cryptographic keys, from the interaction of a software implementation with the cache.

In Section 2, we briefly recall preliminaries on the BB84 protocol and cache-side-channel analysis. In Section 3, we describe the original implementation of privacy amplification on which we base and our simplifications of the implementation for the analysis. Finally, we present our analysis results in Section 4 and conclude in Section 5.

## 2 Preliminaries

QKD relies on the exchange of quantum objects carrying information called qubits (quantum bits) over a channel that does not disturb the state of the qubit and therefore the encoded information (quantum channel). The distribution of qubits requires special hardware allowing for preparation of specific quantum states, encoding the information within the phase or polarization of those objects and their noiseless distribution and detection, i.e. the correct extraction of the encoded information. Security proofs of QKD typically assume that all signal distortions are accounted to an eventual eavesdropper. The eavesdropper is assumed to have access to any theoretically possible physical device helping to compromise the QKD system. The security proofs give certain constraints for the noise level in the quantum channel of the setup. If the error rate of the protocol is below a certain threshold, information-theoretic security of the QKD session can be guaranteed. Then only run-time attacks, which occur during the key exchange, on the involved devices could be successful.

The first part of a quantum key exchange is the distribution and measurement of quantum objects, the so-called raw key exchange. As a result, both parties obtain a list of bits, which could contain errors. In the second step, the postprocessing, pure classical algorithms are applied to the raw key.

*Raw Key Exchange.* The first protocol using quantum objects as information carriers (qubits) was proposed by Bennet and Brassard in 1984 (BB84) [2]. The information is encoded in the polarization or phase of single photons, which

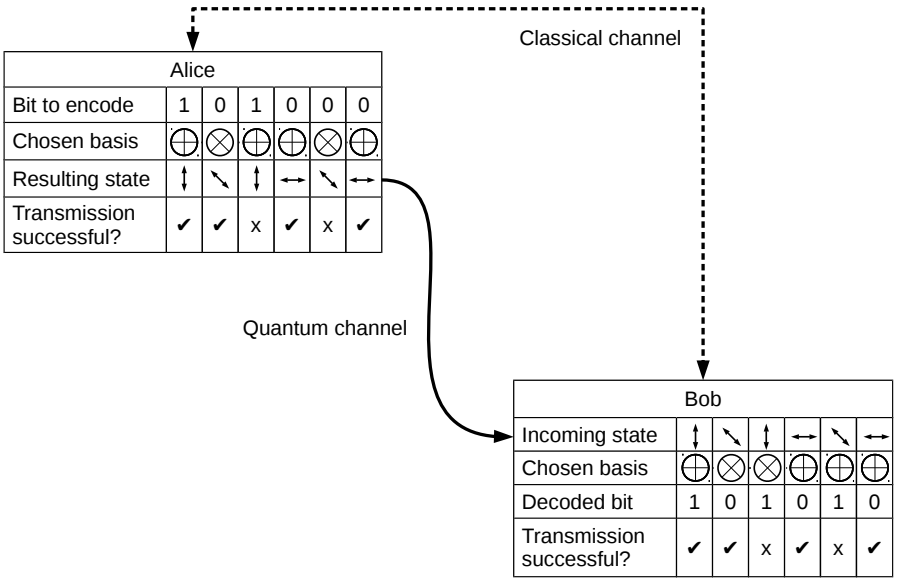


**Fig. 1.** Schematic of BB84 protocol. Alice encodes classical bits in single photon states using polarization encoding within one of two bases $\oplus$ or $\otimes$. Bob tries to guess the chosen basis and detects the photon. If the chosen bases are not equal, the transmission is not successful and corresponding bits are discarded.

are sent through a quantum channel to the recipient. Although this protocol is the oldest one, it remains (with some improvements) one of the protocols most commonly implemented by experimental researchers.

The scheme of the protocol is shown in Fig. 1. The sender Alice desires to send a private message to recipient Bob. Alice distributes a cryptographic key to Bob, using polarization of single photons. She then encodes her message using the secret key and AES or one-time pad, for example.

To distribute the key, Alice chooses randomly one of two possible polarization bases: rectilinear $\oplus$ or diagonal $\otimes$ and encodes a random bit within the chosen basis [3]. For example, bit 1 is then encoded as polarization $\nearrow$ or $\uparrow$, with respect to the used basis and 0 is encoded as $\nwarrow$ or $\rightarrow$. Then, the prepared photon is sent to Bob over the quantum channel.

