Cache-Side-Channel Quantification and Mitigation for Quantum Cryptography

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

[1] Department of Computer Science, Technical University of Darmstadt
[2] Department of Physics, Technical University of Darmstadt

Abstract. Quantum cryptography allows one to transmit secret information securely, based on the laws of quantum physics. It consists of (1) the transmission of physical particles like photons and (2) the software-based processing of measurements during the transmission. Quantum key distribution (QKD), e.g., transmits material for establishing a shared crypto key in this way. The key material is encoded into the particles in a way that leakage can be detected and mitigated via so-called privacy amplification.

In this article, we investigate the role of the software implementation for the security of quantum cryptography. More concretely, we quantify the security of QKD software against cache side channels and show how to integrate cache-side-channel mitigation with the privacy amplification in QKD. We evaluate our approach at one variant of a QKD software that is in practical use. During our evaluation, we detect a cache-side-channel vulnerability, for which we develop a parametric mitigation that combines privacy amplification and program rewriting. We propose a cost model for the combined mitigation, which allows one to optimize the interaction between privacy amplification and program rewriting for the mitigation.

1 Introduction

Quantum cryptography [31,56] is a promising approach to protect secret communication. It is based on the laws of quantum physics, which ensure that an attacker will be detected if he intercepts the communication. Quantum crypto is fundamentally different from post-quantum crypto, which is based on mathematical problems that are hard to solve even with quantum computers.

A prominent example of quantum crypto is Quantum Key Distribution (QKD). QKD is a promising candidate for post-quantum key exchange [30]. The German government, e.g., is investing 165M€ in a QKD network among governmental agencies [25] and Quapital [36] aims at a QKD network across Europe. In China, a QKD network between Beijing, Shanghai and other cities was established in 2018 and is in a trial period for applications like banking [78].

QKD consists of a physical part and a software part. In the physical part, the key material on Alice's side is encoded into quantum properties of particles, usually photons. The particles are transmitted to Bob via a so-called quantum

Published in:
E. Bertino et al. (Eds.): ESORICS 2021, LNCS 12973, pp. 235–256, 2021.

The final authenticated version is available online at
https://doi.org/10.1007/978-3-030-88428-4_12

channel, and Bob measures their properties to recover the key material. In the software part, the key material on both sides is converted into a shared key.

Information leakage during the physical part can be quantified based on traces that an attacker inevitably leaves in intercepted particles. It is mitigated by privacy amplification, which increases the number of particles and compresses the resulting key in the software part. Information leakage during the software part can be quantified using Quantitative Information Flow (QIF) [3,70] and mitigated by privacy amplification or, more traditionally, by program rewriting.

When the physical and software parts are combined, new threats arise. For instance, an attacker might combine attacks on the physical part with cache-side-channel attacks, which exploit that a software unintentionally uses a shared cache in a secret-dependent way [1,7,33,43,59,69,77]. In QKD setups like [67] (for secure video conferencing at Mitsubishi) or [68], it is common to use regular PCs for postprocessing. PCs, especially those that process measurements, are often multi-purpose. That is, when setups like [67,68] go into production, communication, e.g., of corporate secrets, might be at risk because the postprocessing software for the encryption key might share a cache with other software. Given efforts to move QKD to the cloud [11,53] and to end-user devices like smartphones [72], cache side channels will be even more dangerous for future QKD.

In this article, we focus on two research questions: (1) How can cache-side-channel mitigation by program rewriting be integrated with privacy amplification? (2) What is a good split between program rewriting and privacy amplification from a performance perspective? Being able to integrate both mitigation techniques in a reliable and cost-efficient fashion is crucial for achieving end-to-end security for QKD across quantum physics and software in practice.

Our first step is to quantify the cache side channels in QKD software. To this end, we develop a program analysis that computes upper leakage bounds. The main novelty of our analysis is that it supports x86 binaries with floating-point instructions. Prior analyses that compute reliable cache-side-channel leakage bounds for x86 binaries (including [8,19,20,51]) did not support floating-point instructions. As QKD software inherently deals with probabilities, e.g., to process imperfect measurements, floating-point support is crucial in this domain.

We evaluate our analysis at the example of the BB84 protocol for QKD [6]. More concretely, we analyze the BB84 implementation from [42,57], which is one of the few publicly available QKD implementations. It uses the low-density parity check (LDPC) technique, which is very popular in QKD [29,34,38,52].

In our evaluation, we discovered a vulnerability in a part of the implementation that optimizes the LDPC. An attacker might recover the entire secret key from one cache trace. We show how to harden the implementation by program rewriting and use our analysis to confirm that the hardening is, indeed, effective.

Naturally, the program rewriting induces performance overhead. To increase flexibility in trading performance against security, we develop a mitigation technique that combines selective program rewriting with privacy amplification. Given a set of key bits that might be leaked by the software, we cannot simply identify the corresponding particles and drop them from the transmission. The

reasons are (1) that the postprocessing relies on relations across multiple bits, e.g., that a certain percentages of bits survives the first postprocessing phase (key-sifting phase) and (2) that privacy amplification relies on hash-functions, i.e., there is no one-to-one mapping between the original bits and the final key bits. Therefore, we use our program analysis to show how much privacy amplification is needed to mitigate potential leakage that remains after selective rewriting. Finally, we develop a cost model for the combined mitigation technique and evaluate it for realistic QKD-setup parameters.

In summary, our main technical contributions are (1) a static program analysis that computes reliable upper bounds on the cache-side-channel leakage of programs that use floating-point operations, [3] (2) an evaluation on real-world QKD software, (3) the detection and mitigation of a vulnerability in the QKD software that might leak the entire key, (4) a mitigation technique that integrates program rewriting with privacy amplification, and (5) an evaluation how to optimize performance across program rewriting and privacy amplification.

2 Basic Notions and Notation

2.1 Cache-Side-Channel Quantification

To quantify the leakage of an implementation, we model the secret as a random variable X with prior $\vec{\pi}$, the potential attacker observations as a random variable Obs, and the implementation as a deterministic discrete memoryless channel $C : X \to Obs$. Let $p(x_i)$ and $p(obs_j)$ be the probabilities for $x_i \in X$ and $obs_j \in Obs$. Let $p(obs_j|x_i)$ be the conditional probability for obs_j given x_i. Min-entropy $H_\infty(X) = -\log_2 \max_i\ p(x_i)$ and conditional min-entropy $H_\infty(X|Obs) = -\log_2 \Sigma_{j=1}^{|Obs|} p(obs_j) \cdot \max_i \frac{p(obs_j|x_i)\cdot p(x_i)}{p(obs_j)}$ [70] capture the uncertainty of a one-try attacker about the secret before, resp., after his observation. The leakage $L(\vec{\pi}, C) = H_\infty(X) - H_\infty(X|Obs)$ is maximized for a uniform $\vec{\pi}$ [41] and then amounts to $L(\vec{\pi}, C) = \log_2 |Obs|$ [70].

We capture Obs for a given implementation using an execution model of the target platform. We call a model that captures the exact values on which the implementation computes (e.g., values of registers) a concrete execution model. It consists of a concrete domain $\mathcal{D}$, which is the set of all possible execution states, and a concrete semantics $\text{upd}_{\mathcal{D}} : \mathcal{D} \times \mathcal{I} \to \mathcal{D}$, which models how executing an instruction from the set $\mathcal{I}$ changes the execution state. Since computing Obs on such a concrete execution model is usually not computationally feasible, we use abstract interpretation [15] to make the computation tractable. We represent concrete execution states by abstract execution states from an abstract domain $\overline{\mathcal{D}}$ and overapproximate the concrete semantics by an abstract semantics $\text{upd}_{\overline{\mathcal{D}}} : \overline{\mathcal{D}} \times \mathcal{I} \to \overline{\mathcal{D}}$. We perform a reachability analysis on $\text{upd}_{\overline{\mathcal{D}}}$ to obtain the set $Obs^{\overline{\mathcal{D}}}$ of observations an attacker might make in any concrete execution modeled by the possible abstract executions. This set overapproximates the actual set Obs,

[3] available at `https://www.mais.informatik.tu-darmstadt.de/qkd-esorics21.html`.

i.e., $\log_2 |Obs^{\overline{\mathcal{D}}}| \geq \log_2 |Obs| \geq L(\vec{\pi}, C)$. Overall, we compute upper bounds on the cache-side-channel leakage of an implementation as $\log_2 |Obs^{\overline{\mathcal{D}}}|$.

