A Piece of QAICCC: Towards a Countermeasure Against Crosstalk Attacks in Quantum Servers

Yoann Marquer
University of Luxembourg
Luxembourg
yoann.marquer@uni.lu, 0000-0002-4607-967X

Domenico Bianculli
University of Luxembourg
Luxembourg
domenico.bianculli@uni.lu, 0000-0002-4854-685X

***Abstract*—Quantum computing, while allowing for processing information exponentially faster than classical computing, requires computations to be delegated to quantum servers, which makes security threats possible. For instance, previous studies demonstrated that crosstalk between attacker and victim's qubits can be exploited to mount security attacks.**

In this idea paper, we propose the QAICCC approach to allocate qubits between users to minimize inter-circuit crosstalk and, thus, possibilities for attacks, while maximizing qubit usage. Also, combined with existing techniques, QAICCC aims to reduce intra-circuit noise. Thus, QAICCC will support quantum computing adoption by securing the usage of quantum servers by a large number of actors.

***Index Terms*—Crosstalk analysis, Quantum server, Qubit allocation, Noise reduction, Security threat, Transpilation.**

I. Introduction

Quantum computing (QC) uses quantum bits (qubits) instead of the bits used in classical computing, enabling massive parallel computation based on quantum physics properties like quantum superposition. This allows quantum computers to process information exponentially faster than any classical computer, with empirical evidence for quantum supremacy [1]. Thus, QC has an impact on many emerging technologies [2] and industrial use cases, especially regarding optimization problems [3]. For this reason, technology giants such as IBM, Google, and Microsoft have heavily invested in QC [4].

Nevertheless, because of cost and needs for ultra-cold temperature, shielded environment, and complex wiring for control, QC is far from becoming a personal commodity [5]. Hence, access to quantum computers is likely to be provided remotely, through *quantum servers*. Sharing quantum hardware between multiple users allows to efficiently use quantum resources, but make some security threats possible [6]. For instance, correlations between attacker and victim's qubits used in the same server can be exploited in various security attacks [5] like fault-injection attacks to disrupt victim's output [7] and data-leakage attacks to retrieve victim's output [8].

Several countermeasures have been designed against the unintended interaction of qubits, called *crosstalk* [9]. Buffer qubits between user circuits could likely prevent crosstalk attacks [7], but would be a waste of quantum resources. Other approaches based on gate scheduling [9] or graph coloring [10] have been used to reduce crosstalk in quantum circuits, but never between user circuits.

In this idea paper, we address the problem of securing user circuits submitted to quantum servers, so that quantum computing can be used in a safe way by more and more actors. We propose the *Qubit Allocation for Inter-Circuit Crosstalk Countermeasure (QAICCC)* approach, which aims to maximize qubit usage, reduce inter-circuit crosstalk as well as intra-circuit noise. QAICCC performs (1) a crosstalk analysis of the targeted platform[1] to determine qubits involved in crosstalk with the largest intensity; (2) the allocation of qubits in the safest possible way; and (3) the application of existing techniques to further reduce crosstalk between circuits and noise in user circuits. The main contribution of this paper is the allocation algorithm, which aims to maximize qubit usage (including making unused qubits available for future usage) while minimizing the largest inter-circuit crosstalk error rate.

II. Background

A. Crosstalk

The qubit *connectivity* of a quantum platform determines how qubits are connected to each other. It is represented using a graph, where qubits are vertices and edges indicates which qubits are connected and thus can be used together by quantum gates to perform quantum operations. Figure 1 illustrates the qubit connectivity of an IBM 5-qubit platform; we will use this as running example.

Before execution, quantum programs must be *transpiled* 1) to add swap gates to match platform connectivity and 2) to simulate unsupported gates by combining supported ones [12]. Transpilation is usually done automatically by frameworks like Qiskit.

Qubits suffer from a short lifetime, leading to a spontaneous loss of qubit state information, called *decoherence*, because of relaxation or dephasing [10]. Another source of noise in quantum computing is the unintended interaction of qubits, called *crosstalk*, which covers a wide range of physical phenomena and varies across quantum platforms [13]. Crosstalk is usually due to qubit connectivity (adjacent qubits, like q_2 and q_3 in Figure 1, are more likely to interact) and operating frequency (qubits tuned to close frequencies can be in resonance [10]).

[1] In this article, we use the term "platform" to indicate the chosen quantum processing unit performing the computation (e.g., the `ibmqx2` platform represented in Figure 1 and IBM available platforms [11]) or a simulator.

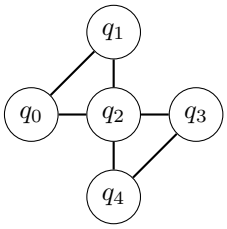

Fig. 1. Qubit connectivity of the `ibmqx2` platform.

Noise (including crosstalk) is measured using various metrics called *error rates* (e.g., Hamiltonian, Stochastic, and Affine error rates) that can be obtained using tools like the pyGSTi simulator [14]. For instance, the crosstalk error rate of qubit q_2 in Figure 1 is larger when qubits q_3 and q_4 are activated, because q_2 is connected to q_3 and q_4, and because q_2's operating frequency is close to q_4's one [7].

Ash-Saki et al. [7] demonstrated that crosstalk can be a larger source of noise than quantum gate error and decoherence. Moreover, they demonstrated that crosstalk between qubits allocated to different users can be exploited for security attacks. In their attack scenario, a user (the victim) requests qubits to the server; then, another user (the attacker) requests several copies of small quantum circuits to control the largest number possible of remaining qubits. The victim runs her circuit to obtain a result in a qubit output, e.g., q_2 in Figure 1; the attacker runs her circuit, e.g., a sequence of CNOT gates involving q_3 and q_4. Due to the presence of crosstalk from q_3 and q_4 to q_2, the magnitude of the victim's expected output can be reduced below the magnitude of other results. In other words, because of crosstalk, an attacker can use qubits (like q_3 and q_4 in the example) impacting a victim's qubit (q_2 in the example) to tamper its quantum state and even the victim's outcome, after the impacted qubit is measured. We call *crosstalk attacks* security attacks made possible by crosstalk, like Ash-Saki et al. [7]'s fault injection attack.