Bob selects randomly and independently of Alice one of the two bases and detects the incoming photon. If Bob chooses the right basis, the obtained bit of information is identical to the bit Alice encoded. In case of the wrong basis,

[3] For more details please see [20].

Bob's result has a probability of 0.5 to be incorrect due to quantum-mechanical calculation. After a series of qubit exchanges, Alice and Bob obtain a long string of bits - the *raw key*.

During the next step, they publicly announce the used bases, corresponding to each measured photon, over a classical channel. In the case of non-matching bases, they discard the corresponding bits and obtain finally a *sifted key*. In the ideal case of error-free transmission of qubits and absence of eavesdroppers the sifted key would be error-free.

During a realistic quantum key distribution some distortions are inevitable. All errors indicate an information loss, which has to be treated as information gain by an attacker. Therefore, the next step is the quantum bit error rate (QBER) estimation. Alice and Bob randomly choose a subset of their results, compare the bits and remove them from the sifted key. If the errors remain below a certain threshold [27], the parties can generate a secure key with post-processing algorithms for error correction and privacy amplification. In the other case, Alice and Bob should restart the raw key exchange.

*Post-Processing.* After the QBER estimation, an error correction step is performed to remove the errors from the raw key. The three most widely-used algorithms for this are: *low density parity check* [10], *cascade* [5] or *polar codes* [14]. During such an error correction, Alice and Bob reveal more information about their key to a possible adversary, simultaneously getting rid of the errors in the key. Subsequently, a privacy amplification step is performed to decrease an attacker's information about the key and to provide a secure key for Alice and Bob.

*Attacks on BB84.* To check the security of the quantum key distribution, several types of theoretical and experimental attacks have been developed. Even though the principle of BB84 is proven to be secure, all implementations of this algorithm suffer from imperfections in real devices, e.g., single photon sources, detectors. One of the most famous attacks was performed by Lydersen in 2010 [17]. He could manipulate the data received by Bob using bright light injected into the quantum channel.

In this report, we focus on *side-channel vulnerabilities*. Side-channel vulnerabilities might allow an attacker to deduce the key from characteristics of the apparatus used for the raw key exchange or from execution characteristics of the software used for postprocessing. Such attacks compromise the key despite security proofs. Finding and getting rid of side-channel vulnerabilities is a serious issue for the security of QKD implementations.

*Cache-Side-Channel Vulnerabilities.* Multiple types of side-channel attacks on software exist, e.g., measuring running time [15] or power consumption [16]. In this report, we focus on cache side channels. Caches are small memories in computers, which are used to store selected entries from the main memory for performance reasons. The memory entries stored in the cache can be accessed more quickly by the processor than the memory entries that are not stored in the

cache. If the CPU accesses an entry that is stored in the cache, it encounters a so-called cache hit. If the CPU accesses an entry that is not stored in the cache, it encounters a so-called cache miss. If the memory entries that a software accesses depend on secret information (like parts of a secret key), the timing difference between cache hits and cache misses might give rise to so-called cache-side-channel vulnerabilities. More concretely, an attacker who observes the execution time [3], the trace of cache hits and misses [1], or the contents of a shared cache [22,11] might be able to deduce secret information from his observations.

*Program Analysis against Cache Side Channels.* Multiple techniques to verify the security of software against cache side channels exist. One approach is to create new, verified software implementations as in [23]. To ensure the security of existing implementations, program analyses can be used to detect side-channel vulnerabilities, e.g, [7], and to quantify their seriousness, e.g., [19]. To compute upper bounds on the cache-side-channel vulnerabilities in x86 binaries of software [4], a static reachability analysis can be used in combination with information theory [8,19]. More concretely, reachability analysis can be performed to determine the possible different observations that an attacker can make about the cache (e.g., possible traces of cache hits and misses). The amount of secret bits leaked to an attacker through a cache-side-channel vulnerability can then be bounded by the logarithm of the number of different possible observations [8]. The tool CacheAudit 0.2c [4], which is based on previous versions of CacheAudit [8,19], takes as input x86 binaries and outputs bounds on the cache-side-channel vulnerability of the x86 binary. More concretely, it returns bounds on the vulnerability of the software to four cache-side-channel attacker models: a model under which an attacker can observe the amount of entries in the cache after a run of the binary; a model in which an attacker can observe the exact entries in the cache after a run of the binary, a model under which an attacker can observe the trace of cache hits and misses that occurred during a run of the binary, and a model under which an attacker can observe the overall time taken for cache hits and misses during a run of the binary. [5] If CacheAudit 0.2c returns bounds of 0 bit on the side-channel vulnerability of a binary, the binary is secure with respect to these four attacker models. In this report, we use CacheAudit0.2c to analyze a implementation of the privacy amplification step in QKD postprocessing.