This combination of abstract interpretation and information theory was pioneered in [40] and then broadened (e.g., [8,19,20,51]), but the particular analysis that we present is a novel contribution of this article.

2.2 Quantum Key Distribution

Quantum Key Distribution (QKD) [56] aims at establishing a shared secret key between two parties, Alice and Bob, in the presence of an attacker, Eve, who has access to a quantum computer. A QKD implementation consists of a physical part and a software part. Overall, there are five stages in a QKD implementation: one stage in the physical part and four stages in the software part.

In the first stage, the *raw-key exchange*, Alice generates a random bitstring, her so-called raw key $b_{\mathrm{r}}^{\mathrm{A}}$. She transmits $b_{\mathrm{r}}^{\mathrm{A}}$ to Bob using qbits (quantum bits). BB84 [6] implements qbits using the polarization of photons. A polarization is a linear combination of two orthogonal base vectors. In BB84, two bases are used: $\oplus$ ($\uparrow$ and $\rightarrow$) and $\otimes$ ($\nearrow$ and $\nwarrow$). For each bit in $b_{\mathrm{r}}^{\mathrm{A}}$, Alice randomly chooses base $\oplus$ or $\otimes$ and polarizes a photon along the base vectors to encode the bit.

Alice sends the photons to Bob, who measures their polarization and obtains his raw key, $b_{\mathrm{r}}^{\mathrm{B}}$. Bob picks the bases for his measurements independently. This is introducing errors, but crucial for security. If Eve intercepts photons without knowing the bases, she will use the wrong base with a 50% chance. Based on the laws of quantum physics, a wrong measurement will disturb the actual polarization. When resending photons to Bob, Eve can only guess the original polarization of the photons that she measured wrongly. Overall, she will, hence, introduce errors in 25% of the intercepted photons, which can be detected later.

Overall, $b_{\mathrm{r}}^{\mathrm{A}}$ and $b_{\mathrm{r}}^{\mathrm{B}}$ might differ due to (i) mismatching bases, (ii) attacks, or (iii) measurement errors. We refer to the error rate from (ii) and (iii) by err_{true}.

Starting from the second stage, the remaining QKD stages happen in software. The second stage of QKD is the *key sifting*. Bob sends his polarization bases to Alice, who checks which of these bases are wrong. Alice and Bob discard all bits from $b_{\mathrm{r}}^{\mathrm{A}}$ and $b_{\mathrm{r}}^{\mathrm{B}}$ for which Bob used a wrong base, and they obtain the shorter bitstrings $b_{\mathrm{si}}^{\mathrm{A}}$ and $b_{\mathrm{si}}^{\mathrm{B}}$. Note that, Bob sends the polarization bases via a conventional electromagnetic channel. Eve might intercept and modify messages on this channel but cannot impersonate other parties and cannot use information she obtains to hide an attack, because the raw-key exchange is over.

The third stage of QKD, *parameter estimation*, has two purposes: (1) to detect attacks and (2) to determine an estimated error rate err_{est} to be used in the fourth step of QKD. Alice and Bob compare randomly selected bits of $b_{\mathrm{si}}^{\mathrm{A}}$ and $b_{\mathrm{si}}^{\mathrm{B}}$ to obtain err_{est}. All bits used to compute err_{est} are discarded, resulting in the shorter bitstrings $b_{\mathrm{s}}^{\mathrm{A}}$ and $b_{\mathrm{s}}^{\mathrm{B}}$, called sifted keys. Alice and Bob restart the QKD if err_{est} exceeds a predefined threshold, which indicates an attack.

The fourth stage, called *error correction* eliminates any remaining differences between $b_{\mathrm{s}}^{\mathrm{A}}$ and $b_{\mathrm{s}}^{\mathrm{B}}$ so that Alice and Bob obtain the bitstrings $x_{\mathrm{ec}}^{\mathrm{A}}$ and $x_{\mathrm{ec}}^{\mathrm{B}}$, respec-

tively, the so-called error-corrected keys. There are multiple coding techniques that can be used for error correction, including Cascade [9] and LDPC [28]. We focus on LDPC, which optimizes the required amount of communication [18]. Alice and Bob first agree on a length $n = k + m$ (for k message bits and m parity bits) and on a public $k \times n$ parity matrix H. The error correction then consists of an *encoding* by Alice and a *decoding* by Bob. Alice splits b_s^A into blocks of length k. For each block, she computes m parity bits using H and sends them to Bob. For herself, Alice sets $x_{ec}^A = b_s^A$. Bob computes the most likely guess for each sifted-key block based on b_s^B and the parity bits. He concatenates the resulting blocks to obtain x_{ec}^B. If m is sufficiently large based on err_{est} (see [61]), then $x_{ec}^B = x_{ec}^A$.

The final step, *privacy amplification*, compresses x_{ec}^B and x_{ec}^A using hash functions to compensate for leakage to Eve. One example of suitable hash functions are Toeplitz matrices (matrices in which each descending diagonal consists of equal values). Alice and Bob agree on a length l and on a public $l \times k$ Toeplitz matrix T. They multiply T with each k-bit block of the error-corrected key and concatenate the results to obtain the final key $x_{pa}^A = x_{pa}^B$. Naturally, the more the key is compressed in the end, the more qbits need to be transmitted during the first stage of the QKD in order to achieve the same key length.

3 Analysis for Cache-Side-Channel Quantification

In the error-correction step of QKD, Bob computes the most likely values of the sifted-key bits based on his measurements and on additional information received from Alice. That is, software implementations for QKD need to handle probabilities, which are most naturally represented using floating-point values.

To enable cache-side-channel quantification for x86 binaries with floating-point instructions, we define an execution model that captures both, the regular x86 architecture and the components specific to floating-point instructions. Based on this model, we define an abstract reachability analysis that handles floating-point values reliably. We provide tool support to automate the analysis.

Attacker Model. We consider an attacker who observes a victim's interaction with a cache. More concretely, we consider an attacker who has access to a trace of the cache hits and misses encountered, e.g., based on CPU performance counters. Such trace-based attacks have been mounted, e.g., on OpenSSL AES [1], and on reference implementations of the CAMELLIA and CLEFIA ciphers [62,64].

We capture such an attacker by the attacker model *csc-att*. An attacker under *csc-att* observes a trace of the victim binary's execution that we call *cache trace*. This trace contains the entry "Hit" for each cache hit, the entry "Miss" for each cache miss, and the entry "None" for each instruction without memory access.

3.1 Execution Model

In our execution model, we model a 32-bit architecture, which is supported by older lab machines, as well as newer machines in compatibility mode. That is, we focus on the execution of 32-bit x86 instructions in combination with x87 floating-point instructions. While regular x86 instructions are executed by the CPU, x87 instructions are executed by the floating-point unit (FPU).

CPU (x86). The CPU operates on 32-bit memory entries and CPU registers. It maintains six 1-bit status flags that store information on the results of comparisons and arithmetic operations [37]: Carry Flag (CF), Parity Flag (PF), Auxiliary Carry Flag (AF), Zero Flag (ZF), Sign Flag (SF), and Overflow Flag (OF).

In our model, $\mathcal{X}_{32}$ is the set of all CPU registers and memory locations. By $\mathcal{F}_{CPU} = \{CF, PF, AF, ZF, SF, OF\}$ we model the set of all CPU flags and by $\mathcal{V}_{32}$ we model the set of all 32-bit values. We capture a CPU state by two functions of types $\mathcal{F}_{CPU} \to \mathbb{B}$ and $\mathcal{X}_{32} \to \mathcal{V}_{32}$ that map each CPU flag to a boolean value and each CPU register and memory entry to a 32-bit value, respectively.

FPU (x87). The FPU operates on the same memory entries as the CPU. In addition, it operates on a stack that consists of 8 dedicated 80-bit FPU registers (consisting of 1 bit sign, 15 bits exponent and 64 bit mantissa) [37]. For each stack entry, the FPU stores a tag: "valid" for a valid number, "zero" for entry 0, "special" for entries with a special value (e.g., not-a-number) or "empty" if the entry is empty. Like the CPU, the FPU maintains status flags, the so-called condition-code flags, to realize conditional control flow: C0, C1, C2, C3.