B. Noise reduction techniques

Current quantum computers are called NISQ (noisy intermediate-scale quantum) computers, because they are larger than small-scale prototypes with a few qubits, but not large enough so that quantum error correction can be applied [3]. Hence, current quantum executions are noisy, which decreases their *success rate*, i.e., the probability to obtain the expected output [10].

Many approaches have been proposed to reduce noise in quantum circuits; below we briefly present those used in our approach. Notably, Murali et al. [9] proposed an approach to reduce the time required to identify crosstalk by considering only adjacent pairs (as crosstalk is usually a short-range phenomenon) and performing the measurements of distant pairs in parallel. Moreover, they proposed a gate scheduling technique to reduce both decoherence and crosstalk, obtaining the XtalkSched scheduler, which writes slightly longer circuits than the current state of the art in Qiskit but outperformed it in terms of error rate. Ding et al. [10] proposed another approach to mitigate crosstalk, called ColorDynamic, for allocating qubit and gate frequencies. They considered the vertex-coloring of the connectivity graph of the targeted platform and the edge-coloring of a crosstalk graph representing relevant qubits pairs, in order to prevent close qubits or gates to share close operating frequencies and thus be in resonance.

III. Approach

A. Motivations and goals

As NISQ computers do not have enough qubits for error correction, they resort to noisy computations that hinder QC adoption. This situation warrants 1) increasing the number of available qubits and 2) reducing noise in quantum computations.

For the former, since each qubit is a precious resource that should not be wasted, we see *maximizing qubit usage*—as in allocating qubits in quantum servers to users or, if current users's needs are already satisfied, being ready for the next user(s)—as a priority.

For the latter, since crosstalk attacks are possible between users of the same quantum servers, in this paper we distinguish two kinds of noise: *inter-circuit crosstalk*, i.e., crosstalk between qubits allocated to different users, and *intra-circuit noise*, i.e., crosstalk and decoherence involving qubits allocated to the same user.

Another priority for QC adoption is that quantum servers can be used in a safe way, without a user being able to interfere with other users' executions. Hence the need to *reduce inter-circuit crosstalk*. More precisely, since crosstalk is quantified using error rate and the targeted platform may exhibit many qubit combinations leading to crosstalk, we aim to minimize the largest inter-circuit crosstalk error rate.

Finally, maximizing qubit usage while reducing inter-circuit crosstalk implies that each combination of qubits involved in crosstalk should be controlled, when possible, by the same user. Such a qubit allocation would tend to increase intra-circuit noise, which should be reduced as well to improve success rate and thus quality of service. Therefore, *intra-circuit noise reduction* is another priority in NISQ computers.

To summarize, our approach aims to achieve the following goals, in decreasing priority:

G1: maximizing qubit usage (including making unused qubits available for future usage);
G2: minimizing the largest inter-circuit crosstalk error rate;
G3: reducing intra-circuit noise (crosstalk and decoherence).

B. Overview

Figure 2 provides a graphical overview of our Qubit Allocation for Inter-Circuit Crosstalk Countermeasure (QAICCC) approach, using the UML activity diagrams notation. It takes as input the targeted quantum platform, the user circuits, and a (potentially empty) list of trusted users; it returns transpiled circuits (i.e., ready to be executed on the platform) that can be executed in a safe way, minimizing the threat of crosstalk attacks.

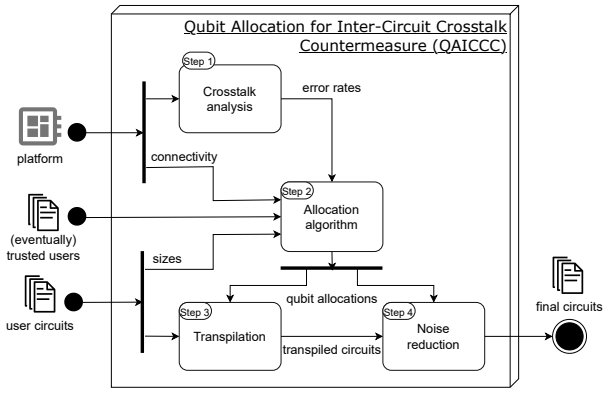

Fig. 2. Activity diagram of the Qubit Allocation for Inter-Circuit Crosstalk Countermeasure (QAICCC) approach.

The approach consists of four main steps. In Step 1, the platform is analyzed to determine crosstalk error rates (§ III-C). In Step 2, based on these error rates, the connectivity of the platform, the sizes of the user circuits, and the list of trusted users, qubits are allocated to users; we detail the allocation algorithm in § III-D. In Step 3, user circuits are transpiled according to the selected qubit allocation to match the platform connectivity and supported gates (§ III-E). In Step 4, transpiled circuits are transformed to reduce noise during quantum executions, using existing techniques like XtalkSched and ColorDynamic (§ III-F).

The QAICCC approach fulfills the goals identified in § III-A as follows. To meet G1, if there are unused qubits, these are allocated so that they are connected to each other, making them available to future users. This is done in Step 2 by introducing a new (idle) user, allocating the unallocated qubits to the idle user, and ensuring that each user's qubit allocation forms a connected component. To meet G2, qubits are allocated to minimize the largest inter-circuit crosstalk error rate. This is done through Step 1, by quantifying error rates, Step 2, by processing them in decreasing order, and through Step 4, by applying XtalkSched and ColorDynamic to reduce remaining inter-circuit crosstalk. To meet G3, XtalkSched and ColorDynamic are applied in Step 4 to minimize intra-circuit noise, while preserving the changes required to meet G2.

