IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 4 November 2024, accepted 23 December 2024, date of publication 31 December 2024, date of current version 8 January 2025.

Digital Object Identifier 10.1109/ACCESS.2024.3524309

RESEARCH ARTICLE

A Toolchain for Assisting Migration of Software Executables Towards Post-Quantum Cryptography

NORRATHEP RATTANAVIPANON, JAKAPAN SUABOOT, AND WARODOM WERAPUN
College of Computing, Prince of Songkla University, Phuket Campus, Phuket 83120, Thailand

Corresponding author: Jakapan Suaboot (jakapan.su@psu.ac.th)

This work was supported by the National Science, Research and Innovation Fund (NSRF) and Prince of Songkla University under Grant COC6701016S.

ABSTRACT Quantum computing poses a significant global threat to modern security mechanisms. As such, security experts and public sectors have issued guidelines to help organizations transition their software to post-quantum cryptography (PQC). However, there is a lack of (semi-)automatic tools to support this transition, particularly for software deployed as binary executables. To address this gap, in this work, we first propose a set of requirements necessary for this type of tool to detect quantum-vulnerable software executables. Following these requirements, we introduce QED: a toolchain for Quantum-vulnerable Executable Detection. QED uses a three-phase approach to identify quantum-vulnerable dependencies in a given set of executables, from file-level to API-level, and finally, precise identification of a static trace that triggers a quantum-vulnerable API. The key benefit of this design is that it provides efficiency without compromising accuracy, as it incorporates fast initial analyses to filter out executables unlikely to be quantum-vulnerable that in turn allows the more resource-intensive analysis to be performed on a smaller subset of executables. To demonstrate this claim, we evaluate QED on both a synthetic dataset with four cryptography libraries and a real-world dataset with over 200 software executables. The results show that: 1) QED discerns quantum-vulnerable from quantum-safe executables with 100% accuracy in the synthetic dataset; 2) QED is practical and scalable, completing analyses on average in less than 4 seconds per real-world executable; and 3) QED reduces the manual workload required by analysts to identify quantum-vulnerable executables in the real-world dataset by more than 90%.

INDEX TERMS Binary analysis, post-quantum cryptography, post-quantum migration, software security.

I. INTRODUCTION

Advancements in quantum computing are accelerating as demonstrated by multiple recent breakthroughs. From two qubits in 1998 [1], quantum computers are now capable of supporting over 1,000 qubits [2]. Although the current number of qubits (and the error-correcting capability) is still insufficient for practical use, the rapid pace of quantum computing development along with substantial investment from major tech companies leads many experts to believe that a fully-functional quantum computer will become commercially viable within the next two decades [3]. This potential has far-reaching implications: both positive and negative.

Quantum computers hold the distinct advantage of leveraging quantum mechanics to solve mathematical problems that are not feasible with classical computers. This capability could positively benefit society in various ways, such as optimizing traffic systems [4], enhancing machine learning [5], and accelerating drug and material discovery [6]. However, this same capability poses significant negative impacts. Malicious actors could exploit quantum computers to undermine today's widely-used security mechanisms, particularly public-key cryptosystems. Previous studies have

The associate editor coordinating the review of this manuscript and approving it for publication was Shuangqing Wei.

shown that breaking the RSA public-key cryptosystem would require approximately 4,098 logical (error-free) qubits [7], while breaking elliptic curve cryptography would need 2,330 logical qubits [8]. Moreover, since symmetric cryptosystems often rely on public-key cryptosystems for their initial key exchange, they are also vulnerable. This vulnerability is demonstrated by "harvest now, decrypt later" strategy [9], where encrypted data is intercepted and stored today with the intent to decrypt it once sufficiently powerful quantum computers become available. This means that any sensitive data encrypted today, even with symmetric-key cryptosystems, could be compromised in the future if proactive measures are not taken immediately.

Given these significant security risks, it is clear that preparations must be made now against potential quantum attacks. In response, researchers and public sectors are urging all organizations to prepare for these risks [9], [10], [11], [12]. NIST, for instance, recommends establishing quantum-readiness teams within organizations to prepare for the migration of their software systems to post-quantum cryptography (PQC) [11]. The preparation includes creating cryptographic inventories to assess the use of cryptography within the organization and conducting risk assessments based on these inventories.

Despite existing recommendations and guidelines, there are currently no (semi-)automatic tools specifically designed for assisting in this task. Consequently, in practice, analysts must rely on a mix of various scattered tools and manual analysis to identify software systems that are vulnerable to quantum attacks. This results in significant and tedious effort, especially given the potential large amount of software systems within organizations. Moreover, the analysts often lack access to the source code and therefore must perform PQC migration based solely on program binaries. This in turn requires advanced knowledge in binary analysis, which translates to increased costs in terms of budget, time, and labor, making it unaffordable for small and medium-sized businesses.

Motivated by these challenges, we first formulate the requirements for a tool to assist in migration of software executables towards PQC. After establishing these, we introduce Quantum-vulnerable Executable Detection (QED), a toolchain designed for detecting quantum-vulnerable executables. The main goal of QED is to aid analysts in identifying quantum-vulnerable (QV) executables located in a specific computer/server within an organization. The design rationale of QED revolves around two key observations.

First, as a security practice, software applications typically do not implement cryptographic functionalities within themselves but rather utilize them via API access to well-established cryptography libraries (e.g., OpenSSL, wolfSSL, MbedTLS). Hence, QV executables can be determined based on the existence of the dependencies with QV APIs provided by these libraries. Second, since most programs either are non-cryptographic or use relatively quantum-safe cryptography (e.g., cryptographic hash functions), QED should employ a fast analysis to eliminate this type of executables in the earlier phases. This allows for an accurate but more resource-intensive analysis to be performed on a smaller set of more-likely-to-be-QV candidates later. To achieve this, we design QED to operate in three phases where:

- QED's first phase aims to determine software executables that have dependencies on QV cryptography libraries. This is accomplished using a fast static analysis that quickly detects and excludes non-cryptographic programs that do not rely on QV cryptography libraries.
- In the second phase, QED first identifies QV APIs within the detected QV cryptography libraries and then uses a lightweight analysis to exclude executables that do not have clear access to these QV APIs.
- The final phase conducts a more precise and resource-intensive static call graph analysis on the remaining executables. It traces function calls from the executable's entry point (e.g., the `main` function) to a QV API, providing precise evidence that the reported executables are indeed QV.

Each phase of QED provides a human-readable report, informing analysts about post-quantum risk assessment of each program executable. This information can then be used for decision-making in later PQC migration plans within organizations.

Contribution: In this work, we aim to make the following contributions:

1) We outline the requirements of a tool aimed at assisting in a PQC migration task of software executables, with a specific focus on identifying QV software executables.
2) We introduce a toolchain, called QED, designed to meet the proposed requirements. QED implementation is open-sourced and publicly available at https://github.com/norrathep/qed.git.
3) We empirically validate the accuracy and efficiency of QED using both synthetic and real-world datasets. The results show that QED produces only one false negative while achieving a 100% true positive rate and reducing the analyst's manual workload in identifying QV software by over 90%.

Overall, we hope QED can become a crucial tool in easing the transition of existing software systems to PQC, especially for small and medium-sized organizations that lack the labor and resources needed for PQC migration.

II. RELATED WORK

In this section, we review related work, focusing on existing tools for detecting cryptographic binaries in Section II-A, current PQC algorithms and standardization process in Section II-B and finally, recent efforts related to PQC migration in Section II-C.

A. CRYPTOGRAPHIC DISCOVERY TOOLS IN SOFTWARE EXECUTABLES

Existing work leverages either static or dynamic analysis to discover cryptography usage in software binaries. Static analysis based approaches typically rely on known static features, such as initialization vectors, look-up tables (e.g., S-Boxes), or a sequence of assembly instructions, to detect cryptography implementation in executables [13], [14]. These approaches are lightweight in nature but may miss detecting some cryptography functions when executables are optimized by an aggressive compiler, which causes a modification to the target static features in binaries.