*Privacy Amplification.* During the error correction session or measurement process of a QKD protocol, Eve could have gained information about the key. The secure key, that Alice and Bob want to use in the end, should be error free and completely unknown to the possible attackers. After a privacy amplification session, any information the attacker Eve might have about the key should vanish. To this end, a cryptographic hash function can be used [12].

For two given sets $U$ and $V$, a function $f : U \to V$ is called a *hash function*, if $|V|$ is fixed. For cryptographic purposes, it should be a one-way function. The

[4] Representations of the software for CPUs with x86 instruction set architecture
[5] For formal definitions of the attacker models, please see [8].

calculation of a function value should be quick and the results identical, if the same parameter is used. The inversion of the hash function should be as hard to calculate as in a brute-force attempt. Also, the probability for collisions of two hash values should be as low as possible. One can find a family $H$ of hash functions with a probability for collisions of $1/|V|$:

$$Pr(f(x) = f(y)) \leq \frac{1}{|V|} \text{ for all } x, y \in U \text{ with } x \neq y \text{ for any } f \in H.$$

In this case, $H$ is called a family of two-universal hash functions. The name emphasizes, that this property of $H$ applies to any pair of elements in $U$ [6]. Applying a two-universal hash function to an error corrected key creates a secure key of length $l$, which depends on the adversary's information about the error-corrected key [24].

*Toeplitz Matrices.* In the implementation considered in this report, multiplication with Toeplitz matrices is used as a family of two-universal hash functions [9,18]. An $l \times m$ matrix $M$ is called Toeplitz matrix, if the values in the main and each of the secondary diagonals are equal. Thus, Toeplitz matrices can be described by a sequence of $m+l-1$ entries $a_i$, which are arranged as follows [9]:

$$M = \begin{pmatrix} a_0 & a_{-1} & a_{-2} & \cdots & a_{-m+1} \\ a_1 & a_0 & a_{-1} & \ddots & \\ a_2 & a_1 & a_0 & \ddots & \vdots \\ \vdots & \vdots & \ddots & & \\ a_{l-1} & & \cdots & & a_{l-m} \end{pmatrix}$$

With an $l \times m$ Toeplitz matrix containing binary entries only, a secure key of the length $l$ can be extracted from an partially compromised key of the length $m$ by simple matrix multiplication. When choosing the parameter $l$, the estimated information obtained by Eve during the key distribution and error correction phases has to be taken into account.

## 3 Analyzed Implementation of Privacy Amplification

In this report we analyze the C++ implementation of privacy amplification from [21] with respect to cache side channels.

### 3.1 Overview of the Implementation from [21]

In the C++ code, privacy amplification is realized in the class `PrivAmp`. Listing 1.1 gives an overview of this class. When generating a `PrivAmp` object, the final length $l_{pa} = |k_{pa}|$ of the privacy amplified key $k_{pa}$ has to be specified.

**Listing 1.1.** Overview of PrivAmp

```
1  // private variables: Toeplitz matrix, privacy amplified key
   and the respective lengths
2          char* toepMat[toepMatLen];
3          int toepMatLen;
4          char* paKey[paLen];
5          int paLen;
6  // variable manipulation
7          void setPAKeyLength(int paKeyLength);
8          void setToeplitzMat(char* toeplitzMat);
9          char* getToeplitzMat();
10         int getToeplitzMatLen();
11         char* getPAKey();
12         int getPAKeyLen();
13 // generating the Toeplitz matrix
14         void generateToeplitzMat();
15 // calculating the final privacy amplified key
16         void calcPAKey(bool* key);
```

On initialization, the Toeplitz matrix and privacy amplified key are created as empty char arrays `toepMat` and `paKey` of length $l_{TM} = l_{pa} + l_{Key} - 1$ and $l_{pa}$ respectively. Here, $l_{Key}$ is the global key length of 1000, as the key is divided into blocks of length 1000 for post-processing. The $l_{pa} \times l_{Key}$ Toeplitz matrix $TM$ can be represented by $l_{TM}$ entries because each entry $TM_{ij}$ depends only on $i - j$.