C. Step 1: Crosstalk analysis

In this step, the platform is analyzed (e.g., with the pyGSTi simulator [14]) to determine crosstalk error rates between qubits. We consider the impact of one or two qubits on another qubit (as in Ash-Saki et al. [7]'s work) and the impact of two qubits on two qubits (i.e., gate errors, as in Murali et al. [9]'s work), which we respectively call 1-1, 2-1, and 2-2 crosstalk.

To save time, we take inspiration from Murali et al. [9]'s methodology (Section II) by considering neighboring qubits and measuring error rates of distant qubits in parallel. More precisely, each m-n crosstalk is measured so that the $m+n$ considered qubits form a connected component. Note that this implies the existence of a path between each two considered qubits; at the same time, it does not imply that all the considered qubits are directly connected to each other. For instance, in Figure 1, qubits q_1, q_2, and q_3 form a connected component; q_1 and q_2 may impact q_3, but q_3 is not directly connected to q_1.

Algorithm 1 Qubit allocation algorithm

type: Allocation $= (Q_\varnothing, Q_T, Q_U)$, where $Q_\varnothing$ is a set of qubits and Q_T and Q_U are sets of sets of qubits
input: $C = (V, E)$, where V is a set of qubits and E is a set of unordered pairs of qubits
input: $Sizes = (S_T, S_U)$, where S_T and S_U are lists of integers
input: $ErrorRates$: List of triples $(score, Q_\rightarrow, Q_\leftarrow)$ as in Step 1 (§ III-C)
output: Set of Allocations or Error

if $\sum_{s\in S_T} s + \sum_{s\in S_U} s > \mathrm{card}(V)$ **then**
  return Error ▷ Not enough qubits
$Sizes \leftarrow \mathrm{updSizes}(V, Sizes)$ ▷ Idle user, if needed
Allocation $alloc_0 \leftarrow \mathrm{initAlloc}(V, ErrorRates)$
Set of Allocations $Pop \leftarrow \{alloc_0\}$ ▷ Initial population
Set of Allocations $Archive \leftarrow \varnothing$
$ErrorRates \leftarrow sort(ErrorRates, reverse = \mathrm{True})$
for $rate \in ErrorRates$ **do** ▷ Largest to smallest
  Set of Allocations $CurrentPop \leftarrow Pop$
  for $alloc_1 \in CurrentPop$ **do**
    Set of Allocations $Allocs \leftarrow \varnothing$
    if $\mathrm{isSafe}(alloc_1, rate)$ **then** ▷ Safe allocation pattern
      if $\mathrm{nbUsers}(alloc_1, rate) \geq 2$ **then**
        $Allocs \leftarrow \mathrm{allocMerge}(alloc_1, rate, E, Sizes)$
      $\mathrm{updAlloc}(alloc_1, rate)$ ▷ Update attributes
    else
      $Allocs \leftarrow \mathrm{allocImpacted}(alloc_1, Q_\leftarrow, C, Sizes)$
      $Allocs \leftarrow \mathrm{allocControl}(Allocs, rate, C, Sizes)$
      $Allocs \leftarrow Allocs \cup \mathrm{allocMerge}(alloc_1, rate, C, Sizes)$
      $Allocs \leftarrow Allocs \cup \mathrm{allocTrusted}(alloc_1, Q_\rightarrow, C, Sizes)$
      $Pop, Archive \leftarrow \mathrm{archAlloc}(alloc_1, Pop, Archive)$
    $Pop \leftarrow \mathrm{updPop}(Allocs, Pop, Archive, rate)$
  if $Pop = \varnothing$ **then** ▷ No new change
    break
return $Pop \cup Archive$ ▷ All the obtained allocations

In QAICCC, crosstalk is quantified using a single metrics like the *composite score*, which is the sum of the Stochastic error rates and the square of the Hamiltonian error rates, as defined in Ash-Saki et al. [7]'s work. Hence, for each m-n crosstalk, we obtain a triple $(score, Q_\rightarrow, Q_\leftarrow)$, where $score$ is the composite score, $Q_\rightarrow$ a set of m impacting qubits, and $Q_\leftarrow$ a set of n impacted qubits. We denote by $ErrorRates$ the list of such triples. For instance, the analysis of the platform shown in Figure 1 using the pyGSTi simulator, yields the following composite scores (as reported by Ash-Saki et al. [7] in their first run): 0.0027 from q_3 and q_4 to q_2, 0.0024 from q_2 and q_4 to q_3, etc. Thus, $ErrorRates = [(0.0027, \{q_3, q_4\}, \{q_2\}), (0.0024, \{q_2, q_4\}, \{q_3\}), \ldots]$.

D. Step 2: Allocation algorithm

In this step, we allocate qubits to users in order to meet G1 and G2, using Algorithm 1. The algorithm takes as input the connectivity of the platform C, the information on the size of the user circuits $Sizes$, and the list $ErrorRates$ returned by Step 1 (§ III-C).

The platform connectivity C is represented as an undirected graph (V, E), where the set of vertices V corresponds to the qubits and the set of edges E corresponds to the connected

qubit pairs. As for the information on the size of the user circuit (i.e., the number of qubits they require), since some users are trusted, we represent $Sizes$ as the pair $Sizes = (S_T, S_U)$, where S_T and S_U are lists of integers, respectively corresponding to the circuit sizes requested by trusted and untrusted users.