We model the set of all FPU registers by $\mathcal{S}_{64}$ and the set of all possible floating-point values by $\mathcal{V}_{64}$. Let $\mathcal{F}_{FPU} = \{C0, C1, C2, C3\}$ be the set of all FPU condition-code flags and $\mathcal{T} = \{\text{valid}, \text{empty}\}$ be the set of FPU-stack tags, where *empty* models the tag "empty" and *valid* models all other tags. We capture an FPU state by two functions of types $\mathcal{F}_{FPU} \to \mathbb{B}$ and $\mathcal{S}_{64} \to \mathcal{T} \times \mathcal{V}_{64}$, which map FPU flags to booleans and FPU registers to tagged floating-point values.

Cache. Frequently-used memory entries are stored in caches for quick access. Most contemporary architectures incorporate a multi-level hierarchy of caches.

We model the memory hierarchy by distinguishing between the main memory and one level of cache. We focus on a 32KB 8-way set-associative data cache with line size 64B (like the L1 cache of the Skylake architecture [37]) and LRU replacement. By C_{pos}, we model the set of all possible positions in the cache, including the position "uncached". We capture a cache state by a function of type $\mathcal{X}_{32} \to C_{pos}$, which maps each memory entry to its position in the cache.

Combined State and Executions. We model the possible execution states by

$$\mathcal{D} = (\mathcal{F}_{CPU} \to \mathbb{B}) \times (\mathcal{F}_{FPU} \to \mathbb{B}) \times (\mathcal{X}_{32} \to \mathcal{V}_{32}) \times (\mathcal{S}_{64} \to \mathcal{T} \times \mathcal{V}_{64}) \times (\mathcal{X}_{32} \to C_{pos}) .$$

That is, a state in the concrete domain $\mathcal{D}$ is a combination of a CPU state, FPU state, and cache state. Let $\mathcal{I}$ be the set of x86 and x87 instructions. We model their concrete semantics by a function $\text{upd}_{\mathcal{D}} : \mathcal{D} \times \mathcal{I} \to \mathcal{D}$.

Note that, overall we made two key simplifications in our model of execution: We focus on 32-bit binaries and on a single level of cache. Both simplifications are sensible ones and also common in existing analyses [8,19,20,51].

3.2 Abstract Reachability Analysis

We define the abstract domain for our abstract reachability analysis by

$$\overline{\mathcal{D}} = ((\mathcal{F}_{CPU} \to \mathbb{B}) \times (\mathcal{F}_{FPU} \to \mathbb{B})) \to ((\mathcal{X}_{32} \to 2^{\mathcal{V}_{32}}) \times (\mathcal{S}_{64} \to 2^{\mathcal{T}} \times 2^{\mathcal{V}_{64}}) \times (\mathcal{X}_{32} \to 2^{C_{pos}})).$$

The values of CPU flags and FPU flags are captured by functions of types $\mathcal{F}_{CPU} \to \mathbb{B}$ and $\mathcal{F}_{FPU} \to \mathbb{B}$ like in the concrete domain. The values of CPU registers and memory entries with concrete type $\mathcal{X}_{32} \to \mathcal{V}_{32}$ are abstracted from by a function of type $\mathcal{X}_{32} \to 2^{\mathcal{V}_{32}}$, i.e., by a set abstraction. Analogously, the tags and values of FPU registers with concrete type $\mathcal{S}_{64} \to \mathcal{T} \times \mathcal{V}_{64}$ are abstracted from by a function of type $\mathcal{S}_{64} \to 2^{\mathcal{T}} \times 2^{\mathcal{V}_{64}}$, and the cache positions with the concrete type $\mathcal{X}_{32} \to \mathcal{C}_{pos}$ are abstracted from by a function of type $\mathcal{X}_{32} \to 2^{\mathcal{C}_{pos}}$.

Unlike in the concrete domain, the state of the flags and the state of memory, registers and cache are not combined using the Cartesian product $\times$ in the abstract domain. Instead, the abstract domain is defined as a mapping $\to$ from the state of the flags to the state of memory, registers and cache. That is, the domain allows to distinguish between the states of memory, registers and cache that are possible across different states of the flags. Since the flags determine the control flow of binaries, this means that the abstract domain allows to distinguish between the execution states across different control-flow branches.

Abstract Semantics. We define the abstract semantics of x86 and x87 instructions by $\text{upd}_{\overline{\mathcal{D}}} : \overline{\mathcal{D}} \times \mathcal{I} \to \overline{\mathcal{D}}$, which overapproximates the effect of instructions on operands that are sets. In the definition, we faced two main challenges.

Firstly, the pointer for the FPU stack is stored in a register and, hence, has an abstract value of type $2^{\mathcal{V}_{32}}$. That is, instructions operate on a set of candidate stack pointers. To treat such situations sufficiently precisely, our abstract semantics takes into account the tag of the stack entries pointed to by the candidate pointers. For instance, on a stack entry tagged {valid}, a pop instruction will yield the tag {valid, empty} and an unmodified stack value, while a push instruction will yield the tag {valid} and a stack value that is the union of the previous value and the singleton set containing the loaded value.

Secondly, values in memory are Byte-aligned but not necessarily aligned to the borders of memory blocks (chunks of memory that are cached together in one cache line) [37]. Large values like floating-point values might cross the border between two memory blocks. Instructions that retrieve such values from the memory might affect multiple cache lines. To reliably overapproximate the effect of such accesses, our abstract semantics splits such values into individual Bytes and updates all cache lines in which at least one of the Bytes is cached.

Computation of Leakage Bounds. Based on an abstract initial configuration $\overline{d}$, we compute all possible abstract executions using $\text{upd}_{\overline{\mathcal{D}}}$. These executions overapproximate the possible concrete executions and corresponding attacker observations. Based on the abstract executions, we determine the set $Obs^{\overline{\mathcal{D}}}$ of attacker observations that an attacker under *csc-att* might make in any concrete execution modeled by the possible abstract executions and compute $\log_2 |Obs^{\overline{\mathcal{D}}}|$ as the leakage bound (see Sec. 2.1) with respect to an attacker under *csc-att*.

3.3 Automation through Tool Support

To create tool support for our analysis, we implemented the model and semantics for the FPU from scratch, because no such implementation for the FPU was

available. Our implementation covers a large set of x87 instructions, including load, store, arithmetic and compare instructions. For the models of the CPU and cache, as well as the semantics of CPU instructions, we aimed at maximum code reuse from existing tools. While no previously existing tool provides support for all required CPU features and instructions, CacheAudit [20] provides a model of the cache and a model of the CPU for selected status flags and instructions. We reused the code from CacheAudit and augmented it with support for 34 additional CPU instructions and the parity flag, which occurs in jumps that are conditional on whether a floating-point value is NaN (not-a-number).

When building on CacheAudit, we noticed that CacheAudit's abstract semantics cannot handle memory entries that map to multiple cache lines. We implemented a solution based on our semantics, splitting values into Bytes.

Overall, the resulting analysis tool takes an x86/x87 binary as input and returns the upper bound $\log_2 |Obs^{\overline{\mathcal{D}}}|$ on the leakage to attackers under *csc-att*.

4 Practical Evaluation

For the evaluation of our program analysis, we consider an implementation of the BB84 protocol. More concretely, we consider the implementation from [42,57], which is both, in practical use and publicly available.

The implementation covers all stages of QKD that happen in software: key sifting, parameter estimation, error correction and privacy amplification. For our evaluation, we selected the two last stages: error correction and privacy amplification, because they involve non-trivial computations on secret bitstrings. More concretely, we selected functions for the encoding part of error correction, for the decoding part of error correction, and for privacy amplification from [42].

Before we apply our program analysis, we perform well-defined simplifications to the functions. We then lift the analysis results to the original functions in a separate step. That is, our results are independent of the simplifications.

For the explanation of our simplified functions, we focus on their high-level C code. For the evaluation of our program analysis, we considered binaries obtained with gcc 7.4.0 with the flags `-m32 -fno-stack-protector`.

Encoding. We created the simplified version of the function `dense_encode` shown in Figure 1. The parameter `sblk` with size $k = 4$ is the sifted-key block to be encoded. It is copied into the vector `u` (Line 9) and passed to function `mod2dense_multiply` (Line 10), which multiplies the sifted-key block with a generator matrix `G` that is derived from the parity matrix H. The resulting code block (the sifted-key block and $m = 3$ parity bits) is stored in `cblk`.