The function `generateToeplitzMat()` fills `toepMat` with 0 and 1 using $rand()\%2$ in a for-loop.

The final privacy amplified key is generated by calling `calcPAKey(bool* ECKey)` with the error corrected key. In this function, the matrix multiplication is implemented by multiplying each element $(k_{ec})_j$ of the error corrected key $k_{ec}$ with the appropriate element $id = i - j + l_{Key} - 1$ of `toepMat` and adding the results modulo 2 to get $(k_{pa})_i$ of the final key.

In the full program, after finishing error correction, the `ThreadMgr` calls its function `doPrivacyAmplification`. This creates a `PrivAmp` object and, if it acts as Alice, calls `generateToeplitzMat()` and sends this `toepMat` to Bob. Afterwards both, Alice and Bob, generate the final key with `calcPAKey`.

### 3.2 Simplified Implementation for Analysis

To quantify the vulnerability of the privacy-amplification implementation with respect to cache side channels, we analyzed the function `calcPAKey` with the tool CacheAudit0.2c.

CacheAudit0.2c was not directly applicable to the original implementation of `calcPAKey` shown in Listing 1.2. We simplified the implementation accordingly to

enable an analysis with CacheAudit0.2c. Our adapted implementation is shown in Listing 1.3, where adapted lines are highlighted in gray.

Firstly, the object-oriented variant of privacy amplification is currently not analyzable with CacheAudit0.2c. Thus, we removed the object orientation from the implementation (from line 3).

Secondly, the removal of object orientation raises the question how to deal with class attributes. Since global variables are not supported by the analysis tool, we store the information previously stored in class attributes in local variables. More concretely, we replaced the class attributes `paLen` (length of the privacy-amplified key), `toepMat` (Toeplitz matrix), and `paKey` (location to store the privacy-amplified key). In the modified implementation, we pass the length of the privacy amplified key to the function `calcPAKey` as a parameter (see line 3). We create local variables for both, the Toeplitz matrix and the array that stores the resulting privacy-amplified key (see lines 4-6). To keep the analysis results independent of a concrete Toeplitz matrix, we initialize the Toeplitz matrix with the value `*((int *)0x4) & 1` (see line 7). The reason is that the analysis tool cannot determine the value stored at address `0x4` and will thus overapproximate the value by considering all possible values. Since the Toeplitz matrix contains only binary entries, only values of length 1 bit need to be considered. We achieve this by masking the value at address `0x4` with `1`.

Finally, we call the function `calcPAKey` from a wrapper function shown in Listing 1.4. As parameters, we pass an uninitialized key and the global key length 1000[6]. Using an uninitialized key causes the analysis tool to treat the key as secret information whose leakage through cache-side-channel vulnerabilities should be quantified. We compiled the modified implementation using the command

```
gcc PrivAmp.cpp -m32 -fno-stack-protector -g
```

and obtained an x86 binary, which we then analyzed.

## 4 Analysis Against Cache Side Channels

The x86 binary we obtained from the modified implementation of `calcPAKey` contains the x86 instructions `SAR` (shift arithmetic right, opcode 0xC0/111) and `SHR` (shift logical right, opcode 0xC0/101). These two instructions were not supported by CacheAudit0.2c. We extended the analysis tool to also support the analysis of these instructions.

We applied the extended analysis tool, using the flag `--unroll 1000000`, to the binary of the modified implementation of privacy amplification. We configured the tool to assume the popular cache replacement strategy LRU (least recently used). For the cache configuration, we fixed parameters that are used, e.g., in the first level cache of the Intel Skylake architecture [13], namely a 32 kByte, 8-way set-associative data cache with a cache-line size of 64 Byte. The analysis results we obtained are shown in Listing 1.5.

[6] 1000 is the default length of key blocks in the original implementation.