Another line of approaches utilizes dynamic analysis to overcome this limitation. One common method is to identify cryptography usage based on runtime information, e.g., the presence of loops [15], loop structures [16], or the input-output relationship of a function [17], [18], collected from execution traces. In practice, generating meaningful runtime execution traces can be challenging, especially when the analyst has no access to the source code or is not fully familiar with the software. To address this challenge, recent approaches have proposed using dynamic features created through symbolic execution, eliminating the need for actual runtime execution traces. For example, ALICE [19] and Harm-DOS [20] perform symbolic execution on binary functions to determine whether their output matches the expected hash output, thereby implementing the expected hash function without running the entire software. Additionally, the work in [21] extends the concept of Data Flow Graph isomorphism [14] with symbolic analysis to detect proprietary cryptography implementations in software binaries.

Our approach falls into the category of static analysis. However, unlike existing static approaches, our method detects cryptography usage based on the API name information, which is always present in binaries despite aggressive compiler optimizations, hence incurring no false negatives. Compared to existing dynamic approaches, QED does not require running executables to collect execution traces or performing heavyweight symbolic analysis. Moreover, none of the existing tools focuses on identifying *quantum-vulnerable* cryptography usage in executables or consider a scenario where cryptography implementations are dynamically linked with executables.

B. PQC ALGORITHMS AND STANDARDIZATION

Currently deployed public-key cryptosystems have been shown to be vulnerable to quantum attacks [7], [8]. This prompted researchers to develop new PQC algorithms that are secure against both quantum and classical computers. As discussed in [22], existing PQC algorithms generally fall into four major families: hash-based cryptography [23], lattice-based cryptography [24], code-based cryptography [25], and multivariate polynomial cryptography [26].

Recently, NIST announced new standards for PQC, which include the Module-Lattice-Based Key-Encapsulation Mechanism Standard [27], the Module-Lattice-Based Digital Signature Standard [28], and the Stateless Hash-Based Digital Signature Standard [29]. These standards offer organizations practical and concrete options for transitioning to PQC by replacing currently used public-key cryptosystems with these quantum-resistant algorithms.

We refer interested readers to [30] for a more comprehensive survey of PQC algorithms and the NIST standardization process for PQC selection.

C. TRANSITIONING ORGANIZATIONS TO PQC

Migrating organizations towards PQC is currently an ongoing global effort. Related work in this area [11], [31], [32], [33] has focused on providing advisories and guidelines for PQC migration to organizations. At high-level, the migration process can be structured into four steps [32]:

1) **Diagnosis.** Identify QV cryptography used within an organization.
2) **Planning.** Use the information collected from the previous step to create a plan and a timeline for migration.
3) **Execution.** Perform the migration according to the previously identified plan.
4) **Maintenance.** Monitor changes within the organization after migration execution to maintain PQC.

Our work falls into the first step that aims to reduce manual workloads required by organizations to identify software executables vulnerable to quantum attacks. The results produced by our tool can be subsequently used in the later steps to execute and maintain the PQC migration plans.

Another closely related concept is crypto agility [34], which refers to the property of a system that can be easily adapted to a different cryptographic algorithm. In contrast to our work, this concept involves the second and third steps of the migration process. We expect that after identifying QV software systems using our tool, the organization can replace them with quantum-safe and crypto-agile alternatives.

The migration process can also be categorized based on where QV cryptography is used: (1) in compiled binary executables and their dependencies, (2) in assets on external/end-user systems, and (3) in the network communication layer [11], [33]. Our work targets the first type of usage, while the others are potential avenues for future research.

Major cryptography libraries are in the process of incorporating PQC into their implementations, such as Open Quantum Safe OpenSSL[1] and Bouncy Castle.[2] However, as these implementations are still under active development, they are not yet ready for deployment in production software [32]. To the best of our knowledge, there is currently

[1] https://openquantumsafe.org/applications/tls.html
[2] https://www.bouncycastle.org/documentation/specification_interoperability/

no other concrete, publicly available tool specifically geared towards the identification of software binaries that utilize QV APIs from cryptography libraries. As such, we believe our tool will be useful in reducing the cost of PQC transition, especially for small to medium-sized organizations.

III. QED TOOLCHAIN

In this section, we begin with a discussion of the problem statement in Section III-A, outline the tool requirements in Section III-B, and conclude with a detailed explanation of our tool's design in Section III-C.

A. PROBLEM STATEMENT & DEFINITIONS

In this work, we envision a scenario where an analyst wants to take the first step toward PQC migration by identifying QV software within an organization. The organization may consist of computers/servers, each containing several software systems. Within the scope of this work, we consider the computers to be Linux machines with software programs written in `C` or `C++`, deployed as Linux executables in the Executable and Linkable Format (ELF). Nonetheless, the overall idea in this work can be extended to programs written in other languages and running under different operating systems.

We assume that a software executable utilizes cryptography by accessing APIs provided by common cryptography libraries through dynamic linking. This means software executables that implement cryptographic functions within themselves or statically link with cryptography libraries are considered out of scope. This assumption aligns with the standard security and program linking practices for Linux machines [35].

We define a software executable as QV if there is at least one possible execution path from its entry point (i.e., the start address of the `main` function) to one of the cryptography library's APIs implementing QV algorithms (e.g., RSA, Diffie-Hellman Key Exchange, Elliptic Curve-based Digital Signatures). This API may reside within the same executable or in a shared library used at runtime by this executable. Then, given a list of software executables, we consider a tool effective if it can identify all executables that are QV according to this definition, along with human-understandable evidence supporting their classification.

B. TOOL REQUIREMENTS

We now formulate the requirements for a tool designed to address the above-mentioned scenario.

R1 **Dynamic Linking.** The tool must be able to identify executables that use QV APIs via dynamic linking.

R2 **Binary-level Analysis.** The tool must not rely on the availability of executables' source code in any part of its analysis. This is because programs deployed in an organization may be developed by third parties who are unwilling to share/publish the source code due to intellectual property concerns.

R3 **Static-only Features.** The tool must operate using only static features of executables without making use of dynamic features (e.g., runtime execution traces). In practice, an organization could contain too many software executables for analysts to know how to meaningfully run each of them in order to generate the required dynamic features.

R4 **Scalability.** The tool must support the analysis for a large number of software executables while being able to complete its analysis within a reasonable runtime, on the order of minutes.

R5 **Effectiveness.** As the main goal is to reduce the analyst's workload by minimizing the number of executables required for manual analysis, the tool must incur no false negatives, i.e., it should effectively identify all QV executables. While high accuracy is crucial, the tool can tolerate a small false positive rate, i.e., it may classify a small set of executables as QV when they are not. The task of analyzing and eliminating these false positives is manageable when they are few and can be handled through the analyst's manual review.

C. PROPOSED SOLUTION

Figure 1 overviews the workflow in the proposed QED toolchain. It takes two inputs: (i) a list of software executables and (ii) descriptions of QV cryptography libraries. For (i), the analyst can supply a directory path storing executables (e.g., `/bin/`) to QED; whereas the second input (ii) can be created using a list of QV APIs that uniquely identify the library of interest. For example, to describe OpenSSL 1.1, the analyst creates a list of its QV APIs, i.e., `{RSA_Sign, EC_Sign, ..., DH_compute_key}` as a JSON file (see Cryptolib Desc in Figure 1) and feeds it to QED. This list can be compiled manually by reviewing all available APIs and identifying the ones that are QV, or semi-automatically by first filtering APIs based on specific QV keywords (e.g., `RSA`, `EC` and `DH`) and then manually analyzing them.