Figure 1 simplifies the original code in two aspects: (1) it stores matrices and vectors in fixed-size arrays instead of dynamically allocated pointer structures and (2) it stores the variables `G` and `cols` locally instead of globally. We tested with a Hamming(7,4) code that the simplifications do not alter the functionality.

Decoding. We created simplified versions of the functions `initprp` and `iterprp` (Figure 2). They perform LDPC decoding with a parity matrix stored

```
1  #define M 3  #define K 4  #define N M+K [...]
2  void mod2dense_multiply(char m1[M][K], char m2[K][1], char r[M][1]){
3   for (int i=0;i<M;i++){ r[i][0]=0; }
4   for(int k=0;k<K;k++){ if (m2[k][0]==1){
5     for(int m=0;m<M;m++){ r[m][0]^=m1[m][k]; }}}}
6  void dense_encode(char sblk[K],char cblk[N],char u[K][1],char v[M][1]){
7   int cols[N]={4,5,6,0,1,2,3}; char G[M][K]; int j;
8   for(j=M;j<N;j++){ cblk[cols[j]] = sblk[j-M]; }
9   for(j=M;j<N;j++){ mod2dense_set(u,j-M,0,sblk[j-M]);}
10  mod2dense_multiply(G,u,v);
11  for(j=0;j<M;j++){cblk[cols[j]]=mod2dense_get(v,j,0);}}
12 void main(){ char sblk[K]; char cblk[N]; char u[N][1]; char v[M][1];
13  dense_encode(sblk,cblk,u,v);}
```

Fig. 1. Target Encoding Implementation

```
1  #include "math.h"  #define M 3  #define K 4  #define N M+K
2  void initprp (char H_val[M][N], double H_lr[M][N], double H_pr[M][N], double
       lratio[N], char dblk[N], double bprb[N]){ int e; int j;
3   for (j = 0; j<N; j++){ for (e = 0; e < M;  e++){
4     if (H_val[e][j] == 1){ H_pr[e][j] = lratio[j]; H_lr[e][j] = 1.0;}}
5    if (bprb) bprb[j] = 1 - 1/(1+lratio[j]); dblk[j] = lratio[j]>=1;}}
6  void iterprp(char H_val[M][N], double H_lr[M][N], double H_pr[M][N], double
       lratio[N], char dblk[N], double bprb[N]){
7   double pr, dl, t; int e, i, j; double temp; double temp2; double dummy;
8   for (i = 0; i<M; i++){ dl = 1; for (e = 0; e < N; e++){
9     if (H_val[i][e] == 1){ H_lr[i][e] = dl; dl *= 2/(1+H_pr[i][e])-1;}}
10   dl = 1; for (e = N-1; e >= 0; e--){
11    if (H_val[i][e] == 1){ t = H_lr[i][e]*dl;
12      H_lr[i][e] = (1-t)/(1+t); dl *= 2/(1+H_pr[i][e]) - 1;}}}
13  for (j = 0; j<N; j++){ pr = lratio[j]; for (e = 0; e < M; e++){
14    if (H_val[e][j] == 1){ H_pr[e][j] = pr; pr *= H_lr[e][j];}}
15   if (isnan(pr)){ pr = 1;} else{dummy=1;__asm__("nop":::);}
16   if (bprb) bprb[j] = 1 - 1/(1+pr); dblk[j] = pr>=1; pr = 1;
17   for (e = M-1; e >= 0; e--){
18    if (H_val[e][j] == 1){ H_pr[e][j] *= pr; temp = H_pr[e][j];
19     if (isnan(temp)){temp = 1;} else{temp2=1;__asm__("nop":::);}
20     H_pr[e][j] = temp; pr *= H_lr[e][j];}}}}
21 void main(){char H_val[M][N]={{1,1,0,1,1,0,0},{1,0,1,1,0,1,0},{0,1,1,1,0,0,1}};
22  double H_lr[M][N]; double H_pr[M][N]; double lratio[N]; char dblk[N];
23  double bprb[N]; initprp(H_val, H_lr, H_pr, lratio, dblk, bprb);
24  iterprp(H_val, H_lr, H_pr, lratio, dblk, bprb);}
```

Fig. 2. Target Decoding Implementation

in an array `H_val`. They compute the correct values of the bits in the sifted-key block based on probabilities stored in the auxiliary arrays `H_lr` and `H_pr`. Function `initprp` initializes the probabilities using a vector `lratio` of likelihood ratios based on Bob's measurements. Function `iterprp` then iteratively updates the probabilities. The resulting error-corrected-key block is stored in `dblk`.

Our simplified implementation (1) uses fixed-size arrays instead of pointer structures, (2) performs only one iteration of `iterprp`, (3) uses a fixed instead of a variable parity matrix, and (4) performs overflow handling on dummy variables if no overflow occurred instead of skipping the step. Again, we tested the functionality of the simplified implementation on a Hamming(7,4) code.

Note that the variables `H_lr`, `H_pr` and `lratio` are of type **`double`**, i.e., floating-point values. Our simplification preserves these datatypes, because the corresponding values represent probabilities and likelihood ratios. Representing them as integers would loose precision and cause a great deviation from the original code due to the additional encoding needed to represent probabili-

ties as integers. Therefore, the binary of our simplified implementation contains floating-point instructions that need to be handled by our program analysis.

Privacy Amplification. We created the simplified version of the function `calcPA` shown in Figure 3. It takes an error-corrected key block `key` and the target length `paLen` for the privacy-amplified key block as inputs. It hashes the error-corrected key block by multiplying it with the Toeplitz matrix in Line 8.

The simplified `calcPA` differs from the original `calcPA` by (1) using fixed-size instead of variable-size arrays, (2) using local variables and parameters instead of class members and (3) initializing each entry of the Toeplitz matrix to the least-significant bit (lsb) of the value stored at the uninitialized address 0x4 (to make explicit that the Toeplitz matrix is binary). Again, we tested the functionality of the simplified implementation on a Hamming(7,4) code.

5 Vulnerability in the QKD Implementation

With our analysis tool, we detected a vulnerability in the QKD implementation from [42], which might leak the entire secret key. In this section, we describe the detection, assessment, and mitigation of this vulnerability.

Note that, the vulnerability is located in the encoding part of the error-correction step. We also analyzed the decoding part of the error-correction step and the privacy-amplification step, but we did not detect vulnerabilities in these implementations. We will describe the security analysis for the hardened QKD software, including decoding and privacy amplification, in Section 6.

Detection and Assessment. For `dense_encode` from Figure 1, our analysis computes the leakage bound 4 bit with respect to *csc-att*. Recall that the sifted-key-block here has 4 bit, i.e., leaking 4 bit might reveal the entire sifted key.

Recall from Section 4 that the differences between the original and the simplified implementation of `dense_encode` are only the datatypes and scope of the variables that store matrices and vectors. These differences do not impact the control flow and the locations of memory accesses in the encoding implementation. Thus, they do not impact the leakage with respect to *csc-att*. That is, our leakage bound indicates a potential leakage also in the original encoding implementation from [42]. We investigate this potential leakage in the following.

Figure 4 shows excerpts of two functions from the original encoding implementation. The function `dense_encode` encodes the sifted-key block `sblk` using the generator matrix `G`. To this end, the sifted-key block is passed to the function

```
1  #include "string.h"  #include "stdbool.h"
2  #define KEYLENGTH 4  #define PAKEYLENGTH 2
3  void calcPAKey(bool* key, int paLen){
4   int toepMatLen=KEYLENGTH+paLen-1; char paKey[paLen]; char toepMat[toepMatLen];
5   for(int i=0;i<toepMatLen;++i){ toepMat[i]= *((int*)0x4)&1;}
6   for(int i=0;i<paLen;i++){ paKey[i]=0;
7    for(int j=0;j<KEYLENGTH;j++){ int id=i-j+KEYLENGTH-1;
8     paKey[i]+=toepMat[id]*key[j]; paKey[i]=paKey[i]%2;}}}
9  void main(){bool key[KEYLENGTH];calcPAKey(key,PAKEYLENGTH);}
```

Fig. 3. Target Privacy-Amplification Implementation

`mod2dense_multiply` in Line 10 via vector `u`. The function `mod2dense_multiply` multiplies the sifted-key block (`m2`) by the generator matrix (`m1`).