We introduce the concept of qubit *allocation*, i.e., how the qubits in V are either unallocated or allocated to users. Each allocation, corresponding to the type Allocation declared on the first line of Algorithm 1, is represented as a triple $(Q_\varnothing, Q_T, Q_U)$, where $Q_\varnothing$ is the set of unallocated qubits and Q_T and Q_U are sets of sets of qubits, representing the qubits allocated to, respectively, trusted and untrusted users. Moreover, each allocation has several attributes: *Incidental* is a list of remaining inter-circuit *incidental crosstalk*, (eventually) to be minimized in Step 4 (§ III-F); *score* is the score used to rank allocations (the lower the score, the more secure the allocation); *penalty* represents the penalty of the allocation, i.e., it is a metric quantifying cumulative incidental crosstalk and used to rank allocations in case of a tie in score; *lastRate* is the first encountered error rate for which the allocation is unsafe. For a given allocation, if a qubit is allocated to a user, we say that this user *controls* the qubit.

The algorithm takes inspiration from genetic search by maintaining an evolving population of possible allocations, but does not involve any randomness. To minimize the largest inter-circuit crosstalk error rate, it processes error rates returned by Step 1 (§ III-C) in decreasing order, so that the qubits involved in crosstalk with the largest intensity are allocated first. For a given error rate, each allocation in the population is tested to determine if it is already safe, i.e., impacting qubits are allocated to trusted users or impacted users control impacting qubits. If the allocation is safe, then it is used in the next iteration. Otherwise, it is removed from the population, and then (when possible, due to size constraints) new safe allocations are generated from it and added to the population. In this way, the last individuals in the population correspond to allocations of qubits done in the safest possible way.

The algorithm starts by testing if the number of qubits to allocate is larger than the number of available qubits (Line 1). If so, then no qubit allocation is possible and QAICCC returns an error (Line 2). Instead, if the number of qubits to allocate is smaller than the number of available qubits, then this means some qubits are not used by the users' circuits. In this case, to meet G1, the algorithm should allocate qubits for potential, future users. To do so, we use the auxiliary function `updSizes`, which works as follows. It introduces an untrusted *idle user* requesting for the rest of the qubits, so that all the qubits will be allocated, then it updates the tuple $Sizes$ accordingly (Line 3). Since each user's allocation will form a connected component, the unused qubits will be connected, which increases the chance of being able to allocate them to new users. If the users already requested all the available qubits, then there is no need for an idle user and, thus, $Sizes$ is not updated.

The initial allocation $alloc_0$ is generated at Line 4 such that all the qubits are unallocated, its *score* is higher than the ones in $ErrorRates$ (to reflect that it is unsafe), its *penalty* is zero, and its *Incidental* is empty. The *population*, denoted by Pop, is a set of allocations which are safe for the crosstalk error rates in $ErrorRates$ investigated so far. The population is initialized at Line 5 with the initial allocation. The *archive*, denoted by $Archive$, is a set of allocations which are unsafe for at least one error rate; it is initialized empty at Line 6.

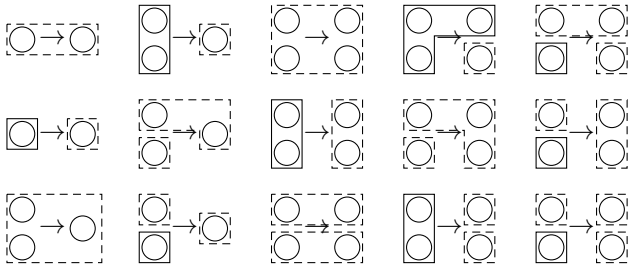

Fig. 3. Safe allocation patterns for 1-1, 2-1, and 2-2 crosstalk: impacting qubits $Q_\rightarrow$ are on the left of an arrow and impacted qubits $Q_\leftarrow$ on the right, each box with plain (resp. dashed) edges represents qubits allocated to a trusted (resp. any) user.

To meet G2, we minimize the largest inter-circuit crosstalk error rate. The list of error rates $ErrorRates$ obtained during crosstalk analysis (§ III-C) is first sorted in decreasing order of $score$ (Line 7); then, each error rate $rate = (score, Q_\rightarrow, Q_\leftarrow)$ is processed to obtain safe allocations (Lines 8-24). More specifically, at each iteration, the current population is stored in an ancillary variable (Line 9) and each allocation $alloc_1$ in the current population is used in the body of the loop to (eventually) update Pop (Line 10). Depending on $alloc_1$'s properties, new allocations are (eventually) generated and added to the population. This set of new allocations is denoted by $Allocs$ and initialized empty at Line 11. Each new allocation inherits its attributes from $alloc_1$ and is generated by updating $alloc_1$ to address the current error rate. Moreover, to ensure user circuits can be transpiled using the obtained qubit allocations, each user's allocation is determined so that it forms a connected component. For instance, if two users respectively submit a 2-qubit and a 3-qubit circuit for the platform illustrated in Figure 1, an allocation where q_0, q_2, and q_3 are controlled by the same user will not be considered, as it is not possible to use q_0, q_2, and q_3 in one circuit and q_1 and q_4 in another one, since q_1 and q_4 are not connected.

First, $alloc_1$ is tested to determine if it is already safe with respect to $rate$ (Line 12), i.e., if the allocation matches a *safe allocation pattern* depicted in Figure 3. In short, a pattern is safe if a trusted user controls at least one impacting qubit or if all impacted users control at least one impacting qubit[2]. Note that trusted users are not necessary, as several safe patterns for 1-1, 2-1, and 2-2 crosstalk do not involve them.

Second, in case qubits involved in crosstalk are allocated to several users, some crosstalk may occur when these qubits are

[2]It is not enough to spread the control of impacting qubits to different users, since they may be in collusion or simply be the same actor having submitted several circuits.

activated at the same time. In the context of this paper, this phenomenon is called *incidental crosstalk* since, as opposed to a crosstalk attack, it is not intentionally triggered by a user. Thus, even if the allocation matches a safe pattern, it is tested to determine the number of users controlling the qubits of $rate$ (Line 13). If only one user is involved, then there is no need for new allocations, thus $Allocs$ remains empty. Otherwise, new allocations involving only one user are generated using the auxiliary function `allocMerge` (Line 14) which reallocates the involved qubits to a unique user, by merging the allocations of users controlling an involved qubit. At this point (Line 15), since $alloc_1$ is safe, it is kept for the next iteration and its attributes are updated using the auxiliary function `updAlloc` as follows: its score becomes $score$ and, if $alloc_1$ involves several users controlling qubits in $rate$, then its penalty is increased by $score$ and $rate$ is appended to $\mathrm{Incidental}(alloc_1)$.