QED consists of three phases where each utilizes increasingly complex analysis that in turn yields more accurate results. All phases rely on only static analysis, eliminating the need to understand the context of executables to run them. At the end of each phase, QED produces human-readable reports that describe the risk level associated with quantum threats for each executable, along with supporting evidence. The analyst can then choose which report to use based on whether speed or accuracy is prioritized. If accuracy is favored, the analyst can work with the output from the final phase, which takes the longest to run but yields the fewest false positives. Meanwhile, if speed is more important, the analyst can stop QED at an earlier phase and use its reports instead.

In what follows, we structure the explanation of each phase of QED by first providing a high-level overview that

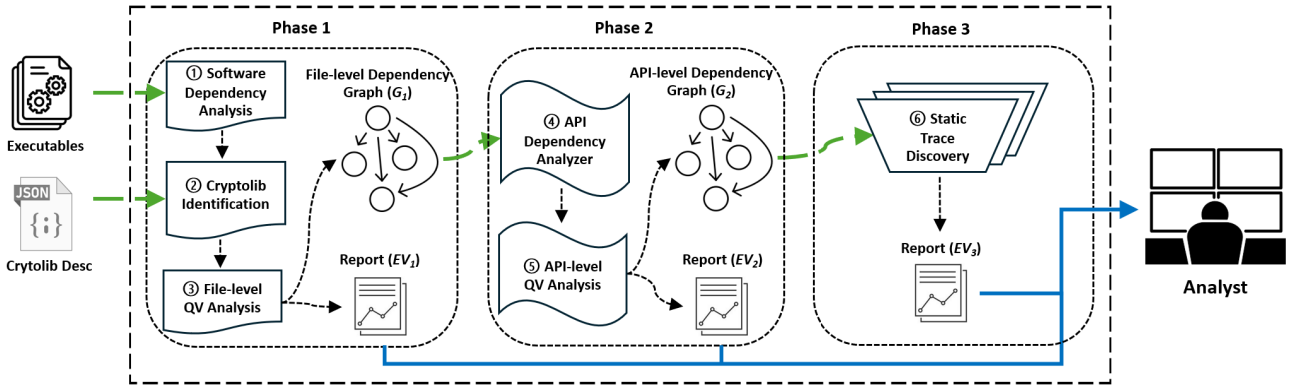

FIGURE 1. QED **components, where green dashed arrows connect each phase of** QED **with its input while the blue arrow represents** QED**'s final output.**

is accessible to a general audience. We then delve into the technical details, which we believe to be useful for technical analysts or researchers who may wish to adapt our work to different settings. Finally, we describe the format and interpretation of the evidence produced by each phase and conclude each phase's explanation with a discussion of its accuracy.

1) PHASE 1: FILE-LEVEL DEPENDENCY ANALYSIS

High-Level Overview: Given a list of executables (*execs*) and QV APIs exported by cryptography libraries (*descs*), the first phase (P1) of QED aims to identify executables in *execs* with file-level dependencies on these libraries. At a high level, this is achieved by first determining all software dependencies associated with *execs* (step ① in Figure 1). Then, in step ②, P1 inspects each binary in the identified dependencies to check for matches with QV cryptography libraries specified in *descs*. Once the relevant dependencies and cryptography libraries are located, P1 conducts a file-level QV analysis to assess whether an executable has a dependency on any of these QV cryptography libraries. If it does not, P1 classifies the executable as non-QV and removes it from further analysis in step ③. P1 then outputs potential QV executables as a file-level dependency graph (G_1) to be analyzed in the next phase.

Technical Details: Algorithm 1 shows details of this phase. First, we model software dependency of *execs* as a directed graph $G_1 = (V_1, E_1)$, where V_1 is the set of files (including *execs* and shared objects they depend on) and E_1 is the set of tuples describing a file-level direct dependency relationship. For example, $(\texttt{sftp}, \texttt{libcrypto.so}) \in E_1$ means that the `sftp` executable directly depends on `libcrypto.so` (OpenSSL library), i.e., `sftp` contains a call to one of the functions exported by `libcrypto.so`.

P1 generates G_1 by performing a depth-first search to discover all dependencies of *execs*, as described in `GenSWDepGraph` of Algorithm 1. Once G_1 is constructed, P1 locates QV cryptography libraries in V_1. This is done in `FindCryptoLibs` of Algorithm 1 by analyzing exported functions of each node and checking whether these functions match with one of crypto libraries' descriptions in *descs*. When they do, it considers the node to implement the same QV APIs, and therefore most likely to be the target cryptography library.

After discovering all cryptography libraries (*cryptoLibs*) in V_1, P1 prunes out the nodes that have no dependency on *cryptoLibs*, which is done by checking whether a path from a node to any of *cryptoLibs* exists (Lines 5-9 in Algorithm 1). As a result, P1 passes a refined G_1, consisting only of executables with file-level dependencies on QV APIs, to QED's next phase.

Evidence: P1 reports EV_1 to analysts, which consists of file-level dependency paths from each executable in *execs* to a cryptography library. An example of EV_1 is shown in Figure 2, illustrating how `sftp` has a direct dependency on `libcrypto.so` while `nmap` has an indirect dependency on `libcrypto.so` through its dependency on the `libssl.so` library, which itself depends on `libcrypto.so`.

Accuracy: P1 yields no false negatives: suppose executable e is QV according to the definition in Section III-A. This means either e or one of its shared libraries must invoke one of the QV APIs. The former case implies e has a direct dependency on the cryptography library, while the latter results in an indirect dependency through the shared library. Therefore, in either case, there must be a file-level dependency between e and a cryptography library, making e appear in G_1 and EV_1.

Nonetheless, P1 incurs substantial false positives since it may incorrectly report executables that depend on a cryptography library but utilize none of its QV APIs (e.g., it uses the quantum-safe `SHA512` algorithm from OpenSSL).

2) PHASE 2: API-LEVEL DEPENDENCY ANALYSIS

High-Level Overview: To reduce the false positives in P1, QED's second phase (P2) analyzes executables' dependencies at API-level. As shown in step ④ of Figure 1, P2 starts by using the file-level dependency information (from P1) to identify the APIs used/exported within each dependency link. With this information, P2 can estimate whether a specific

FIGURE 2. **Evidence reports produced by** QED**'s first phase (left), second phase (middle) and third phase (right), where** `libcrypto.so.1.1` **represents a shared library object for OpenSSL v1.1.**

API is reachable by a given executable. Then, in step ⑤, P2 performs an API-level QV analysis to evaluate whether any QV APIs specified in *descs* can be reached by an executable in *execs*. If no QV API is reachable, P2 treats the executable as non-QV and removes it from further analysis. Finally, P2 outputs the remaining executables as an API-level dependency graph (G_2) to the next phase.

Technical Details: P2 is detailed in Algorithm 2. It aims to construct an API dependency graph $G_2 = (V_2, E_2)$, where V_2 is the set of files, similar to V_1, while E_2 corresponds to the set of three-element tuples describing an API-level dependency relationship. As a concrete example, (`sftp`, `libcrypto.so`, {`DSA_do_sign`, `...`}) $\in E_2$ indicates file `sftp` executable invokes external functions, including `DSA_do_sign`, that are implemented and exported by `libcrypto.so`.

P2 starts by initializing G_2 as a copy of G_1 and determining predecessor nodes of *cryptoLibs*, i.e., those with direct dependencies on *cryptoLibs*. Then, it checks whether these nodes contain function calls to QV APIs, specified in *descs*. If no such calls exist, it identifies the node as a false positive and its API-level dependency is excluded in G_2 by removing the edge connecting this node and the library node.