For each column of the generator matrix and result matrix (Line 2), the function `mod2dense_multiply` iterates over the sifted-key block (Line 3). If a bit in the sifted-key block is set (Line 4), the function adds the corresponding row of the generator matrix `m1` to the intermediate result `r` of the multiplication in Line 5 and Line 6. That is, the function `mod2dense_multiply` only accesses those rows of `m1` that correspond to set bits of the sifted-key block.

Consider an attacker under *csc-att*. He observes the cache trace while Alice executes the encoding. He recognizes which parts of the trace correspond to iterations of the loop in Line 3 in which the branch in Line 4 is taken because in these iterations cache accesses occur between the non-memory-access steps corresponding to the loop guard. As the branch in Line 4 is taken exactly for each set bit in the sifted-key block `m2`, the attacker learns which bits in the sifted-key block are 1 and which are 0. That is, he learns the entire sifted-key block.

The attacker can obtain all blocks of Alice's sifted key by observing all executions of the encoding function. By concatenating the blocks, he obtains Alice's entire sifted key. Recall that Alice's sifted key is identical to the error-corrected key (Section 2.2). That is, once the attacker knows Alice's sifted key, he can emulate the privacy-amplification step by applying the public hash function to the error-corrected key. As a result, he obtains the privacy-amplified key and can use it to decrypt the subsequent communication between Alice and Bob.

This vulnerability is not only a concern for the QKD software from [42]. It is caused by the optimization to skip multiplications with zero, which is a very natural optimization in a matrix multiplication. Furthermore, the encoding implementation in [42] reuses the original implementation of LDPC encoding by Radford Neal [55], who rediscovered LDPC codes together with MacKay in the 1990s [46]. The vulnerability we detected is also contained in Neal's implementation [55], which has been forked by many others [32]. Overall, a solution for hardening implementations against the vulnerability is highly desirable.

Hardening of the QKD Software. We provide a hardened implementation of `mod2dense_multiply` in Figure 5. The implementation iterates through all rows of the generator matrix, independently of the value of the respective sifted-key bit. That is, control flow and memory accesses are independent of the sifted key.

To preserve the functionality of the multiplication, the term to be added to the intermediate result is masked by the value of the sifted-key bit in Line 3. This

```
1  void mod2dense_multiply(mod2dense *m1,mod2dense *m2, mod2dense *r){ [...]
2   for(j=0;j<mod2dense_cols(r);j++){
3    for(i=0;i<mod2dense_rows(m2);i++){
4     if(mod2dense_get(m2,i,j)){
5      for(k=0;k<r->n_words;k++){
6       r->col[j][k]^=m1->col[i][k];}}}}}
7  void dense_encode(char *sblk,char *cblk, mod2dense *u,mod2dense *v){ int j;
8   for(j=M;j<N;j++){cblk[cols[j]]=sblk[j-M];}
9   for(j=M;j<N;j++){ mod2dense_set(u,j-M,0,sblk[j-M]);}
10  mod2dense_multiply(G,u,v);
11  for(j=0;j<M;j++){ cblk[cols[j]]=mod2dense_get(v,j,0);}}
```

Fig. 4. Excerpt from [42]: Encoding

technique is inspired by conditional assignment [54]. We tested the functionality of the hardened implementation with a Hamming(7,4) code.

As the modified function `mod2dense_multiply` preserves the functionality of the multiplication function from Figure 1, it can be plugged into the simplified encoding implementation from Figure 1. To apply the hardening to [42] and [55], the data types of `m1`, `m2` and `r` in the function `mod2dense_multiply` need to be changed back to pointer structures before integrating the function.

6 Security of the Hardened Implementation

By applying our program analysis to the hardened QKD software, consisting of the mitigated encoding implementation, the decoding implementation, and the privacy-amplification implementation, we obtain security guarantees with respect to attackers under *csc-att*. In the following, we describe the security guarantees for each of the analyzed QKD steps.

Hardened Encoding. For the hardened encoding implementation from Section 5, our analysis computes the leakage bound 0 bit. That is, the implementation does not leak any information to attackers under *csc-att*. If the hardening is deployed in the encoding implementation from [42] as described in Section 5, the resulting implementation will also be secure with respect to *csc-att*, because the data type and scope of the variables do not influence the possible traces.

Decoding. For the implementation from Figure 2, our analysis also returns the leakage bound 0 bit, i.e., there is no leakage to attackers under *csc-att*.

The implementation simplifies [42] in four aspects (see Sec. 4). We lift the leakage bound to the original implementation by discussing each simplification:

1. While [42] iterates through a pointer structure storing only the set bits of the parity matrix, the simplified version iterates over the complete matrix. This eliminates a dependence of the cache trace on the parity matrix. Since the parity matrix is public, this simplification does not influence the leakage.
2. The parity matrix is fixed to a random matrix in the simplified implementation. This eliminates another dependence of the cache trace on the parity matrix. Again, the leakage is not affected because the parity matrix is public.
3. The simplified implementation calls `iterprp`, which would usually be iterated, only once. Since the leakage bound for decoding is 0 bit, no leakage might accumulate, i.e., the simplification does not influence the leakage.
4. Overflows of probabilities are handled in [42] by resetting the probabilities to 50%. The cache trace reveals the number of overflows encountered. This

```
1  void mod2dense_multiply(char m1[M][K], char m2[K][1], char r[M][1]){
2   for (int m=0; m<M; m++){r[m][0] = '0';
3    for (int k=0; k<K; k++){r[m][0]^= m1[m][k] & m2[k][0];}
4    r[m][0] = r[m][0] % 2;}}
```

Fig. 5. Hardening of the Encoding Function

is hidden by the simplified version. This simplification does not influence the leakage since overflowing probabilities are reset to 50%. Nothing about the final probabilities is revealed by the occurrence of an overflow.

Overall, the zero leakage bounds can be translated to the original decoding implementation, which is, hence, also secure against attackers under *csc-att*.

Privacy Amplification. For Figure 3, our analysis yields the leakage bound 0 bit. That is, the implementation does not leak to attackers under *csc-att*.

Recall from Section 4 that the only differences between Figure 3 and the original implementation are the fixed size of variables, the scope of variables and the explicit marking of the Toeplitz matrix as uninitialized binary matrix. None of these has an influence on the control flow or memory accesses during the privacy amplification. That is, the zero leakage bound also applies to the original implementation. No need for additional hardening arises.

Together with the bounds for the encoding and decoding implementations, we now have a set of security guarantees for the hardened QKD software.

7 Combining Rewriting and Privacy Amplification

While our hardening by program rewriting in Section 5 is effective, it also removes an optimization and thereby increases the running time of the encoding step (by about 50% on average for a uniformly distributed sifted key). In this section, we present a more flexible mitigation strategy, which allows to optimize performance cost across the physical and software-based part of QKD.

Parametric Mitigation of the Vulnerability. Figure 6 shows a parametric program rewriting of the encoding implementation. Instead of iterating through all of the block `m2` as in the hardening from Figure 5, the function iterates through the first `SEC_PARAM` bits of `m2` and only iterates through the remaining bits if they are set. That is, the optimization is disabled selectively. While Figure 6 still leaks a consecutive key portion, additionally randomizing the indices where the optimization is disabled would even hide where the leaked bits belong.

We use our program analysis to verify whether the security guarantees are, indeed, improved incrementally by incrementally disabling the optimization. We obtain the following leakage bounds: 3 bit for SEC_PARAM= 1, 2 bit for SEC_PARAM= 2, and 1 bit for SEC_PARAM= 3. That is, our mitigation allows us to trade performance against security locally within the encoding function.

Integration with Privacy Amplification. Traditional security analyses of QKD solutions [65] aim at guarantees for the secret key resulting after pri-

```
5  void mod2dense_multiply(char m1[M][K], char m2[K][1], char r[M][1]){
6   for (int i=0; i<M; i++){r[i][0] = 0;}
7   for (int k=0; k<K; k++){
8    if (k < SEC_PARAM || m2[k][0] == 1){
9     for (int m = 0; m < M; m++){r[m][0] ^= m1[m][k] & m2[k][0];}}}}
```

Fig. 6. Parametric Mitigation in the Encoding Function

vacy amplification. They take into account leakage to Eve during the raw-key exchange and through the electromagnetic channel.