If $alloc_1$ is unsafe, then new safe allocations are generated and stored in $Allocs$ as follows (Lines 17-20). First, all the unallocated impacted qubits are allocated to users (Line 17); then, function `allocControl` updates the allocations so that each impacted user u_2 controls an impacting qubit (Line 18). These updates are achieved either by allocating an unallocated (if any) impacting qubit to u_2 or by merging u_2's qubits with qubits controlled by a user u_1 controlling an impacting qubit. At this stage, the set of allocation is further updated, by allocating all the qubits involved in the error rate to the same user (Line 19)—as in Line 14—and by allocating an unallocated (if any) impacting qubit to a trusted user (Line 20). Then, since $alloc_1$ is unsafe, it has to be removed from the current population. This is achieved by calling function `archAlloc`, which sets $\mathrm{lastRate}(alloc_1)$ to $rate$, removes $alloc_1$ from Pop, and adds it to $Archive$. Since only allocations in the population will be investigated in future iterations, $alloc_1$'s attributes will not change until the end of the algorithm.

After these tests, $Allocs$ (if not empty) contains allocations which are safe for all the crosstalk error rates investigated so far. The population is (eventually) updated at Line 22. More precisely, for each $alloc_2$ in $Allocs$, if $alloc_2$ does not already belong to $Pop \cup Archive$, then its attributes are updated (as in Line 15) and it is added to the population. The purpose of this condition is to prevent a conflict of attributes between several occurrences of the same allocation.

Finally, if no allocation can address $rate$, then the population becomes empty (Line 23). Since no population change can occur from this state, the loop stops (Line 24). In this case or if all crosstalk error rates were exhausted, Algorithm 1 returns all the obtained allocations (Line 25).

The additional auxiliary functions used in Algorithm 1 are detailed in Appendix A.

E. Step 3: Transpilation

In this step, user circuits are transpiled according to the selected qubit allocation.

Allocations returned by Algorithm 1 are first sorted by increasing score; in case of a tie, they are further sorted by increasing $\mathrm{penalty}$ In this way, the initial (empty) allocation is the last one, while the allocations obtained last with Algorithm 1 (i.e., those addressing the largest number of error rates) are the first ones. Starting from the first allocation, an allocation is selected and tested as follows. Qiskit is called to transpile the user circuits according to the current allocation, to match the platform connectivity and supported gates. If the transpilation is successful, then the transpiled circuits are selected for Step 4. Otherwise, the next allocation is selected and tested. This process continues until an allocation is successful. If no allocation is successful, then QAICCC returns an error.

During transpilation, gates acting on more than two qubits are decomposed in binary gates supported by the platform; hence, a transpiled circuit will contain only unary and binary gates. Moreover, since qubits allocated to each circuit form a connected component (§ III-D) and swap gates ensure that, in a connected component, any pair of qubits can be used in a quantum operation, it is likely that all allocations would be successfully transpiled (depending on Qiskit's behavior).

F. Step 4: Noise reduction

In this step, QAICCC uses existing techniques like the XtalkSched scheduler [9] and ColorDynamic [10] (Section II) on the transpiled circuits to reduce noise further. These techniques can be used either in isolation or in combination. ColorDynamic can be used only if the platform supports frequency allocation, while there are no preconditions for XtalkSched. If used in combination, since ColorDynamic does not impact the circuit schedule, it should be used first (to reduce crosstalk) and then followed by XtalkSched (to reduce the remaining noise).

To meet G2, the priority is to reduce first remaining inter-circuit crosstalk of the qubit allocation selected in Step 3 (§ III-E), i.e., error rates in its attribute *Incidental*; then, error rates starting from its attribute *lastRate* are also considered as candidates for reduction. Note that these error rates are already sorted by decreasing order of $score$ (§ III-D). Hence, noise reduction techniques can be applied following the same order to reduce the error rates.

To meet G3, the aforementioned noise reduction techniques can be also independently applied to each circuit, to reduce intra-circuit noise (crosstalk and decoherence). Since G2 has higher priority than G3, such techniques have to be applied in a way that preserves the changes (e.g., frequency allocation for ColorDynamic and gate scheduling for XtalkSched) required to meet G2.

G. Application to the Running Example

Let us assume that two untrusted users respectively submit a 2-qubit and a 3-qubit circuit for the platform illustrated in Figure 1. Step 1 obtains $ErrorRates = [\,(0.0027, \{q_3, q_4\}, \{q_2\}),$ $(0.0017, \{q_1, q_2\}, \{q_0\}),\ (0.0013, \{q_2, q_4\}, \{q_0\}),\ \ldots]$.