**Listing 1.2.** Original implementation of privacy amplification

```
1  #define KEYLENGTH 1000
2    [...]
3  void qkdtools::PrivAmp::calcPAKey(bool* key){
4    //matrix multiplication
5    for(int i=0;i<paLen;i++){
6      paKey[i]=0;
7      for(int j=0;j<KEYLENGTH;j++){
8        //id of Toeplitz sequence for matrix element i,j
9        int id=i-j+KEYLENGTH-1;
10       paKey[i]+=toepMat[id] * key[j];
11       //keep it binary (mod 2)
12       paKey[i]=paKey[i]%2;
13     }
14   }
15 }
```

**Listing 1.3.** Modified implementation of privacy amplification

```
1  #define KEYLENGTH 1000
2  [...]
3  void calcPAKey(bool* key, int paKeyLength){
4    int toepMatLen=KEYLENGTH + paKeyLength - 1;
5    char toepMat[toepMatLen];
6    char paKey[paKeyLength];
7    for(int i=0;i<toepMatLen;++i){toepMat[i]=*((int *)0x4) & 1; }
8    //matrix multiplication
9    for(int i=0;i<paKeyLength;i++){
10     paKey[i]=0;
11     for(int j=0;j<KEYLENGTH;j++){
12       //id of Toeplitz sequence for matrix element i,j
13       int id=i-j+KEYLENGTH-1;
14       paKey[i]+=toepMat[id] * key[j];
15       //keep it binary (mod 2)
16       paKey[i]=paKey[i]%2;
17       }
18     }
19 }
```

**Listing 1.4.** Wrapper for privacy amplification

```
1  int main(int argc, char *argv[]) {
2    bool key[KEYLENGTH];
3    calcPAKey(key, KEYLENGTH);
4    return 0;
5  }
```

**Listing 1.5.** Analysis results for modified privacy amplification

```
1 Number of valid cache configurations: 1, (0.000000 bits)
2 Number of valid cache configurations (blurred):
3                                       1, (0.000000 bits)
4 # traces: 1, 0.000000 bits
5 # times: 1.000000, 0.000000 bits
6
7 Analysis took 408143 seconds.
```

The analysis results are 0 bit of leakage through cache-side-channel vulnerabilities with respect to all four cache-side-channel attacker models we consider:

- the attacker model where an attacker can observe the time taken for cache hits and misses during a run of privacy amplification,
- the attacker model where an attacker can observe the trace of cache hits and misses during a run of privacy amplification,
- the attacker model where an attacker can observe the amount of memory entries in the cache after a run of privacy amplification, and
- the attacker model where an attacker can observe the contents of the cache after a run of privacy amplification.

Hence, no secret information about the secret key is leaked to attackers under these attacker models when running the simplified implementation of privacy amplification.

## 5 Conclusion

In this report, we considered a simplified implementation of privacy amplification, the last step in the postprocessing of a quantum key distribution. We used program analysis to compute upper bounds on the cache-side-channel leakage with respect to four attacker models. The leakage is bounded by 0 bit for all four attacker models. That is, a cache-side-channel attacker under any of these models cannot learn information about the secret key through a cache side channel. The security of the privacy amplification step in QKD postprocessing is particularly important because privacy amplification is the final step in the postprocessing and no subsequent step can make up for leakage in this step.

*Acknowledgements.* We thank Pascal Notz, who developed the implementation of privacy amplification that we analyze. This work has been funded by the DFG as part of the projects P4 (“Quantum Key Hubs”) and E3 (“Secure Refinement of Cryptographic Algorithms”) within the CRC 1119 CROSSING.