Next, P2 locates other nodes that do not depend on predecessors of *cryptoLibs* by checking for the absence of a path between these nodes and any *cryptoLibs* nodes in G_2 (Lines 12-16 in Algorithm 2). Once these nodes are identified, P2 removes them, leaving G_2 with only nodes that have dependencies on QV predecessors.

After that, P2 goes through each directed edge in G_2 and aims to embed it with a set of APIs used/exported by the connecting nodes. For an edge from node n_1 to node n_2, P2 identifies a set of external APIs used in n_1 and a set of APIs exported by n_2 (found in `.dynsym` section of n_1 and n_2 ELF executables, respectively). It then intersects these sets to find the APIs invoked by n_1 that belong to n_2, and updates the edge with this API-level dependency information (Lines 17-20 in Algorithm 2). It then passes G_2 to the next next phase.

Evidence: P2 outputs EV_2 to analysts. Similar to EV_1, EV_2 contains file-level dependency paths where the first element is in *execs* and the last is in *cryptoLibs*. Unlike EV_1, EV_2 ensures that a predecessor of *cryptoLibs* (the second last element) in this path contains a call to QV APIs, which are also documented in EV_2. It also implies that when an executable e has a direct dependency on one of *cryptoLibs* (i.e., e is itself a predecessor of the library), EV_2 guarantees e to contain invocations to QV APIs.

An example of EV_2 is shown in Figure 2, in which it reports an indirect dependency of `nmap` executable on OpenSSL (`nmap` → `libssl.so` → `libcrypto.so`), indicating that `nmap` could potentially invoke any of QV APIs listed in `QV_APIs` during runtime execution.

Accuracy: P2 also avoids false negatives for a similar reason as in P1. However, it addresses false positives in which e exclusively uses quantum-safe APIs from *cryptoLibs* without relying on other QV shared libraries. Nevertheless, false positives may still occur in two scenarios:

1) e contains invocations to external functions provided by a shared library (different from *cryptoLibs*), where

Algorithm 1 File-Level Dependency Analysis (P1)

```
   Input: execs – list of software executables
1        descs – list of crypto library's descriptions
   Output: G1, EV1
2  Algorithm P1 (execs, descs)
3      G1 ← GenSWDepGraph (execs)
4      cryptoLibs ← FindCryptoLibs (G1.nodes)
5      foreach n ∈ G1.nodes do
6          if ¬ G1.hasPath(n, cryptoLibs) then
7              G1.removeNode(n)
8          end
9      end
10     EV1 ← {}
11     foreach e ∈ execs do
12         EV1.append(G1.paths(e, cryptoLibs))
13     end
14     return G1, EV1, cryptoLibs
15
16 Procedure GenSWDepGraph (execs)
       // Initialize G as empty directed
          graph
17     G ← DiGraph ()
18     foreach e ∈ execs do
19         G ← GenSWDepGraphHelper (G, e)
20     end
21     return G
22
23 Procedure GenSWDepGraphHelper (G, e)
24     if e ∈ G.nodes then
25         return G
26     end
27     G.addNode(e)
28     libs ← GetDirectDep(e)
29     if libs is φ then
30         return G
31     end
32     foreach l ∈ libs do
33         G.addEdge(e, l)
34         G ← GenSWDepGraphHelper (G, l)
35     end
36     return G
37
38 Procedure FindCryptoLibs (nodes, descs)
39     cryptoLibs ← {}
40     foreach n ∈ nodes do
41         f ← GetExportFuncs(n)
42         if ∃ d ∈ descs s.t. d ⊆ f then
43             cryptoLibs.append(d)
44         end
45     end
47     return cryptoLibs
48
```

these functions do not use QV APIs but other functions exported by the same shared library do. In this case, P2 still classifies e as QV since it propagates QV detection from the shared library, even though e does not use any of its QV exported functions.

2) e may contain calls to QV APIs, but this function call could be inside "dead" code that is never executed at runtime. In this scenario, P2 incorrectly considers e as QV since it can only determine whether these calls exist in e, but not whether they will actually be executed at runtime.

Algorithm 2 API-Level Dependency Analysis (P2)

```
   Input: G1, execs, cryptoLibs
   Output: G2, EV2
1  Algorithm P2 (G1, execs, cryptoLibs)
2      G2 ← G1
3      foreach c ∈ cryptoLibs do
4          pred ← G2.predecessors(c)
5          foreach p ∈ pred do
6              f ← GetExternFuncs(p)
7              if f ∩ c.desc = φ then
8                  G2.removeEdge(p, c)
9              end
10         end
11     end
12     foreach n ∈ G2.nodes do
13         if not G2.hasPath(n, cryptoLibs) then
14             G2.removeNode(n)
15         end
16     end
17     foreach (src, dst) ∈ G2.edges do
18         apis ← GetExternFuncs(src) ∩
             GetExportFuncs(dst)
19         G2.setAPI((src, dst), apis)
20     end
21     EV2 ← {}
22     foreach e ∈ execs do
23         EV2.append(G2.paths(e, cryptoLibs))
24     end
25     return G2, EV2
```

3) PHASE 3: STATIC TRACE ANALYSIS

High-Level Overview: The third phase (P3) of QED utilizes the API-level dependency information (from P2) as input and aims to accurately identify executables that meet the QV definition in Section III-A.

To achieve this, P3 performs static trace discovery (step ⑥ in Figure 1) to identify execution traces starting from the executable entry point (e.g., `main` address) leading to a QV API call. These identified traces are recorded as entries in EV_3. As each entry serves as proof of how a QV API can be invoked by the given executable, it gives assurance to analysts that the reported executable is indeed QV.

P3 supports two modes: normal and conservative. In normal mode, the absence of an execution trace directly indicates the executable is non-QV. In contrast, the conservative mode treats missing traces as potential false negatives and returns the evidence from the previous phase (i.e., EV_2) to the analyst for further manual verification. These modes offer a trade-off between minimizing false negatives and reducing manual effort, allowing the analysts to select based on their specific use cases.

Technical Details: Recall that a path in EV_2 corresponds to a list of files: $[n_1, n_2, \ldots, n_k]$, where $n_1 \in execs$, $n_k \in cryptoLibs$ and n_i has direct dependency on n_{i+1}. In the special case of n_{k-1}, it additionally guaratnees that

Algorithm 3 Static Trace Analysis (P3)

```
   Input: G_2, EV_2, conservative
   Output: EV_3
1  Algorithm P3(G_2, EV_2)
2      EV_3 ← {}
3      foreach path ∈ EV_2 do
4          mainAddr ← GetMainAddress(path[0])
5          trace ← GetStaticTrace(G_2, path, 0, mainAddr)
6          if trace is not φ then
7              EV_3.append(trace)
8          else if conservative then
9              EV_3.append(path)
10         end
11     end
12     return EV_3
13
14 Procedure GetStaticTrace(G, path, i, fromFunc)
15     apis ← G.getAPI((path[i], path[i+1]))
16     CG ← GenCallgraph(path[i])
17     foreach toFunc ∈ apis do
18         t1 ← CG.path(fromFunc, toFunc)
19         if i is len(path)−1 then
20             return t1
21         end
22         if t1 is not φ then
23             t2 ← GetStaticTrace(G, path, i+1, toFunc)
24             if t2 is not φ then
25                 return t1||t2
26             end
27         end
28     end
29     return φ
```

n_{k-1} contains a call to a QV API provided by *cryptoLibs*. For instance, as shown in Figure 2, one path in EV_2 is [`nmap`, `libssl.so`, `libcrypto.so`], in which `nmap` is an executable in *execs* and `libcrypto.so` is one of the identified QV cryptography library (i.e., OpenSSL v1.1). This path also indicates that `libssl.so` calls QV APIs (e.g., `DSA_do_sign`) exported by `libcrypto.so`.

