Quantum secured Ethereum Client – PHASE I – using QRNG to provide better randomness of the protocol

Summary

In this communication we describe the first phase of our work to build quantum sequred version of Ethereum client. We have addressed two key operations that need better entropy sources than provided by pseudo-random generators.

This phase assumes the use of Quantum Random Number Generator for the generation of entropy for the client function of private and public keys as well as the adress generation. In the work reported here we have used our QRNGBase service providing Entropy-as-a-Service via REST based, SLL protected API, and a two open-source Ethereum clients. We have built a new code into one of the clients code that uses QRNG to generate entropy.

Basic info about clients

We have considered two Ethereum Clients: EthereumJS Monorepo client[1] built by Ethereum Javascript Community[2] and Ethereum Besu client[3] . The Besu client is one of the most excellent implementation of Ethereum protocol. It is an open-source client developed under the Apache 2.0 license and written in Java, well suited to high-performance transaction processing in private networks.

We have identified the use of randomness in both clients in the implementation of RLPx Transport Protocol. For example, the EthereumJS Monorepp RPLx code uses ECIES (Elliptic Curve Integrated Encryption Scheme) asymmetric encryption. One of the key elements of ECIES security is the generation of elliptic curve public key and the computation of MAC (Message Authentication Code).

The Besu Ethereum client uses the same protocol hence its use of randomness in ECIES Encryption is the same.

We have also identified that Besu has an interesting functionality allowing for the generation of keys and addresses to be used on Ethereum.

We have successfully installed and run small networks of both clients. However, as the Besu client is much more advanced, matured and stable code, and its use of randomness is more serious, we have implemented QRNG only to the Besu client. Depending on the possible future choices of our development team we can use both clients.

[1] https://github.com/ethereumjs/ethereumjs-monorepo , documentation at: https://ethereumjs.readthedocs.io/
[2] https://github.com/ethereumjs
[3] https://besu.hyperledger.org/

Protecting key generation, RLPx protocol and enhancing the consensus algorithm of the Besu client.

We have modified parts of the Besu client source code. Essentially, we have provided SecureRandom class extension by our new SecureRandomQuantum class:

```
public class SecureRandomQuantum extends SecureRandom {

  private static final String URI =
      "https://qrng.qbck.io/ac50674e-0725-42c5-9aa3-f4de0a2b390c/qbck/block/";
  private static final String TYPE_BIN = "bin";
  private static final String TYPE_INT = "int";
  private static final String TYPE_LONG = "long";
  private static final String TYPE_FLOAT = "float";
  private static final int MIN = 0;
  public SecureRandomQuantum() {
    super();
  }

  @Override
  public int nextInt() {
    return nextInt(Integer.MAX_VALUE);
  }

  @Override
  public int nextInt(final int bound) {
    try {
      String answer = getRandom(genereteUriNumericRange(TYPE_INT, 1, MIN,
Integer.toString(bound)));
      System.out.println("RANDOM INT: " + Integer.parseInt(answer));
      return Integer.parseInt(answer);
    } catch (URISyntaxException | InterruptedException | IOException e) {
      System.out.println("ERROR QUANTUM: " + e.getMessage());
      return super.nextInt();
    }
  }

  @Override
  public void nextBytes(final byte[] bytes) {