Privacy amplification requires the transmission of more particles during the raw-key exchange, which lead to more remaining key material after error correction. This key material of length l_{block} is then compressed with a hash function to a key of target length l_{target}. To compensate for potential leaks across the different stages of QKD, privacy amplification relies on upper leakage bounds b_{raw} for the raw-key exchange and b_{parity} for the electromagnetic channel. The length l_{target} of the privacy-amplified key should be $l_{target} < l_{block} - b_{raw} - b_{parity}$ bit. The smaller l_{target} is compared to $l_{block} - b_{raw} - b_{parity}$, the more secure is the final key.

We integrate our leakage bounds as follows. Let b_{cache} be the leakage bound derived with our analysis. To compensate the potential cache-side-channel leakage in addition to the leakage from the raw-key exchange and the electromagnetic channel, the target length should be $l_{target} < l_{block} - b_{raw} - b_{parity} - b_{cache}$ bit. The reason is that the information leaked during a traditional attack on QKD and the information leaked through the side channel might be distinct.

Consider the parametric cache-side-channel mitigation from Figure 6 for `SEC_PARAM` = 3. In this case, only one bit of cache-side-channel leakage remains to be mitigated, i.e., $l_{target} < 4 - b_{raw} - b_{parity} - 1 = 3 - b_{raw} - b_{parity}$ bit. That is, only a slightly stronger privacy amplification than usual will be required.

But how should one choose how much leakage to mitigate by program rewriting and how much to mitigate by privacy amplification?

Optimizing the Combined Mitigation. The combination of program rewriting and privacy amplification allows one to split the mitigation overhead across the physical and software parts of QKD. In the following, we present a cost model that allows one to choose an optimal split based on a given QKD setup.

Let c_{raw} be the cost for the raw-key exchange, sifting and parameter estimation for one sifted-key block. Let c_{ec} and c_{pa} be the costs of the encoding and privacy amplification, respectively, for one sifted-key block without any software-based cache-side-channel mitigation. Given a bound b_{cache} on cache-side-channel leakage, the cost for obtaining a key of the length l_{target} is $\frac{(c_{raw}+c_{ec}+c_{pa})\cdot l_{target}}{l_{block}-b_{raw}-b_{parity}-b_{cache}}$

That is, the cost for the same level of security in the privacy-amplified key increases linearly with b_{cache}. If b_{cache} is too high compared to l_{block}, the key exchange becomes impossible, as the cost would become infinite.

With our mitigation that decreases b_{cache} to b'_{cache}, the cost becomes

$$\frac{(c_{raw} + c_{ec} \cdot \frac{l_{block}-0.5b'_{cache}}{0.5\cdot l_{block}} + c_{pa}) \cdot l_{target}}{l_{block} - b_{raw} - b_{parity} - b'_{cache}}$$

The encoding becomes more expensive because instead of performing actions for 50% of the bits in the block (on average given a uniform distribution of the sifted key), it needs to perform actions for 50% of the b'_{cache} bits which are not protected and for all of the $l_{block} - b'_{cache}$ bits, which are protected.

If we know the cost of the individual phases, we can find the leakage bound b'_{cache} which minimizes the cost. Given the desired b'_{cache}, we can then set the mitigation strength `SEC_PARAM` to $l_{block} - b'_{cache}$ in the code.

We evaluate this technique for a set of realistic parameters for cost and leakage bounds. Consider a QKD setup that implements LDPC following IEEE standard 802.11n-2009 [35] like, e.g., the implementation in [73]. Let the block length be $l_{block} = 1458$ bit with corresponding parity-bit length $b_{parity} = 486$ bit.

The leakage per bit during the raw-key exchange is $-err_{\text{true}} \cdot \log_2(err_{\text{true}}) - (1 - err_{\text{true}}) \cdot \log_2(1 - err_{\text{true}})$ [22]. Assuming an error rate $err_{\text{true}} = 5\%$, we arrive at a leakage of about 0.2864, i.e., $b_{raw} \approx 0.2864 \cdot l_{block} \approx 418$ bit.

The bit-rates that can be achieved for the raw-key exchange vary across different experimental setups between 10^{-3} and 13 Mbit/s [76]. Consider a setup with a bit-rate of $10Mbit$ of sifted key per second, i.e., where the transmission of one block takes $c_{raw} = 145.8 \cdot 10^{-6}$ seconds. Let the cost of encoding and privacy amplification be $c_{ec} = 0.001$ and $c_{pa} = 0.01$ seconds per sifted-key block. The latter two numbers are rounded based on performance measurements we performed on the code from Figure 1 and 3 across random inputs on a Lenovo ThinkPad X250 M93p with an i7-5600U CPU at 2.60GHz.

If we now aim to minimize the cost for the exchange of, e.g., $l_{target} = 5,000$ bit of symmetric key, we arrive at $(60.729 - 0.003429355 \cdot b'_{cache})/(554 - b'_{cache})$.

Hence, in this scenario, $b'_{cache} = 0$bit minimizes the overhead. That is, the complete hardening from Section 5 would be the most beneficial here, because the cost of privacy amplification outweighs the software overhead.

For different QKD setups or implementations, the result might differ. In particular, as research in physics is pushing the limits of photon detectors to achieve higher bit-rates [17], the effect of software performance might become more relevant. We hope that our program analysis and flexible mitigation technique will also be beneficial for future QKD setups and implementations.

8 Related Work

We are not aware of any prior work in the intersection of cache side channels and the security of QKD software. The closest are works on cache-side-channel assessment in general, on cache-side-channel attacks on other software, on the security of the quantum-channel transmission in QKD, and on the combination of QIF with concepts from quantum theory. We discuss these areas briefly below.

Assessment of Cache-Side-Channel Security. Side channels are traditionally detected manually as in [24,43]. Recently, complementary approaches have become popular, ranging from systematic testing [75] to empirical methods [14] like distinguishing experiments [49,50] to program analysis [16,20,48]. In the following, we describe the most related tools for cache-side-channel analysis.

The tool CacheAudit [20] quantifies the cache-side-channel leakage of x86 binaries through upper bounds on Shannon-entropy leakage and min-entropy leakage. To this end, CacheAudit performs an abstract reachability analysis for x86. Different versions of CacheAudit have been used in multiple side-channel analyses, e.g., of AES implementations [51], lattice-based cryptography [8] and modular exponentiation [19]. While CacheAudit provides reliable upper bounds

on cache-side-channel leakage, it covers only parts of the x86 instruction set architecture. None of the existing versions supports floating-point instructions.

CaSym [10] uses symbolic execution and SMT solving to explore the possible program executions and deduce the existence of a leaking pair of executions. CaSym does not provide upper bounds on cache-side-channel leakage. It only provides security guarantees for software that is free from cache side channels.

The tools LeakiEst [12] and LeakWatch [13] quantify side-channel leakage using statistical methods based on samples of side-channel observations. Both tools provide an estimation of leakage within a confidence interval. That is, they do not provide sound upper bounds on the leakage.

The tools DATA [75] and CacheD [74] detect potential cache-side-channel leakage in software based on execution traces. DATA is based on statistical methods and CacheD is based on symbolic execution and constraint solving. Both, DATA and CacheD are intended to support developers in debugging against cache side channels. They do not provide any security guarantees.

Cache-Side-Channel Attacks. Cache-side-channel attacks were first discussed by Page [60] in 2002. Since then, multiple variants have been developed, including access-based attacks like Prime+Probe, Evict+Time [59] or Flush+Reload [77], trace-based attacks [1] and time-based attacks [7]. Cache-side-channel attacks have also been mounted in the cloud [66] and on mobile devices [43,71]. Recently, cache-side-channel attacks became an even more serious concern based on their role in the Spectre [39] and Meltdown [44] attacks.

Security of Quantum-Channel Transmissions. The key idea of QKD is to create a setup where eavesdropping can be detected. To this end, information is encoded into photons [6,21], e.g., using equipment like beam splitters and photodiodes as in [23]. Atoms have been considerd as an alternative to photons, e.g., in [26]. An eavesdropper will likely destroy the correlation (measured, e.g., using quantum discord, classical correlation or quantum entanglement [2,58]) between the particles on the sender and receiver side. Thus, there will be a higher error rate in the transmission, by which the eavesdropper can be detected.

Existing research about attacks on QKD focuses primarily on attacking the quantum channel. Existing attacks include, e.g., the injection of bright light [45], the manipulation of the data received by Bob through adversarially constructed light pulses [47], tricking Alice's hardware into producing an erroneous encoding in the photons [27], or time-shifting the transmitted qbits [63].