To create EV_3, P3 first analyzes each file for each path in EV_2 using `GetStaticTrace` of Algorithm 3. For the first file, n_1, it enumerates n_2's external functions (stored in E_2) and, for each n_2's external function f_{ex}, checks whether a call to f_{ex} is reachable from the executable entry point, i.e., the `main` function in this case. P3 performs this check by constructing a static callgraph of n_1, represented as a directed graph with functions as nodes and invocations as edges. Then, it validates reachability by checking whether a path from the `main` node to the f_{ex} node exists within this callgraph.

If reachability is not found, P3 concludes that f_{ex} cannot be accessed by n_1's runtime execution and thus moves on the analysis to the next n_2's external function. Upon discovering reachability, P3 repeats the analysis on the next file n_2, i.e., by constructing n_2's callgraph and performing the reachability analysis with f_{ex} as the entry point and n_3's external function as the endpoint. This process continues until it finds a reachable path in n_{k-1}, as indicated in Algorithm 3.

Evidence: P3 generates a static trace in EV_3 by concatenating all paths returned by the reachability analyses from files $n_1,\ldots, n_{k-1}$. An example of this is shown in Figure 2, where EV_3 reports that `nmap` is QV since a QV API, `EVP_PKEY_get0_RSA`, can be invoked via the following function calls: `main` (from `nmap`) → `sub_3f340` (from `nmap`) → `...` → `SSL_CTX_new` (from `libssl.so`) → `...` → `EVP_PKEY_get0_RSA` (from `libcrypto.so`).

When P3 fails to find a static trace for a specific executable and is configured as the conservative mode, P3 considers this executable a false negative and returns the corresponding entry from EV_2 for that executable (Lines 8-10 of Algiorthm 3).

Accuracy: EV_3 minimizes false positives since an entry in EV_3 represents a static execution trace from an executable entry point to a QV API call, demonstrating how this QV API can be invoked at runtime. Nonetheless, P3's reachability analysis may incur false negatives as constructing a complete callgraph for `C` programs is still considered an open problem [36]. To avoid false negatives, EV_3 may fall back to the results from EV_2 in such cases (as discussed above), prompting the analyst to perform manual analysis on them later.

IV. EVALUATION

A. IMPLEMENTATION AND EXPERIMENTAL SETUP

We implemented the QED toolchain using ≈800 lines of Python3 code. It leverages pyelftools[3] library to detect ELF files and extract relevant information, such as dynamic symbols as well as exported and external functions, from the detected ELF files. Directed graphs in QED are built and manipulated using NetworkX library.[4] Meanwhile, QED's P3 constructs static callgraphs from executables via the angr framework [36]. We conducted experiments with QED on an Ubuntu 20.04 system running atop an Intel i5-8520U @ 1.6GHz with 24GB of RAM.

To measure QED's accuracy, we use true positive rate (TPR) and true negative rate (TNR) metrics. A true positive (TP) occurs when QED correctly identifies a QV executable, while a false positive (FP) happens when QED incorrectly classifies a non-QV executable as QV. True negatives (TN) and false negatives (FN) are defined similarly. For clarity, these prediction outcomes are illustrated in Figure 3.

The main challenge in evaluating QED's accuracy is the difficulty of precisely identifying false negatives (i.e., QV executables that QED might miss) incurred by QED. In real-world datasets, each binary executable may depend on multiple complex shared libraries. Thus, to accurately identify all false negatives in this case, it would require manually checking each shared library for QV API usage (in addition to the application binaries); this represents a large

[3]https://github.com/eliben/pyelftools
[4]https://networkx.org/

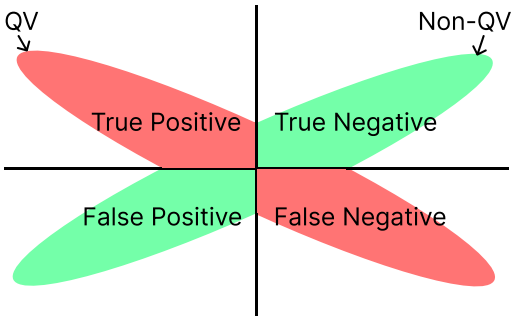

FIGURE 3. QV and non-QV prediction outcomes.

amount of manual effort that becomes practically infeasible as the number of software executables in the dataset grows.

To overcome this, we evaluated QED using two datasets: synthetic and real-world. The synthetic dataset was specifically crafted to allow us to determine the ground truth (i.e., whether an executable is QV or non-QV) of executables, enabling us to precisely measure QED's accuracy (including false negatives). In contrast, our evaluation of the real-world dataset focuses more on QED's scalability and quantifies the reduction in manual workload saved by QED.

B. SYNTHETIC DATASET: DESCRIPTION

The synthetic dataset includes four cryptography libraries: OpenSSL v1.1.1,[5] OpenSSL v3.3.1,[6] MbedTLS v2.28.8,[7] and wolfSSL v5.7.2.[8] To create this dataset, we ported example programs provided by the respective libraries, where each program directly calls the library APIs to implement a specific cryptographic primitive. We selected five primitives (and thus five programs in total) for each library: SHA-512, AES-256, Diffie-Hellman Key Exchange, RSA, and ECDSA, with only the last three being QV in this dataset. This amounts $5 \times 4 = 20$ executables in the first set. As the programs in this set utilize the library APIs directly within their executable, we call this set "Direct Dependency".

Further, to test QED against indirect usage of QV APIs, we developed a second set of C programs that have indirect dependency on the crypto library, i.e., they do not directly call the crypto library APIs but rather access them through a separate shared library; as such, we denote this set "Indirect Dependency". To implement this set, we created a single shared library exporting five APIs, each implementing one of the aforementioned cryptographic primitives. We then constructed five C programs, each dynamically linking with this shared library to invoke one of its exported APIs. As with the first set, we utilized the same four cryptography libraries, resulting in $5 \times 4 = 20$ executables in the second set. In total, the synthetic dataset comprises 40 executables with 24 QV and 16 non-QV executables, as summarized in Table 1.

TABLE 1. Breakdown of number of executables in synthetic dataset.

Synthetic Dataset	QV	Non-QV	Total
Direct Dependency	12	8	20
Indirect Dependency	12	8	20
Total	24	16	40

C. SYNTHETIC DATASET: ACCURACY RESULTS

Table 2 depicts the accuracy of QED at each phase. The results show that P1 identifies all executables in the synthetic dataset as positives (or QV) because all of them have file-level dependencies on cryptography libraries. This leads to a 100% TPR, but also in the misclassification of 16 non-QV executables, resulting in a 0% TNR.

After running P2, QED removes all false negatives (i.e., 8 non-QV executables) in the Direct Dependency set in which, despite dynamically linking with cryptography libraries, they exclusively use their non-QV APIs. However, P2 still fails to identify non-QV executables in the Indirect Dependency set because it cannot track API usages across executables. Despite this, TNR improves to 50% in P2 while maintaining a perfect TPR. Finally, by performing static trace analysis, P3 can discover the remaining false positives and eliminate all of them, achieving 100% TPR and TNR.

D. REAL-WORLD DATASET: DESCRIPTION

In the second dataset, we evaluated QED on real-world software executables. Specifically, we selected four sets of Linux software in this dataset:

- **Coreutils**[9] and **UnixBench**[10] are non-cryptographic software commonly used to benchmark the performance of Unix-like machines. As executables in this set are non-cryptographic, they are considered to be non-QV.
- **Network** includes 13 popular Linux networking utility programs, including curl, dig, netcat, nmap, nslookup, ping, scp, sftp, ssh, sshd, telnet, tracepath and wget. Due to the small number, we managed to manually inspect their source codes/binaries to determine the ground truth. Out of these, we found 7 programs to be QV as they rely on QV APIs provided by OpenSSL v1.1.
- **tpm2-tools**[11] implements TPM functionalities in software and uses cryptographic features through the tpm2-tss library,[12] which in turn calls the cryptography APIs from OpenSSL v1.1. Unlike the previous set, this set contains a large number of programs (86), making it impractical to determine their ground truths manually.