In Step 2, the allocation algorithm determines that all the qubits are used, hence no idle user is necessary (Line 3). The population is initialized (Line 5) with the initial individual $(Q_\varnothing, Q_T, Q_U)$ where all the qubits are

unallocated, i.e., $Q_{\varnothing} = \{q_0, q_1, q_2, q_3, q_4\}$, $Q_T = \varnothing$, and $Q_U = \varnothing$. Since there is no trusted user in this example, we simply denote an allocation by (Q_U), omitting $Q_{\varnothing} = E \setminus Q_U$ and $Q_T = \varnothing$. The iteration (Line 8) starts with the largest error rate $(0.0027, \{q_3, q_4\}, \{q_2\})$. Since the only individual present in Pop is $(\varnothing)$, which is unsafe since qubits involved in the error rate could be allocated to any user, it is archived and new allocations are generated. First, the impacted qubit is allocated (Line 17), obtaining the allocation $(\{\{q_2\}\})$. Second, impacting qubits are allocated to the user controlling the impacted qubit (Line 18), so $(\{\{q_2\}\})$ is replaced by $(\{\{q_2, q_3\}\})$ and $(\{\{q_2, q_4\}\})$. Because the unallocated controlling qubit could be allocated to another user, these allocations have a penalty of 0.0027. Then, the algorithm adds allocation $(\{\{q_2, q_3, q_4\}\})$, where only one user is involved, to the population (Line 19). This allocation has the same score (i.e., 0.0027) as the allocations $(\{\{q_2, q_3\}\})$ and $(\{\{q_2, q_4\}\})$, but its penalty is 0. Since no user is trusted in this example, Line 20 never generates new allocations. Then, the algorithm continues with the next error rate $(0.0017, \{q_1, q_2\}, \{q_0\})$. To address this error rate, the population evolves from $\{(\{\{q_2, q_3\}\}), (\{\{q_2, q_4\}\}), (\{\{q_2, q_3, q_4\}\})\}$ to $\{(\{\{q_0, q_1\}, \{q_2, q_3\}\}), (\{\{q_0, q_1\}, \{q_2, q_4\}\}), (\{\{q_0, q_1\}\{q_2, q_3, q_4\}\})\}$. Finally, the algorithm continues with the next error rate $(0.0013, \{q_2, q_4\}, \{q_0\})$. This time, no allocation is safe and no safe allocation can be generated. Hence, the allocations are archived (Line 21) and the iteration stops (Line 24). Finally, Algorithm 1 returns the archived allocations (Line 25).

In Step 3, QAICCC will select allocation $(\{\{q_0, q_1\}, \{q_2, q_3, q_4\}\})$, as it has the same score as the other best allocations, but has a smaller penalty. Both circuits would successfully be transpiled according to this allocation, which reduces inter-circuit crosstalk compared to the allocation $(\{\{q_0, q_1, q_2\}, \{q_3, q_4\}\})$ exploited by Ash-Saki et al. [7]'s injection attack.

However, even if reduced, inter-circuit crosstalk may still subsist. In Step 4, noise reduction techniques may be used to further reduce noise. For instance, XtalkSched can be used to reduce inter-circuit crosstalk, e.g., from q_2 and q_4 to q_0, then to reduce intra-circuit crosstalk, e.g., from q_2 and q_3 to q_4.

IV. Related Work

Ash-Saki et al. [7] proposed buffer qubits between user circuits to prevent inter-circuit crosstalk as well as the injection attack they demonstrated. While such an approach can meet G2, it would be a waste of quantum resources, failing to meet G1. This contrasts with QAICCC, which does not need to unallocate qubits to prevent attacks and makes unused qubits accessible for the next users.

To the best of our knowledge, existing reducing-crosstalk techniques like XtalkSched [9] and ColorDynamic [10] do not take into account security vulnerabilities. Nevertheless, they are useful to meet G3 and, if they can be updated to prioritize inter-circuit over intra-circuit crosstalk and larger error rates over smaller ones (§ III-F), then they can contribute to meeting G2. Moreover, ColorDynamic can only be used when qubit operating frequencies can be controlled by the software (e.g., tunable qubit architectures proposed in some prototypes [15] or by Google [16]), while XtalkSched and QAICCC can be used on any platform. Finally, XtalkSched focuses on gate errors, i.e., 2-2 crosstalk (§ III-D), while QAICCC uses qubit allocation to also reduce 1-1 and 2-1 crosstalk.

V. Research Outlook and Conclusions

In this idea paper, we presented QAICCC, an approach for securing users' program executions from crosstalk attacks in quantum servers. We proposed a qubit allocation algorithm that maximizes qubit usage while minimizing the largest inter-circuit crosstalk error rate. Below, we outline our research plan to further develop this idea into a full-fledged solution.

First, we plan to set-up an environment for Qiskit and pyGSTi to replicate Ash-Saki et al. [7]'s crosstalk attack on several platforms. This attack acts on qubits involved in crosstalk and consists of executing several times a CNOT gate on impacting qubit(s) to reduce the magnitude of the desired output in the impacted qubit(s) below the magnitude of other results. Thus, for various platform and qubit combinations, we will be able to determine the correlation between the number of CNOT gate executions necessary for the attack and the different error rates measured by pyGSTi. Such a correlation will allow us to assess whether the composite score they proposed in their study is the best surrogate metric to quantify the strength of the threat; shouldn't this be the case, we will investigate alternative metrics.

Second, we will implement QAICCC in Python for better integration with frameworks like Qiskit and PennyLane, and apply it to various platforms and quantum circuits.

Furthermore, we plan to conduct an empirical evaluation of QAICCC (either with a simulator or with a quantum processor, depending of available ressources). The qubit allocation algorithm will be evaluated by comparing the largest inter-circuit crosstalk error rate between its allocation and an allocation performed by the last version of Qiskit. QAICCC will be evaluated by simulating various combinations of attacker, victim, trusted, and untrusted users and comparing—in terms of success rate—the execution of QAICCC's transpiled circuits with transpiled circuits obtained by baselines like Qiskit and XtalkSched alone.

Acknowledgment

This project has received funding from SES and the Luxembourg National Research Fund under the Industrial Partnership Block Grant (IPBG), ref. IPBG19/14016225/INSTRUCT.

References

[1] F. Arute, K. Arya, R. Babbush, D. Bacon, J. Bardin, R. Barends, R. Biswas, S. Boixo, F. Brandao, D. Buell, B. Burkett, Y. Chen, Z. Chen, B. Chiaro, R. Collins, W. Courtney, A. Dunsworth, E. Farhi, B. Foxen, and J. Martinis, "Quantum supremacy using a programmable

superconducting processor," *Nature*, vol. 574, pp. 505–510, 10 2019.