    StringBuilder buildRequest = new StringBuilder();
    buildRequest.append(URI);
    buildRequest.append(TYPE_BIN);
    buildRequest.append('?');
    buildRequest.append("size=");
    buildRequest.append(1);
    buildRequest.append("&length=");
    buildRequest.append(bytes.length);

    try {
      String answer = getRandom(buildRequest.toString());

      byte[] tmp = new BigInteger(answer, 2).toByteArray();

      System.arraycopy(tmp, 0, bytes, 0, bytes.length);
      System.out.println("FINAL BYTES: " + Arrays.toString(bytes));
    } catch (URISyntaxException | IOException | InterruptedException e) {
      System.out.println("ERROR QUANTUM: " + e.getMessage());
      super.nextBytes(bytes);
    }
  }

  @Override
  public long nextLong() {
    try {
      String answer =
          getRandom(genereteUriNumericRange(TYPE_LONG, 1, MIN,
Long.toString(Long.MAX_VALUE)));
      System.out.println("RANDOM LONG: " + Long.parseLong(answer));
      return Long.parseLong(answer);
    } catch (URISyntaxException | InterruptedException | IOException e) {
      System.out.println("ERROR QUANTUM: " + e.getMessage());
```

```
      return super.nextLong();
    }
  }

  @Override
  public float nextFloat() {
    try {
      String answer =
          getRandom(genereteUriNumericRange(TYPE_FLOAT, 1, MIN,
Float.toString(Float.MAX_VALUE)));
      System.out.println("RANDOM FLOAT: " + Float.parseFloat(answer));
      return Float.parseFloat(answer);
    } catch (URISyntaxException | InterruptedException | IOException e) {
      System.out.println("ERROR QUANTUM: " + e.getMessage());
      return super.nextFloat();
    }
  }
  private String getRandom(final String uriString)
      throws URISyntaxException, IOException, InterruptedException {
    URI uri = new URI(uriString);
    System.out.println("QUANTUM - " + "send request: " + uri);
    HttpRequest request = HttpRequest.newBuilder(uri).GET().build();

    HttpResponse<String> response =
        HttpClient.newBuilder()
            .sslContext(insecure_Context())
            .build()
            .send(request, HttpResponse.BodyHandlers.ofString());

    JSONObject obj = new JSONObject(response.body());
    JSONObject obj1 = obj.getJSONObject("data");
    JSONArray array = obj1.getJSONArray("result");
    return array.get(0).toString();
  }

  static SSLContext insecure_Context() {
    TrustManager[] noopTrustManager =
        new TrustManager[] {
          new X509TrustManager() {
            @Override
            public void checkClientTrusted(final X509Certificate[] xcs, final String string)
{}

            @Override
            public void checkServerTrusted(final X509Certificate[] xcs, final String string)
{}

            @Override
            public X509Certificate[] getAcceptedIssuers() {
              return null;
            }
          }
        };
    try {
      SSLContext sc = SSLContext.getInstance("ssl");
      sc.init(null, noopTrustManager, null);
      return sc;
    } catch (KeyManagementException | NoSuchAlgorithmException ex) {
      System.out.println(ex.getMessage());
    }
    return null;
  }
  private String genereteUriNumericRange(
      final String type, final int size, final int min, final String max) {
    StringBuilder buildRequest = new StringBuilder();
    buildRequest.append(URI);
    buildRequest.append(type);
    buildRequest.append('?');
    buildRequest.append("size=");
    buildRequest.append(size);
    buildRequest.append("&min=");
    buildRequest.append(min);
    buildRequest.append("&max=");
    buildRequest.append(max);
    return buildRequest.toString();
  }
}
```