Table 3 summarizes the statistics of all sets in our real-world dataset. In total, this dataset contains 226 executables with an

[5] https://github.com/openssl/openssl/tree/OpenSSL_1_1_1f
[6] https://github.com/openssl/openssl/tree/openssl-3.3.1; note that we distinguish between OpenSSL v3.3 and v1.1, as many APIs from OpenSSL v1.1 are deprecated in v3.3.
[7] https://github.com/Mbed-TLS/mbedtls/tree/v2.28.8
[8] https://github.com/wolfSSL/wolfssl/tree/v5.7.2-stable
[9] https://github.com/coreutils/coreutils/tree/v9.5
[10] https://github.com/kdlucas/byte-unixbench/tree/v5.1.3
[11] https://tpm2-software.github.io.
[12] https://github.com/tpm2-software/tpm2-tss

TABLE 2. Accuracy of QED evaluated on the synthetic dataset, where # QV (# Non-QV) represents the number of QV (non-QV) executables input to each phase and TPR/TNR refers to the true positive/negative rate.

QED's Phases (→)	P1		P1 +P2		P1 +P2 +P3	
Synthetic Dataset (↓)	TP/FN (TPR)	TN/FP (TNR)	TP/FN (TPR)	TN/FP (TNR)	TP/FN (TPR)	TN/FP (TNR)
Direct Dependency	12/0 (100%)	0/8 (0%)	12/0 (100%)	8/0 (100%)	12/0 (100%)	8/0 (100%)
Indirect Dependency	12/0 (100%)	0/8 (0%)	12/0 (100%)	0/8 (0%)	12/0 (100%)	8/0 (100%)
Total	24/0 (100%)	0/16 (0%)	24/0 (100%)	8/8 (50%)	24/0 (100%)	16/0 (100%)

TABLE 3. Statistics of real-world dataset; "# Execs" represents the number of executables included in each set, "# Unique Deps" is the number of unique dependencies required for each set and "Avg. # Deps" is the average number of dependencies per executable.

Set	# Execs	Avg. Exec Size (in KB)	# Unique Deps	Avg. # Deps	Avg. Dep Size (in KB)
Coreutils	109	355	7	1.52	492
UnixBench	18	19	2	1.06	1660
Network	13	950	76	13.38	898
tpm2-tools	86	53	48	9.45	433
Total	226	248	84	5.19	834

average size of 248KB and each depending on an average of 5 shared libraries with an average size of 834KB.

E. REAL-WORLD DATASET: RESULTS

1) ACCURACY

As mentioned in Section IV-A, our evaluation of the real-world dataset focuses on performance/manual effort reduction rather than accuracy, as obtaining the ground truth of large-scale real-world executables is practically infeasible. However, for the sake of completeness, we report the accuracy results on the Coreutils, UnixBench, and Network sets, where their ground truths can be reliably determined. The results for these sets are summarized in Table 4.

As non-cryptographic software, Coreutils and UnixBench have no dependencies on any cryptography library, they are non-QV. QED identifies these and ends its analyses at P1 without the need to further run P2 and P3, yielding 100% TPR and TNR. In contrast, the Network set potentially relies on cryptography implementations from OpenSSL. In this set, P1 classifies 9 programs as QV, passing them to the subsequent phases. Of these, 2 are non-QV, resulting in 2 false positives for P1; these false positives are then detected and excluded in P2. We manually inspect these false positives and found that they are caused from sfp and scp programs that dynamically link with OpenSSL but invoke only its non-QV APIs (e.g., `RAND_bytes`); this is not detectable in P1.

More importantly, the results in Table 4 support QED's design choice that aims to minimize false negatives. Both P1 and P2 produce no false negatives, and only one false negative of the curl program is reported by P3 in the Network set. Upon closer inspection, this false negative occurs because, although curl uses QV APIs from OpenSSL, its usage is based on indirect calls (e.g., through function pointers), which are notoriously difficult to detect through static analysis [36]. Nonetheless, we emphasize that if avoiding any false negatives is critical, the analyst can opt for the conservative version of P3. This version would classify curl as QV, providing its EV_2 evidence to the analyst for further manual analysis to verify its QV status.

2) PERFORMANCE

Runtime results of QED on the real-world dataset are reported in Figure 4. As expected by QED's design, P1 and P2 complete their analyses very quickly (under 8 seconds) across all sets. In contrast, as P3 involves expensive static callgraph analysis, it takes significantly more time and thus dominates QED's overall runtime. Overall, QED takes around 15 minutes to process all 226 executables, resulting in an average of 4 seconds per executable.

To better understand how the size of analyzed files affects P3's runtime, we measured the time required to construct a static callgraph (representing the main overhead in P3) for each analyzed file in the Network and tpm2-tools sets. The results shown in Table 5 indicate that while P3 processes 70 more files in Network than in tpm2-tools, the majority of the analyzed files in tpm2-tools are much smaller, averaging around 54KB for 86 executables. Consequently, P3 can build callgraphs of these smaller files very quickly, under 3 seconds.

However, the Network set encompasses 15 files that are over 500KB in size, leading to more than 10 times longer for the callgraph construction time. This relationship is further illustrated in Figure 5, which shows the time required to construct a callgraph for each individual file. The figure confirms that larger files require more time to process, as they tend to contain more functions, resulting in larger and more complex callgraphs.

We report the peak RAM usage in Table 6. Consistent with the runtime results, P1 and P2 use a small amount of RAM, around 180MB. In contrast, P3 requires significantly more memory, ranging from 3 to 5GB. Despite this higher memory requirement, P3's RAM usage remains within a reasonable

TABLE 4. Accuracy of QED evaluated on real-world dataset, where n/a means the analysis in a specific phase was not conducted since QED completed the analysis in the earlier phase. tpm2-tools is excluded since we cannot determine their ground truth reliably.

Phases (→)	P1		P1 +P2		P1 +P2 +P3	
Set (↓)	TP/FN (TPR)	TN/FP (TNR)	TP/FN (TPR)	TN/FP (TNR)	TP/FN (TPR)	TN/FP (TNR)
Coreutils	0/0 (100%)	109/0 (100%)	n/a	n/a	n/a	n/a
UnixBench	0/0 (100%)	18/0 (100%)	n/a	n/a	n/a	n/a
Network	7/0 (100%)	4/2 (67%)	7/0 (100%)	6/0 (100%)	6/1 (86%)	6/0 (100%)

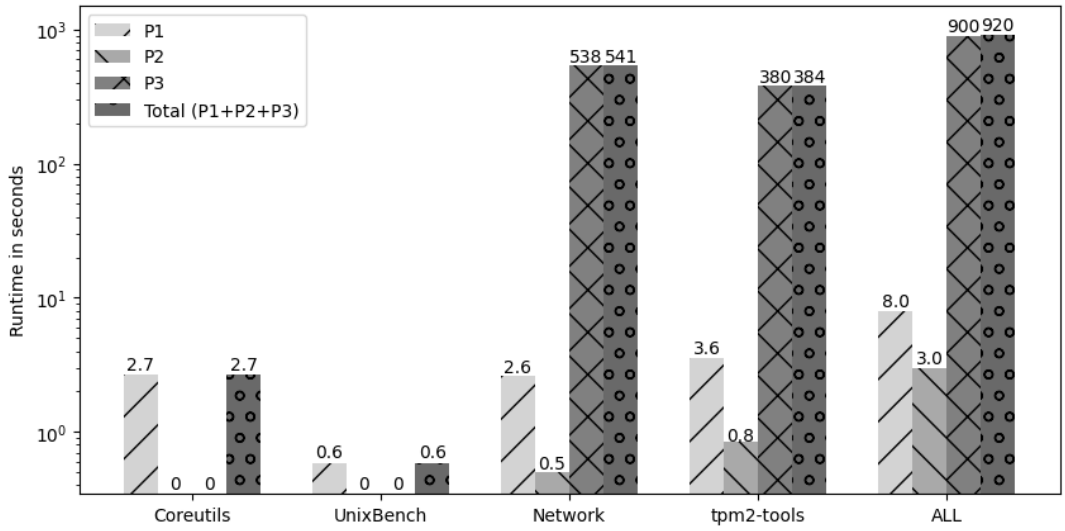

FIGURE 4. Runtime to complete each phase of QED on real-world dataset, where "ALL" contains all sets, i.e., Coreutils+UnixBench+Network+tpm2-tools.