[2] M. A. Akbar, A. A. Khan, S. Mahmood, and S. Rafi, "Quantum software engineering: A new genre of computing," in *Proceedings of the 1st ACM International Workshop on Quantum Software Engineering: The Next Evolution*, ser. QSE-NE 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 1–6. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3663531.3664750

[3] P. Liimatta, P. Taipale, K. Halunen, T. Heinosaari, T. Mikkonen, and V. Stirbu, "Research versus practice in quantum software engineering: Experiences from credit scoring use case," *IEEE Software*, vol. 41, no. 6, pp. 9–16, Nov 2024.

[4] A. A. Khan, M. Azeem Akbar, and P. Liang, "First international workshop on quantum software engineering: The next evolution (qse-ne) summary 2024," *SIGSOFT Softw. Eng. Notes*, vol. 49, no. 4, p. 26–28, Oct. 2024. [Online]. Available: https://doi.org/10.1145/3696117.3696124

[5] A. A. Saki, M. Alam, K. Phalak, A. Suresh, R. O. Topaloglu, and S. Ghosh, "A survey and tutorial on security and resilience of quantum computing," in *2021 IEEE European Test Symposium (ETS)*. 3 Park Avenue, New York City, U.S.: Institute of Electrical and Electronics Engineers, May 2021, pp. 1–10.

[6] S. Ghosh, S. Upadhyay, and A. A. Saki, "A primer on security of quantum computing," 2023. [Online]. Available: https://arxiv.org/abs/2305.02505

[7] A. Ash-Saki, M. Alam, and S. Ghosh, "Analysis of crosstalk in nisq devices and security implications in multi-programming regime," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 25–30. [Online]. Available: https://doi.org/10.1145/3370748.3406570

[8] A. Ash-Saki and S. Ghosh, "Qubit sensing: A new attack model for multi-programming quantum computing," 04 2021.

[9] P. Murali, D. C. Mckay, M. Martonosi, and A. Javadi-Abhari, "Software mitigation of crosstalk on noisy intermediate-scale quantum computers," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1001–1016. [Online]. Available: https://doi.org/10.1145/3373376.3378477

[10] Y. Ding, P. Gokhale, S. F. Lin, R. Rines, T. Propson, and F. T. Chong, "Systematic crosstalk mitigation for superconducting qubits via frequency-aware compilation," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 3 Park Avenue, New York City, U.S.: Institute of Electrical and Electronics Engineers, Oct 2020, pp. 201–214.

[11] IBM, "Quantum processing units," (Accessed 2025-02-25). [Online]. Available: https://quantum.ibm.com/services/resources

[12] M. Paltenghi and M. Pradel, "Analyzing quantum programs with lintq: A static analysis framework for qiskit," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3660802

[13] M. Sarovar, T. Proctor, K. Rudinger, K. Young, E. Nielsen, and R. Blume-Kohout, "Detecting crosstalk errors in quantum information processors," *Quantum*, vol. 4, p. 321, Sep. 2020. [Online]. Available: https://doi.org/10.22331/q-2020-09-11-321

[14] pyGSTi, "A python implementation of gate set tomography," 0.9.13. [Online]. Available: https://github.com/sandialabs/pyGSTi

[15] M. D. Hutchings, J. B. Hertzberg, Y. Liu, N. T. Bronn, G. A. Keefe, M. Brink, J. M. Chow, and B. L. T. Plourde, "Tunable superconducting qubits with flux-independent coherence," *Phys. Rev. Appl.*, vol. 8, p. 044003, Oct 2017. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevApplied.8.044003

[16] R. Barends, C. M. Quintana, A. G. Petukhov, Y. Chen, D. Kafri, K. Kechedzhi, R. Collins, O. Naaman, S. Boixo, F. Arute, K. Arya, D. Buell, B. Burkett, Z. Chen, B. Chiaro, A. Dunsworth, B. Foxen, A. Fowler, C. Gidney, M. Giustina, R. Graff, T. Huang, E. Jeffrey, J. Kelly, P. V. Klimov, F. Kostritsa, D. Landhuis, E. Lucero, M. McEwen, A. Megrant, X. Mi, J. Mutus, M. Neeley, C. Neill, E. Ostby, P. Roushan, D. Sank, K. J. Satzinger, A. Vainsencher, T. White, J. Yao, P. Yeh, A. Zalcman, H. Neven, V. N. Smelyanskiy, and J. M. Martinis, "Diabatic gates for frequency-tunable superconducting qubits," *Phys. Rev. Lett.*, vol. 123, p. 210501, Nov 2019. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.123.210501

APPENDIX

A. Auxiliary functions

The additional auxiliary functions used in Algorithm 1 are detailed in Figure 4. Function `allocImpacted` allocates unallocated impacted qubits; it takes as input the current allocation $alloc_1$ and the impacted qubits $Q_{\leftarrow}$, and returns the set of allocations $Allocs_1$. Function `allocControl` updates allocations in $Allocs_1$ so that each impacted user controls at least one impacting qubit; it takes as input the set of allocations $Allocs_1$ and the current error rate $rate$, and returns the updated set of allocations $Allocs_1$. Function `allocMerge` generates new allocations involving only one user; it takes as input the current allocation $alloc_1$ and the current error rate $rate$, and returns the set of allocations $Allocs_1$. Function `allocTrusted` allocates unallocated impacting qubits to trusted users; it takes as input the current allocation $alloc_1$ and the impacting qubits $Q_{\rightarrow}$, and returns the set of allocations $Allocs$.