Quantum Quantitative Information Theory. While we apply classical information theory in the context of quantum cryptography, Américo and Malacaria [4] apply concepts from the context of quantum systems to information theory. The applications presented in [4] are based on classical systems. They use a notion of quantum quantitative information flow (QQIF) to model the choice between different attacks and to model probabilistic program behavior. But the aim of QQIF is to also quantify the security of quantum systems. In the future, a combination of QQIF and our analysis might be an interesting direction for generalizing the integration between security guarantees in quantum cryptography.

9 Conclusion

We presented a solution for quantitative security of QKD, which takes Physics and Computer Science into consideration, at the example of cache side channels.

Our program analysis is the first for automatically computing reliable upper bounds on the cache-side-channel leakage of x86 binaries that use floating-point instructions. We evaluated the analysis on a simplified implementation of the QKD protocol BB84 and subsequently lifted the results to the original code.

During the evaluation, we discovered a vulnerability in the original code that might leak the entire secret key. The vulnerability is caused by an optimization in the application of an LDPC matrix to the secret key. Note that QKD is not the only place where LDPC is applied to secret information. Applications of LDPC to secrets can also be found, for instance, in post-quantum cryptography [5] and, hence, our findings should be of interest beyond QKD.

We showed how to mitigate the vulnerability by program rewriting and presented a parametric mitigation that combines the physical and software parts of QKD. Since currently the exchange of polarized photons is the performance bottleneck, a purely rewriting-based mitigation provides the best results to date. As improving the performance of the photon exchange is subject to intensive research, we expect progress in the future that might affect the trade-off, and our finding about the best trade-off should then be revisited.

Acknowledgements Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - SFB 1119 - 236615297. We also gratefully acknowledge support by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE. We thank the reviewers and Boris Köpf for their helpful comments, Tim Weißmantel for his implementation contributions, and the authors of CacheAudit for making the tool publicly available.

References

1. Aciiçmez, O., Koç, Ç.K.: Trace-driven cache attacks on AES (Short Paper). In: ICICS. pp. 112–121. LNCS 4307 (2006)
2. Ali, M., Rau, A.R.P., Alber, G.: Quantum discord for two-qubit X states. Phys. Rev. A **81**(4), 042105–1 – 042105–7 (2010)
3. Alvim, M.S., Chatzikokolakis, K., Palamidessi, C., Smith, G.: Measuring information leakage using generalized gain functions. In: CSF. pp. 265–279 (2012)
4. Américo, A., Malacaria, P.: QQIF: Quantum Quantitative Information Flow (invited paper). In: HotSpot. pp. 1–10 (2020)
5. Baldi, M., Bianchi, M., Maturo, N., Chiaraluce, F.: Improving the efficiency of the LDPC code-based McEliece cryptosystem through irregular codes. In: ISCC. pp. 000197–000202 (2013)
6. Bennett, C.H., Brassard, G.: Quantum cryptography: Public key distribution and coin tossing. In: CSSP. pp. 175–179 (1984)

7. Bernstein, D.J.: Cache-timing attacks on AES. Tech. rep., University of Illinois (2005)
8. 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: FPS. pp. 225–241. LNCS 10723 (2017)
9. Brassard, G., Salvail, L.: Secret-key reconciliation by public discussion. In: EUROCRYPT. pp. 410–423. LNCS 765 (1994)
10. Brotzman, R.L., Liu, S.L., Zhang, D., Tan, G., Kandemir, M.T.: CaSym: Cache aware symbolic execution for side channel detection and mitigation. In: S&P. pp. 505–521 (2018)
11. Cho, J.Y., Szyrkowiec, T., Griesser, H.: Quantum key distribution as a service. In: QCrypt. pp. 1–3 (2017)
12. Chothia, T., Kawamoto, Y., Novakovic, C.: A tool for estimating information leakage. In: CAV. pp. 690–695. LNCS 8044 (2013)
13. Chothia, T., Kawamoto, Y., Novakovic, C.: LeakWatch: Estimating information leakage from Java programs. In: ESORICS. pp. 219–236. LNCS 8713 (2014)
14. Cock, D., Ge, Q., Murray, T., Heiser, G.: The last mile: An empirical study of timing channels on seL4. In: CCS. pp. 570–581 (2014)
15. Cousot, P., Cousot, R.: Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. pp. 238–252 (1977)
16. Dewald, F., Mantel, H., Weber, A.: AVR processors as a platform for language-based security. In: ESORICS. pp. 427–445. LNCS 10492 (2017)
17. Diamanti, E., Lo, H.K., Qi, B., Yuan, Z.: Practical challenges in quantum key distribution. Npj Quantum Inf. **2**(1), 1–12 (2016)
18. Dixon, A.R., Sato, H.: High speed and adaptable error correction for megabit/s rate quantum key distribution. Sci. Rep. **4**(7275), 1–6 (2014)
19. Doychev, G., Köpf, B.: Rigorous analysis of software countermeasures against cache attacks. In: PLDI. pp. 406–421 (2017)
20. Doychev, G., Köpf, B., Mauborgne, L., Reineke, J.: CacheAudit: A tool for the static analysis of cache side channels. ACM Trans. Inf. Syst. Secur. **18**(1), 4:1–4:32 (2015)
21. Ekert, A.K.: Quantum cryptography based on Bell's theorem. Phys. Rev. Lett. **67**(6), 661–663 (1991)
22. Elkouss, D., Leverrier, A., Alléaume, R., Boutros, J.H.: Efficient reconciliation protocol for discrete-variable quantum key distribution. In: ISIT. pp. 1879–1883 (2009)
23. Euler, S., Beier, M., Sinther, M., Walther, T.: Spontaneous parametric down-conversion in waveguide chips for quantum information. AIP Conference Proceedings **1363**(1), 323–326 (2011)
24. Fardan, N.J.A., Paterson, K.G.: Lucky Thirteen: Breaking the TLS and DTLS record protocols. In: S&P. pp. 526–540 (2013)
25. Fraunhofer HHI: Fraunhofer HHI participates in major initiative for Quantum Communication supported by German Federal Ministry of Education and Research. https://www.hhi.fraunhofer.de/en/press-media/news/2019/fraunhofer-hhi-participates-in-major-initiative-for-quantum-communication-supported-by-german-federal-ministry-of-education-and-research.html (2019), [accessed Sep-30-2020]
26. Fry, E.S., Walther, T., Li, S.: Proposal for a loophole-free test of the Bell inequalities. Phys. Rev. A **52**(6), 4381–4395 (1995)
27. Fung, C.H.F., Qi, B., Tamaki, K., Lo, H.K.: Phase-remapping attack in practical quantum-key-distribution systems. Phys. Rev. A **75**(3), 032314–1 – 032314–12 (2007)
28. Gallager, R.: Low-density parity-check codes. IRE Trans. Inf. Theory **8**(1), 21–28 (1962)