TABLE 5. Callgraph construction (C.C.) time for the tpm2-tools and network set.

Set	tpm2-tools		Network	
	Execs	Deps	Execs	Deps
# files analyzed	86	3	7	8
Avg. size (in KB)	54	543	1574	596
Avg. C.C. Time (in sec)	2.7	38	34	37
Total C.C. Time (in sec)	232	114	238	296

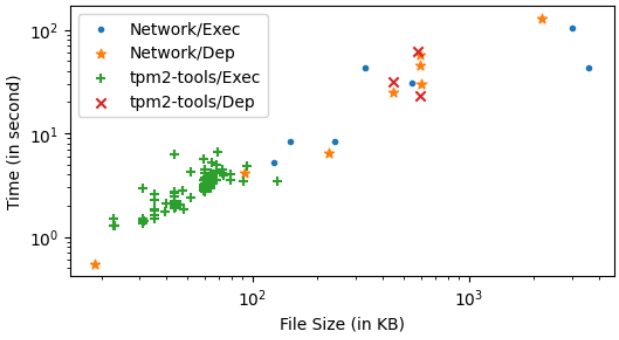

FIGURE 5. Time to construct callgraph of each executable (Exec) and dependency (Dep) in network and tpm2-tools.

range that can easily be acquired in standard commodity computers.

TABLE 6. Peak RAM usage (in MB) of QED on real-world dataset.

Phase	P1	P1 +P2	P1 +P2 +P3
Coreutils	179	n/a	n/a
UnixBench	179	n/a	n/a
Network	181	182	4,958
tpm2-tools	183	187	3,561
All	184	190	6,791

3) MANUAL WORKLOAD REDUCTION

Lastly, we quantify the reduction in manual workload as the number of files requiring further manual analysis to identify their QV status. We note that the normal P3 produces EV_3 evidence, which includes static QV execution traces. While this necessitates some manual checking to confirm its presence in the binaries, it is negligible compared to the effort needed to manually analyze EV_1 or EV_2, which involve reverse engineering binaries or source code (if available). Therefore, we exclude the normal P3 from this evaluation.

Instead, we consider QED with the conservative version of P3, which guarantees no false negatives by falling back to EV_2 evidence for executables where no static QV execution trace is found. Unlike the normal P3, the analyst must manually review this EV_2 for its QV API usage.

TABLE 7. Manual effort reduction of QED on real-world dataset; note that P3 is used in the conservative mode; the real-world dataset contains a total of 226 executables.

Phase	P1	P1 + P2	P1 + P2 + P3
# execs required further manual review	95	93	20
% of manual workload reduction	57.96%	58.84%	91.15%

Table 7 illustrates the manual workload reduction achieved by QED. With just P1 and P2, QED can already reduce the analyst's manual workload by over 50%. Incorporating the conservative version of P3 further reduces the manual effort, leaving only 20 possible QV executables ($< 10\%$) for manual inspection.

V. MEETING THE REQUIREMENTS

We argue that QED satisfies all requirements in Section III-B via the descriptions of QED's design in Section III-C and experimental results in Section IV as follows:

- **R1** **Dynamic Linking.** QED performs file-dependency analysis in P1 to determine all dependencies of the executables, including shared cryptography libraries. This information is subsequently used in P2 and P3 to accurately detect whether these executables not only dynamically link with the cryptography library but also directly or indirectly invoke its QV APIs.
- **R2** **Binary-level Analysis.** All phases in QED operate on the binary executable level and do not require the corresponding source code at any point during its analyses.
- **R3** **Static-only Features.** P1 and P2 require extracting dynamic symbols from the executables, while P3 necessitates constructing static call graphs. All of these features are static and can be found in the executables without the need to run them.
- **R4** **Scalability.** Our experimental results demonstrate that QED completes its analyses on 226 executables in under 10 minutes, averaging about 4 seconds per executable, which is practically reasonable.
- **R5** **Effectiveness.** QED is designed to minimize the number of false negatives, as supported by our experimental results: out of all tested executables, only one is classified as a false negative by P3, while P1 and P2 incur no false negatives, albeit with higher false positive rates. We emphasize that if avoiding false negatives is an important concern, the conservative version of P3 can be employed, though this would increase the analyst's manual workload to identify and eliminate the resulting false positives.

VI. CONCLUSION AND FUTURE WORK

We identified the requirements for a tool to assist in migrating software executables towards post-quantum cryptography. Based on these requirements, we developed a toolchain called QED that detects quantum-vulnerable executables. It operates in three phases, each employing increasingly sophisticated techniques. Its design rationale is to quickly locate and eliminate quantum-safe software using fast analyses in the earlier phases while applying more precise but resource-intensive methods in the final phase. We evaluated QED on both synthetic and real-world datasets, demonstrating its effectiveness, scalability, and ability to reduce the manual workload required for analysts to migrate their organizations toward post-quantum cryptography.

QED has some limitations that suggest avenues for future research. First, as a static analysis tool, QED cannot detect quantum-vulnerable API usages arising from indirect calls (i.e., function pointers), leading to potential false negatives. A possible future direction is to incorporate lightweight dynamic analysis to identify these indirect calls and hence eliminate such false negatives. Second, QED currently assumes executables are dynamically linked with quantum-vulnerable cryptography libraries. As a result, it does not account for scenarios where executables implement cryptographic functions within themselves or statically link with cryptography libraries. Extending QED to detect cryptographic functions directly within executables could address this limitation. Lastly, QED could be expanded beyond merely identifying quantum-vulnerable usages to also (semi-)automatically patch them with quantum-safe and crypto-agile implementations.

APPENDIX A
DATA AVAILABILITY

The source code of the proposed QED is accessible through the GitHub repository at:
https://github.com/norrathep/qed.git.

REFERENCES

[1] I. L. Chuang, N. Gershenfeld, and M. G. Kubinec, "Experimental implementation of fast quantum searching," *Phys. Rev. Lett.*, vol. 80, no. 15, pp. 3408–3411, Apr. 1998.

[2] M. AbuGhanem, "IBM quantum computers: Evolution, performance, and future directions," 2024, *arXiv:2410.00916*.

[3] L. Zhang, A. Miranskyy, W. Rjaibi, G. Stager, M. Gray, and J. Peck, "Making existing software quantum safe: A case study on IBM Db2," *Inf. Softw. Technol.*, vol. 161, Jan. 2021, Art. no. 107249.

[4] F. Neukart, G. Compostella, C. Seidel, D. V. Dollen, S. Yarkoni, and B. Parney, "Traffic flow optimization using a quantum annealer," *Frontiers ICT*, vol. 4, p. 29, Aug. 2017.

[5] M. Schuld, I. Sinayskiy, and F. Petruccione, "An introduction to quantum machine learning," *Contemp. Phys.*, vol. 56, no. 2, pp. 172–185, Oct. 2014.