These four auxiliary functions also take as input (but without using them directly) the platform connectivity $C = (V, E)$ and the circuit sizes $Sizes$ because they call the `connect` and `newAlloc` functions, which require these inputs. Function `connect` is responsible for finding possible paths to connect qubits in a set of qubits $User$ with qubits in a set of qubits S and for using these paths to obtain (if any) connected components matching the circuit size limits. It takes as input an allocation $alloc_1$, the sets of qubits $User$ and S, the platform connectivity C and the circuit sizes $Sizes$; it returns a set of allocations $Allocs$. If $User$ is empty (e.g., when `connect` is called but no qubit has been allocated so far), then paths consist of either S itself, if it is connected, or connected components containing S. Finally, `newAlloc` generates a new allocation; it takes as input an allocation $alloc_1$, the set of qubits $User$, the current error rate $rate$, the platform connectivity C and the circuit sizes $Sizes$, and returns a set of allocations $Allocs$. A new allocation $alloc_2$ is generated based on $alloc_1$, so qubits in $User$ are allocated to the same user. Then, `newAlloc` checks if $alloc_2$ is compatible with the platform connectivity C and satisfies the circuit size limits $Sizes$. If so, $alloc_2$'s attributes are updated by the auxiliary function `updAlloc` according to the error rate $rate$, then $Allocs = \{alloc_2\}$ is returned; otherwise, $Allocs = \varnothing$ is returned.

procedure allocImpacted($alloc_1, Q_{\leftarrow}, E, Sizes$)
  Set of Allocations $Allocs_1 \leftarrow \{alloc_1\}$
  for Qubit $q \in Q_{\leftarrow}$ **do**
    Set of Allocations $Allocs_2 \leftarrow \varnothing$
    for Allocation $alloc_2 = (Q_{\varnothing}, Q_T, Q_U) \in Allocs_1$ **do**
      if $q \in Q_{\varnothing}$ **then**
        for Set of Qubits $User \in Q_T \cup Q_U$ **do**
          $Allocs_2 \leftarrow Allocs_2 \cup \text{connect}(alloc_2, User, \{q\}, C, Sizes)$
      else
        $Allocs_2 \leftarrow Allocs_2 \cup \{alloc_2\}$
    $Allocs_1 \leftarrow Allocs_2$
  return $Allocs_1$

procedure allocControl($Allocs_1, rate = (score, Q_{\rightarrow}, Q_{\leftarrow}), C, Sizes$)
  for Qubit $q_1 \in Q_{\leftarrow}$ **do**
    Set of Allocations $Allocs_2 \leftarrow \varnothing$
    for Allocation $alloc_2 = (Q_{\varnothing}, Q_T, Q_U) \in Allocs_1$ **do**
      Set of Qubits $User_1 \leftarrow \text{getUser}(q_1, alloc_2)$
      for Qubit $q_2 \in Q_{\rightarrow}$ **do**
        if $q_2 \in Q_{\varnothing}$ **then**
          $Allocs_2 \leftarrow Allocs_2 \cup \text{connect}(alloc_2, User_1, \{q_2\}, C, Sizes)$
        else
          Set of Qubits $User_2 \leftarrow \text{getUser}(q_2, alloc_2)$
          $Allocs_2 \leftarrow Allocs_2 \cup \text{connect}(alloc_2, User_1, User_2, C, Sizes)$
    $Allocs_1 \leftarrow Allocs_2$
  return $Allocs_1$

procedure allocMerge($alloc_1 = (Q_{\varnothing}, Q_T, Q_U), rate = (score, Q_{\rightarrow}, Q_{\leftarrow}), C, Sizes$)
  Set of Allocations $Allocs_1 \leftarrow \varnothing$
  Set of Qubits $S \leftarrow Q_{\rightarrow} \cup Q_{\leftarrow}$ ▷ involved qubits
  Set of Qubits $User_{merge} \leftarrow \bigcup_{q \in S} \text{getUser}(q, alloc_1)$
  if $User_{merge} = \varnothing$ **then** ▷ are unallocated
    Set of Allocations $Allocs_2 \leftarrow \text{newAlloc}(alloc_1, S, rate, C, Sizes)$
    if $Allocs_2 \neq \varnothing$ **then**
      $Allocs_1 \leftarrow Allocs_2$
    else
      for $User \in Q_T \cup Q_U$ **do**
        $Allocs_1 \leftarrow Allocs_1 \cup \text{connect}(alloc_1, User, S, C, Sizes)$
  else
    $Allocs_1 \leftarrow \text{newAlloc}(alloc_1, User_{merge} \cup S, rate, C, Sizes)$
  return $Allocs_1$

procedure allocTrusted($alloc_1 = (Q_{\varnothing}, Q_T, Q_U), Q_{\rightarrow}, C, Sizes$)
  Set of Allocations $Allocs \leftarrow \varnothing$
  for Set of Qubits $S \subseteq Q_{\rightarrow} \cap Q_{\varnothing}$ **do**
    for Set of Qubits $User \in Q_T$ **do**
      if $Q_{\rightarrow} \cap (User \cup S) \neq \varnothing$ **then**
        $Allocs \leftarrow Allocs \cup \text{connect}(alloc_1, User, S, C, Sizes)$
  return $Allocs$

procedure connect($alloc_1 = (Q_{\varnothing}, Q_T, Q_U), User, S, C, Sizes$)
  Set of Allocations $Allocs \leftarrow \varnothing$
  Integer $maxLen \leftarrow \text{rem}(User, alloc_1, Sizes) - \text{card}(S \setminus User)$
  for Set of Qubits $Path \in \text{getPaths}(User, S, Q_{\varnothing}, C, maxLen)$ **do**
    $Allocs \leftarrow Allocs \cup \text{newAlloc}(alloc_1, User \cup S \cup Path, rate, C, Sizes)$
  return $Allocs$

Fig. 4. Auxiliary functions used in Algorithm 1