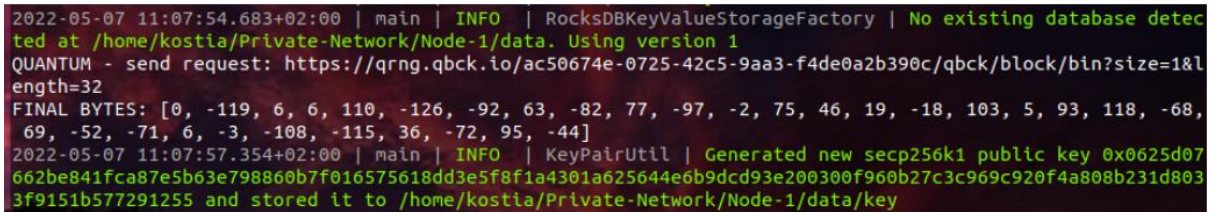
The results

The code modification resulted in the use of Quantum Random Number Generator for:

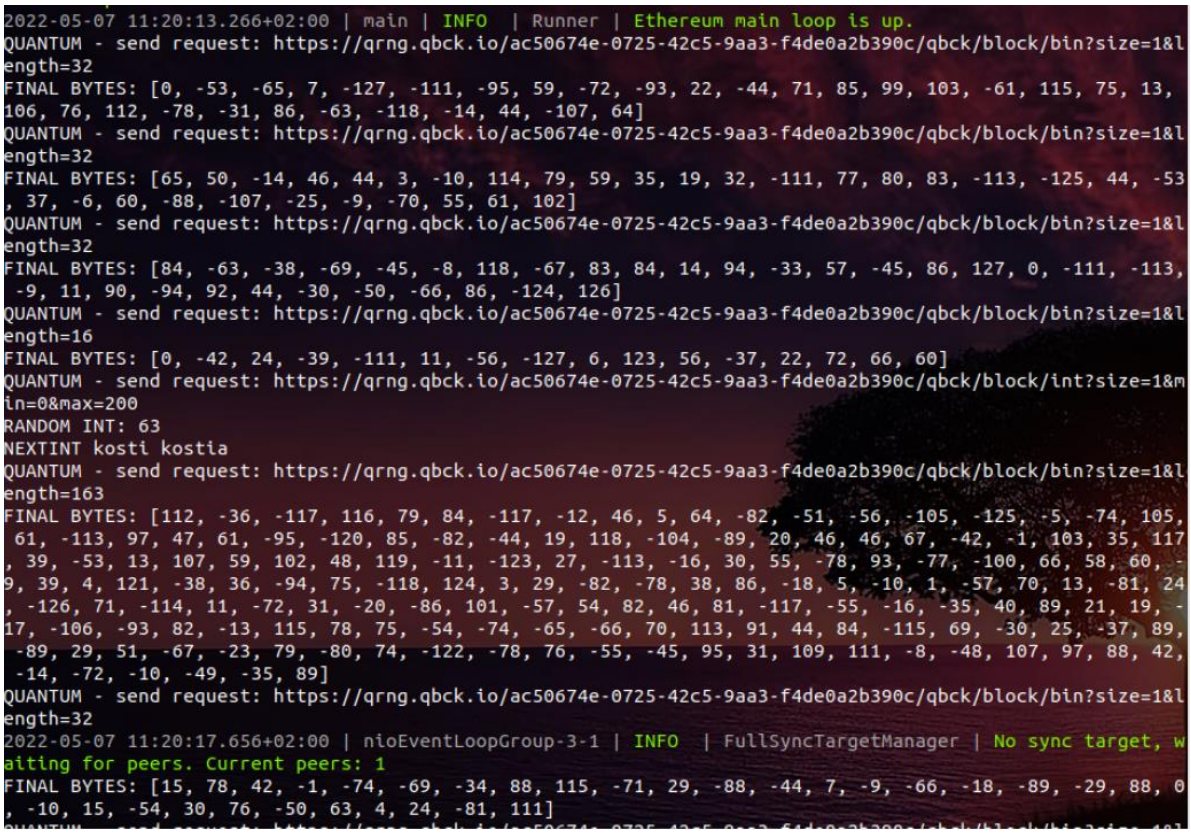
- Better entropy generation for creation of private and public keys by the client, in the case when users do not use keys provided by their wallets.
- Better and more secured randomness used in messages authentication codes inside the encrypted handshake protocol
- Better randomness for the Proof-of-Work consensus mechanism (during “nonce” generation):

We provided the client generated output illustration the calls to the QRNG:

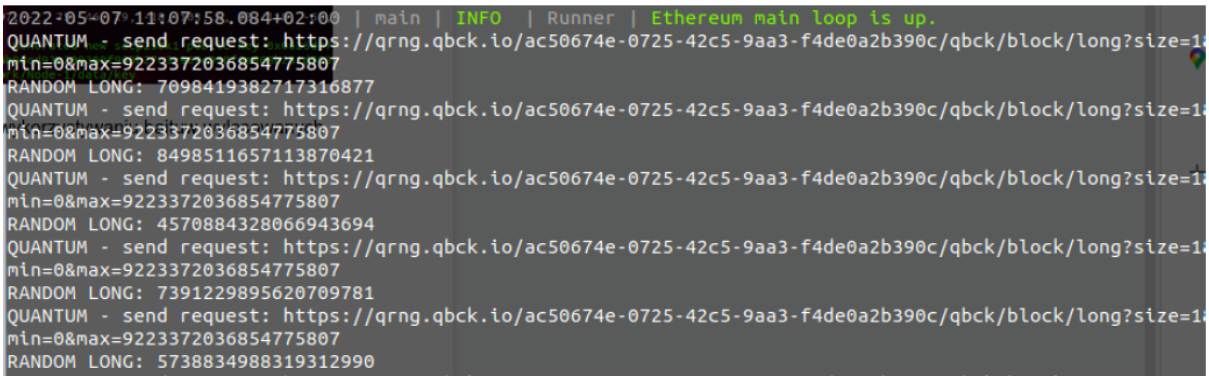
- Quantum generated entropy is used by the key generation routines of the Besu Ethereum client:



- Quantum randomness used in the encrypted handshake of RLPx protocol:



- Quantum randomness for the PoW consensus (“nonce” generation for blocks creation):



A short recording shows the console output of the modified client on an experimental three-node private Ethereum network:

https://youtu.be/KpSxUwH3SdY

The conclusion

We have demonstrated the modification of Besu Ethereum client leading to the use of Quantum Random Number Generators for augmented entropy generation and unpredictability which ultimately makes the resulting Ethereum client much safer.

The work reported here will be used as a first step of further and deeper modification of the Ethereum client.