[6] Y. Cao, J. Romero, and A. Aspuru-Guzik, "Potential of quantum computing for drug discovery," *IBM J. Res. Develop.*, vol. 62, no. 6, pp. 6:1–6:20, Nov. 2018.

[7] T. Häner, M. Roetteler, and K. M. Svore, "Factoring using 2n+ 2 qubits with Toffoli based modular multiplication," *Quantum Inf. Comput.*, vol. 17, no. 7, pp. 673–684, 2017.

[8] M. Roetteler, M. Naehrig, K. M. Svore, and K. Lauter, "Quantum resource estimates for computing elliptic curve discrete logarithms," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, Jan. 2017, pp. 241–270.
[9] D. Ott and C. Peikert, "Identifying research challenges in post quantum cryptography migration and cryptographic agility," 2019, *arXiv:1909.07353*.
[10] D. Joseph, R. Misoczki, M. Manzano, J. Tricot, F. D. Pinuaga, O. Lacombe, S. Leichenauer, J. D. Hidary, P. Venables, and R. Hansen, "Transitioning organizations to post-quantum cryptography," *Nature*, vol. 605, no. 7909, pp. 237–243, May 2022.
[11] National Institute of Standards and Technology, "Migration to post-quantum cryptography quantum readiness: Cryptographic discovery," *NIST SPECIAL PUBLICATION 1800-38B*, 2023.
[12] A. Pandey, A. Banati, B. Rajendran, S. D. Sudarsan, and K. K. S. Pandian, "Cryptographic challenges and security in post quantum cryptography migration: A prospective approach," in *Proc. Int. Conf. Public Key Infrastructure Appl. (PKIA)*, Sep. 2023, pp. 1–8.
[13] L. Benedetti, A. Thierry, and J. Francq, "Detection of cryptographic algorithms with grap," *Cryptol. ePrint Arch.*, vol. 2017, p. 1119, Jan. 2017.
[14] P. Lestringant, F. Guihéry, and P.-A. Fouque, "Automated identification of cryptographic primitives in binary code with data flow graph isomorphism," in *Proc. ACM Symp. Inf., Comput. Commun. Secur.*, Apr. 2015, pp. 203–214.
[15] N. Lutz, "Towards revealing attacker's intent by automatically decrypting network traffic," M.S. thesis, Mémoire de maıtrise, ETH, Zürich, Switzerland, 2008.
[16] D. Xu, J. Ming, and D. Wu, "Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2017, pp. 921–937.
[17] F. Gröbert, C. Willems, and T. Holz, "Automated identification of cryptographic primitives in binary programs," in *Proc. Int. Symp. Recent Adv. Intrusion Detection*, Jan. 2011, pp. 41–60.
[18] R. Zhao, D. Gu, J. Li, and R. X. Yu, "Detection and analysis of cryptographic data inside software," in *Proc. Int. Conf. Inf. Secur.*, Jan. 2011, pp. 182–196.
[19] K. Eldefrawy, M. E. Locasto, N. Rattanavipanon, and H. Saidi, "Towards automated augmentation and instrumentation of legacy cryptographic executables," in *Proc. Int. Conf. Appl. Cryptogr. Netw. Secur.*, Jan. 2020, pp. 364–384.
[20] N. Weideman, H. Wang, T. Kann, S. Zahabizadeh, W.-C. Wu, R. Tandon, J. Mirković, and C. Hauser, "Harm-DoS: Hash algorithm replacement for mitigating denial-of-service vulnerabilities in binary executables," in *Proc. Int. Symp. Res. Attacks, Intrusions Defenses*, Oct. 2022, pp. 276–291.
[21] C. Meijer, V. Moonsamy, and J. Wetzels, "Where's crypto: Automated identification and classification of proprietary cryptographic primitives in binary code," in *Proc. USENIX Secur. Symp.*, 2021, pp. 555–572.
[22] L. Chen, L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. A. Perlner, and D. Smith-Tone, "Report on post-quantum cryptography," Nat. Inst. Standards Technol., U.S. Dept. Commerce, USA, Tech. Rep. NISTIR 8105, 2016.
[23] D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe, "The SPHINCS+ signature framework," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2019, pp. 2129–2146.
[24] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehle, "CRYSTALS–kyber: A CCA-secure module-lattice-based KEM," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, Apr. 2018, pp. 353–367.
[25] R. Overbeck and N. Sendrier, "Code-based cryptography," in *Post-Quantum Cryptography*. Cham, Switzerland: Springer, 2009, pp. 95–145.
[26] J. Ding and A. Petzoldt, "Current state of multivariate cryptography," *IEEE Secur. Privacy*, vol. 15, no. 4, pp. 28–36, Aug. 2017.
[27] National Institute of Standards and Technology, "Module-lattice-based key-encapsulation mechanism standard," Tech. Rep. FIPS 203, 2024.
[28] National Institute of Standards and Technology, "Module-lattice-based digital signature standard," Tech. Rep. FIPS 204, 2024.
[29] National Institute of Standards and Technology, "Stateless hash-based digital signature standard," Tech. Rep. FIPS 205, 2024.
[30] D.-T. Dam, T.-H. Tran, V. Hoang, C. Pham, and T.-T. Hoang, "A survey of post-quantum cryptography: Start of a new race," *Cryptography*, vol. 7, no. 3, p. 40, Aug. 2023.
[31] B. LaMacchia, "The long road ahead to transition to post-quantum cryptography," *Commun. ACM*, vol. 65, no. 1, pp. 28–30, Dec. 2021.
[32] C. Näther, D. Herzinger, S.-L. Gazdag, J.-P. Steghöfer, S. Daum, and D. Loebenberger, "Migrating software systems towards post-quantum-cryptography—A systematic literature review," 2024, *arXiv:2404.12854*.
[33] K. F. Hasan, L. Simpson, M. A. R. Baee, C. Islam, Z. Rahman, W. M. Armstrong, P. Gauravaram, and M. McKague, "A framework for migrating to post-quantum cryptography: Security dependency analysis and case studies," *IEEE Access*, vol. 12, pp. 23427–23450, 2024.
[34] N. Alnahawi, N. Schmitt, A. Wiesmaier, A. Heinemann, and T. Grasmeyer, "On the state of crypto-agility," *Cryptol. ePrint Arch.*, pp. 1–19, Apr. 2023. [Online]. Available: https://eprint.iacr.org/2023/487
[35] C. Collberg, J. H. Hartman, S. Babu, and S. K. Udupa, "Slinky: Static linking reloaded," in *Proc. USENIX Annu. Tech. Conf.*, Apr. 2005, p. 34.
[36] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SOK: (State of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 138–157.

NORRATHEP RATTANAVIPANON received the Ph.D. degree in computer science from the University of California, Irvine, in 2019. Currently, he is an Assistant Professor with the College of Computing, Prince of Songkla University, Phuket Campus. His research interests include security and privacy, particularly in embedded systems and the IoT security, software and binary analysis, and security/privacy in machine learning systems.

JAKAPAN SUABOOT received the B.Eng. and M.Eng. (research) degrees in computer engineering from Prince of Songkla University, Hatyai Campus, Thailand, in 2007 and 2010, respectively, and the Ph.D. degree from RMIT University, Australia, in 2021. He is currently a Lecturer with the College of Computing, Prince of Songkla University. His research interests include malware detection, data breach prevention, machine learning technologies, and DeFi security.

WARODOM WERAPUN received the Ph.D. degree in computer engineering from ENSEEIHT/INP University, France, in 2012. He is currently an Assistant Professor, the Head of the BLOCK Research Team, and the Associate Dean for Research and Innovation of the College of Computing, Prince of Songkla University, Phuket Campus. He has received four awards in blockchain-related projects at national and international levels. His research interests include blockchain, smart contracts, web programming, and cybersecurity.