29. Gehring, T., Händchen, V., Duhme, J., Furrer, F., Franz, T., Pacher, C., Werner, R.F., Schnabel, R.: Implementation of continuous-variable quantum key distribution with composable and one-sided-device-independent security against coherent attacks. Nat. Commun. **6**(8795), 1–7 (2015)
30. Geihs, M., Nikiforov, O., Demirel, D., Sauer, A., Butin, D., Günther, F., Alber, G., Walther, T., Buchmann, J.: The status of quantum-key-distribution-based long-term secure Internet communication. IEEE T-SUSC **6**(1), 19–29 (2021)
31. Gisin, N., Ribordy, G., Tittel, W., Zbinden, H.: Quantum cryptography. Rev. Mod. Phys. **74**(1), 145–195 (2002)
32. GitHub, Inc.: Forks of radfordneal/LDPC-codes. `https://github.com/radfordneal/LDPC-codes/network/members` (2019), [accessed Sep-30-2020]
33. Gullasch, D., Bangerter, E., Krenn, S.: Cache games - Bringing access-based cache attacks on AES to practice. In: S&P. pp. 490–505 (2011)
34. Hui, C., Wang, Y., Lu, X.: Implementation of a high throughput LDPC codec in FPGA for QKD system. In: ICSICT. pp. 1494–1496 (2016)
35. IEEE Computer Society: IEEE Standard for Information technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications – Amendment 5: Enhancements for Higher Throughput. Tech. Rep. IEEE Std 802.11n-2009, IEEE (2009)
36. Institute for Quantum Optics and Quantum Information, Austrian Academy of Sciences: QUAPITAL: Building the first reliable Quantum Internet on top of Europe's glass fiber network. `https://quapital.eu/` (2020), [accessed Sep-30-2020]
37. Intel Corporation: Intel® 64 and IA-32 Software Developer's Manual. Order Number: 325462-069US (2019)
38. Jouguet, P., Kunz-Jacques, S., Leverrier, A., Grangier, P., Diamanti, E.: Experimental demonstration of long-distance continuous-variable quantum key distribution. Nat. Photonics **7**(5), 378—-381 (2013)
39. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. In: S&P. pp. 1–19 (2019)
40. Köpf, B., Mauborgne, L., Ochoa, M.: Automatic quantification of cache side-channels. In: CAV. pp. 564–580. LNCS 7358 (2012)
41. Köpf, B., Smith, G.: Vulnerability bounds and leakage resilience of blinded cryptography under timing attacks. In: CSF. pp. 44–56 (2010)
42. Laser and Quantum Optics group (LQO) at TU Darmstadt: Open source software for control of the quantum key distribution and its postprocessing. `https://git.rwth-aachen.de/oleg.nikiforov/qkd-tools` (2020), [accessed Sep-07-2020]
43. Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., Mangard, S.: ARMageddon: Cache attacks on mobile devices. In: USENIX Security. pp. 549–564 (2016)
44. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading kernel memory from user space. In: USENIX Security. pp. 973–990 (2018)
45. Lydersen, L., Wiechers, C., Wittmann, C., Elser, D., Skaar, J., Makarov, V.: Hacking commercial quantum cryptography systems by tailored bright illumination. Nat. Photonics **4**(10), 686–689 (2010)
46. MacKay, D.J.C., Neal, R.M.: Near Shannon limit performance of low density parity check codes. Electron. Lett. **32**(18), 1645–1646 (1996)
47. Makarov, V., Hjelme, D.R.: Faked states attack on quantum cryptosystems. J. Mod. Opt. **52**(5), 691–705 (2005)

48. Malacaria, P., Khouzani, M., Pasareanu, C.S., Phan, Q., Luckow, K.S.: Symbolic side-channel analysis for probabilistic programs. In: CSF. pp. 313–327 (2018)
49. Mantel, H., Schickel, J., Weber, A., Weber, F.: How secure is green IT? The case of software-based energy Side Channels. In: ESORICS. pp. 218–239. LNCS 11098 (2018)
50. Mantel, H., Starostin, A.: Transforming out timing leaks, more or less. In: ESORICS. pp. 447–467. LNCS 9326 (2015)
51. Mantel, H., Weber, A., Köpf, B.: A systematic study of cache side channels across AES implementations. In: ESSoS. pp. 213–230. LNCS 10379 (2017)
52. Milicevic, M., Feng, C., Zhang, L.M., Gulak, P.G.: Quasi-cyclic multi-edge LDPC codes for long-distance quantum cryptography. Npj Quantum Inf. **4**(21), 1–9 (2018)
53. Mohammad, O.K.J., Abbas, S.: Detailed quantum cryptographic service and data security in cloud computing. In: ICC. pp. 43–56 (2019)
54. Molnar, D., Piotrowski, M., Schultz, D., Wagner, D.: The program counter security model: Automatic detection and removal of control-flow side channel attacks. In: ICISC. pp. 156–168. LNCS 3935 (2006)
55. Neal, R.M.: Software for Low Density Parity Check Codes, Version 2012-02-11. `http://www.cs.utoronto.ca/~radford/ftp/LDPC-2012-02-11/` (2012), [accessed Sep-20-2020]
56. Nielsen, M.A., Chuang, I.L.: Quantum computation and quantum information: 10th anniversary edition. Cambridge University Press, New York, NY, USA (2011)
57. Notz, P., Nikiforov, O., Walther, T.: Software bundle for data post-processing in a quantum key distribution experiment. Tech. rep., TU Darmstadt (2020)
58. Ollivier, H., Zurek, W.H.: Quantum discord: A measure of the quantumness of correlations. Phys. Rev. Lett. **88**(1), 017901–1–017901–4 (2002)
59. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of AES. In: CT-RSA. pp. 1–20. LNCS 3860 (2006)
60. Page, D.: Theoretical use of cache memory as a cryptanalytic side-channel. IACR Cryptology ePrint Archive **2002**(169), 1–23 (2002)
61. Pearson, D.: High-speed QKD reconciliation using forward error correction. AIP Conference Proceedings **734**(1), 299–302 (2004)
62. Poddar, R., Datta, A., Rebeiro, C.: A cache trace attack on CAMELLIA. In: InfoSecHiComNet. pp. 144–156. LNCS 7011 (2011)
63. Qi, B., Fung, C.H.F., Lo, H.K., Ma, X.: Time-shift attack in practical quantum cryptosystems. Quantum Inf. Comput. **7**(1), 73–82 (2006)
64. Rebeiro, C., Mukhopadhyay, D.: Differential cache trace attack against clefia. IACR Cryptology ePrint Archive **2010**(012), 1–11 (2010)
65. Renner, R.: Security of quantum key distribution. Phd thesis, Swiss Federal Institute of Technology Zurich (2005)
66. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In: CCS. pp. 199–212 (2009)
67. Sasaki, M., et al.: Field test of quantum key distribution in the Tokyo QKD network. Opt. Express **19**(11), 10387–10409 (2011)
68. Schmitt-Manderbach, T., Weier, H., Fürst, M., Ursin, R., Tiefenbacher, F., Scheidl, T., Perdigues, J., Sodnik, Z., Kurtsiefer, C., Rarity, J.G., Zeilinger, A., Weinfurter, H.: Experimental demonstration of free-space decoy-state quantum key distribution over 144 km. Phys. Rev. Lett. **98**(1), 010504–1 – 010504–4 (2007)
69. Schwarz, M., Lipp, M., Gruss, D., Weiser, S., Maurice, C., Spreitzer, R., Mangard, S.: KeyDrown: Eliminating software-based keystroke timing side-channel attacks. In: NDSS. pp. 1–15 (2018)

70. Smith, G.: On the foundations of quantitative information flow. In: FOSSACS. pp. 288–302. LNCS 5504 (2009)
71. Spreitzer, R., Moonsamy, V., Korak, T., Mangard, S.: Systematic classification of side-channel attacks: A case study for mobile devices. IEEE Commun. Surv. Tutor. **20**(1), 465–488 (2018)
72. Vest, G., Rau, M., Fuchs, L., Corrielli, G., Weier, H., Nauerth, S., Crespi, A., Osellame, R., Weinfurter, H.: Design and evaluation of a handheld quantum key distribution sender module. IEEE J. Sel. Top. Quantum Electron. **21**(3), 131–137 (2015)
73. Walenta, N., Burg, A., Caselunghe, D., Constantin, J., Gisin, N., Guinnard, O., Houlmann, R., Junod, P., Korzh, B., Kulesza, N., Legré, M., Lim, C.W., Lunghi, T., Monat, L., Portmann, C., Soucarros, M., Thew, R.T., Trinkler, P., Trolliet, G., Vannel, F., Zbinden, H.: A fast and versatile quantum key distribution system with hardware key distillation and wavelength multiplexing. New J. Phys. **16**(013047), 1–20 (2014)
74. Wang, S., Wang, P., Liu, X., Zhang, D., Wu, D.: CacheD: Identifying cache-based timing channels in production software. In: USENIX Security. pp. 235–252 (2017)
75. Weiser, S., Zankl, A., Spreitzer, R., Miller, K., Mangard, S., Sigl, G.: DATA – Differential Address Trace Analysis: Finding address-based side-channels in binaries. In: USENIX Security. pp. 603–620 (2018)
76. Xu, F., Ma, X., Zhang, Q., Lo, H.K., Pan, J.W.: Secure quantum key distribution with realistic devices. Rev. Mod. Phys. **92**(2), 025002–1–025002–60 (2020)
77. Yarom, Y., Falkner, K.: FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In: USENIX Security. pp. 719–732 (2014)
78. Zhang, Q., Xu, F., Li, L., Liu, N.L., Pan, J.W.: Quantum information research in China. Quantum Sci. Technol. **4**(040503), 1–7 (2019)