## References


1. Aciiçmez, O., Koç, Ç.K.: Trace-Driven Cache Attacks on AES (Short Paper). In: Proceedings of the 8th International Conference on Information and Communications Security (ICICS). pp. 112–121 (2006)
2. Bennett, C.H., Brassard, G.: Quantum cryptography: Public key distribution and coin tossing. In: Proceedings of IEEE International Conference on Computers, Systems, and Signal Processing. pp. 175–179 (1984)
3. Bernstein, D.J.: Cache-timing attacks on AES. Tech. rep., University of Illinois at Chicago (2005)
4. Bindel, N., Buchmann, J., Krämer, J., Mantel, H., Schickel, J., Weber, A.: Bounding the cache-side-channel leakage of lattice-based signature schemes using program semantics. In: Postproceedings of the 10th International Symposium on Foundations & Practice of Security (FPS) (2017), To appear.
5. Brassard, G., Salvail, L.: Secret-key reconciliation by public discussion. In: Proceedings of the 12th Workshop on the Theory and Application of of Cryptographic Techniques (EUROCRYPT). pp. 410–423 (1993)
6. Carter, J.L., Wegman, M.N.: Universal classes of hash functions. Journal of Computer and System Sciences 18(2), 143 – 154 (1979)
7. Dewald, F., Mantel, H., Weber, A.: AVR Processors as a Platform for Language-Based Security. In: Proceedings of the 22nd European Symposium on Research in Computer Security (ESORICS) - Part I. pp. 427–445 (2017)
8. Doychev, G., Köpf, B., Mauborgne, L., Reineke, J.: Cacheaudit: A tool for the static analysis of cache side channels. ACM Transactions on Information and System Security 18(1), 4:1–4:32 (2015)
9. Fung, C.H.F., Ma, X., Chau, H.F.: Practical issues in quantum-key-distribution postprocessing. Phys. Rev. A 81(1), 012318 (2010)
10. Gallager, R.: Low-density parity-check codes. IRE Transactions on information theory 8(1), 21–28 (1962)
11. Gullasch, D., Bangerter, E., Krenn, S.: Cache Games - Bringing Access-Based Cache Attacks on AES to Practice. In: Proceedings of the 32nd IEEE Symposium on Security and Privacy (S&P). pp. 490–505 (2011)
12. Impagliazzo, R., Levin, L.A., Luby, M.: Pseudo-random generation from one-way functions. In: Proceedings of the 21st annual ACM symposium on Theory of computing (STOC). pp. 12–24 (1989)
13. Intel Corporation: Intel® 64 and IA-32 Architectures Optimization Reference Manual. Order Number: 248966-032 (2016)
14. Jouguet, P., Kunz-Jacques, S.: High performance error correction for quantum key distribution using polar codes. Quantum Information & Computation 14(3-4), 329–338 (2014)
15. Kocher, P.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: Proceedings of the 16th Annual International Cryptology Conference (CRYPTO). pp. 104–113 (1996)
16. Kocher, P., Jaffe, J., Jun, B.: Differential Power Analysis. In: Proceedings of the 19th Annual International Cryptology Conference (CRYPTO). pp. 388–397 (1999)
17. Lydersen, L., Wiechers, C., Wittmann, C., Elser, D., Skaar, J., Makarov, V.: Hacking commercial quantum cryptography systems by tailored bright illumination. Nature Photonics 4(10), 686–689 (2010)
18. Mansour, Y., Nisan, N., Tiwari, P.: The computational complexity of universal hashing. In: Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (STOC). pp. 235–243 (1990)

19. Mantel, H., Weber, A., Köpf, B.: A Systematic Study of Cache Side Channels across AES Implementations. In: Proceedings of the 9th International Symposium on Engineering Secure Software and Systems (ESSoS). pp. 213–230 (2017)
20. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information: 10th Anniversary Edition. Cambridge University Press, New York, NY, USA (2011)
21. Notz, P.: Implementing a Post-Processing Procedure for Quantum Cryptography. Bachelor's thesis, Technische Universität Darmstadt (2012)
22. Osvik, D.A., Shamir, A., Tromer, E.: Cache Attacks and Countermeasures: The Case of AES. In: Proceedings of the Cryptographer's Track at the 2006 RSA Conference (CT-RSA). pp. 1–20 (2006)
23. Protzenko, J., Zinzindohoué, J.K., Rastogi, A., Ramananandro, T., Wang, P., Zanella-Beguelin, S., Delignat-Lavaud, A., Hriţcu, C., Bhargavan, K., Fournet, C., Swamy, N.: Verified Low-Level Programming Embedded in F*. Proceedings of the ACM on Programming Languages 1(ICFP), 17:1–17:29 (2017)
24. Renner, R.: Security of quantum key distribution. International Journal of Quantum Information 6(01), 1–127 (2008)
25. Shannon, C.E.: Communication theory of secrecy systems. Bell Labs Technical Journal 28(4), 656–715 (1949)
26. Shor, P.W.: Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. SIAM J. Comput. 26(5), 1484–1509 (1997)
27. Shor, P.W., Preskill, J.: Simple proof of security of the BB84 quantum key distribution protocol. Physical review letters 85(2), 441–444 (2000)
28. Vernam, G.S.: Cipher printing telegraph systems: For secret wire and radio telegraphic communications. Journal of the A.I.E.E. 45(2), 109–115 (1926)