Research Internship Report

Analysis of a Comparison Tool for Searchable Encryption Techniques for Order Queries

by

Michaël P. van de Weerd

Wednesday 14th August, 2019

Abstract

Several techniques exists that allow order queries to be executed on ciphertexts. In order to give insight into the properties of two of these techniques — Order-Revealing Encryption (ORE) and Garbled Circuits (GC) — this research aims to compare them in terms of performance. To this end, a tool has been developed that can measure the performance of each technique. The tool features a simulation of a client and a server, in which the client sends order queries to the server, which stores the encrypted data. Furthermore, a small collection of tests is available to validate the functionality of both techniques and the specialized data structure — *treaps* — being used to store the ciphertexts. One of the outcomes of this research is an outline of future research that needs to conducted in order for the experimentation to be completed.

Contents

Acronyms

Chapter 1

Introduction

Security is one of the greater challenges of the IT industry. Encryption is a great tool to secure data, either by encrypting complete systems or individual data objects. When applying the latter technique to data stored in databases, data is encrypted by a client, sent to — and stored by — a server, only to be decrypted by the client. A problem with this approach is that queries to the server by the client will be applied to the encrypted data instead of the underlying plaintext. This means that the client will be unable to request specific data. Several techniques exist to solve this issue, by allowing *order queries* to be applied to encrypted data. Two of these methods will be the main focus of this report, ORE and GC, allowing the use of earlier research.

This chapter introduces the problems to be solved by the research and the approach used to find a solution. Furthermore, the works related to the subject of this research are described in section 1.2.

1.1 Problem Definition

This project will result in the following deliverables:

Code In order to perform the experiments, the provided code base might require alteration, extension or improvement. Therefore, the code base will be "returned" at the end of the internship to ensure reproducibility of the experiments.

Report The current report is an account of the various activities undertaken during the internship, but also provide documentation of the code and a concise description of various theoretical concepts in the form of a literature review. This knowledge is key to understanding the intricate workings of the code base.

Results To support further work on this subject, data will be collected during experiments with the code-base.

In order to compare the two encryption techniques for order queries, a research tool will be utilized. This tool will provide insight into the properties of both techniques in terms of performance. The tool is based on earlier work by Grim and Wiersma [2017]. However, the functionality of this tool is incomplete, and requires analysis and further implementation. Therefore, the tool's software will be investigated by using testing methods that are appropriate to the used techniques. The results from these tests will be used to specify which parts of the software require further implementation. Finally, after having implemented the missing or incomplete functionality, the experiments can be executed.

1.2 Related Works

Earlier work on a similar projects has resulted in the master's thesis *A Secure Roundtrip Index for Range Queries* by Tobias Boelter, Rishabh Poddar and Raluca Ada Popa. Here, the authors extend the GC scheme, which the current project aims to analyze. This work will be the main reference for understanding the concept of GC and the extended scheme.

In the initial version of the code, a library produced by Kevin Lewi and David J. Wu is used to realize ORE. To ensure correspondence between this library and the understanding of ORE, their accompanying work *Practical Order-Revealing Encryption with Limited Leakage* will be consulted for definitions and theory.

During the implementation of code for the product, the handbook *Clean Code* by Robert C. Martin[1] will be consulted for best practices and conventions to ensure readability and maintainability. This is crucial for a project that will be handed over between different researchers that have different levels of understanding of the subject at hand. Furthermore, the Python-specific code conventions defined by Guido van Rossum in PEP 8[2] will be applied to the Python code.

[1] ISBN 978-0-13-235088-4

[2] `https://www.python.org/dev/peps/pep-0008/`

Chapter 2

Background

This chapter will provide a review of the knowledge that is fundamental to the subject of *order queries*. The subjects covered are that of *treaps*, OREs and GCs, the prior of which is dealt with in section 2.1. In section 2.2 the subjects of the cryptographic techniques ORE and GC are explored.

2.1 Treaps and Related Data Structures

Whereas the *binary search tree* and *binary heap* data structures provide distinct characteristics, *treaps* combine the capabilities of both, hence its name being a portmanteau of “tree” and “heap”. In this section, the specifics of both binary search trees and binary heaps will be recapped, allowing for a smooth transition to the definition of treaps. In the literature, one may find a concept closely related to treaps called *randomized binary search trees*, which will not be covered in this report. Both the binary search tree and the binary heap are extensions upon the concept of a binary tree. For completeness sake, the definition of binary trees is also covered in this section.

2.1.1 Binary Trees

Liang [2013] shows us that the binary tree is a data structure, consisting of *nodes*, each of which optionally has a *left child* or *right child*, which are nodes as well. A node that is not a child of any other node in the tree is called the *root*, while a node having neither a left nor right child is called a *leaf*. Nodes that are a child of the same node — i.e. the *parent* — are called *siblings*. The length of the path between a node and the root is referred to as *depth*, allowing for sets of nodes having the same depth to be defined as a *level*. We can construct subtrees by considering a node in the tree to be a root. Several of these concepts have been illustrated in Figure 2.1.

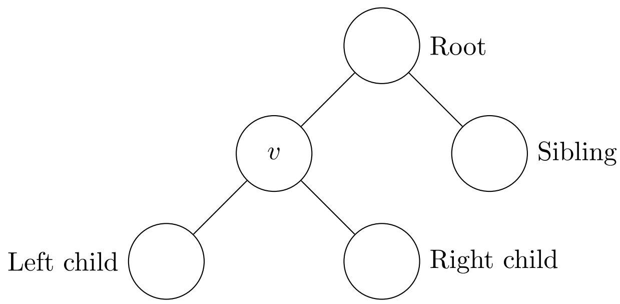


Figure 2.1: An illustration of several definition related to binary trees. Each label refers to the relation of the node in question to node v.

2.1.2 Binary Search Trees

According to Liang [2013], what separates the Binary Search Tree (BST) from an "ordinary" binary tree is that, for every node v in a binary tree, having a left child and a right child being the roots of sub-trees A and B respectively:

$$\forall a \in A, \forall b \in B : a < v < b.$$

Intuitively, this means that every node in a BST will have nodes of a *lower* value on its left and nodes of a *higher* value on its right when traversing down the tree structure. As a result, visiting the nodes in a BST using *in-order traversal* will result in a list of nodes ordered by increasing value. This concept is illustrated in Figure 2.2.

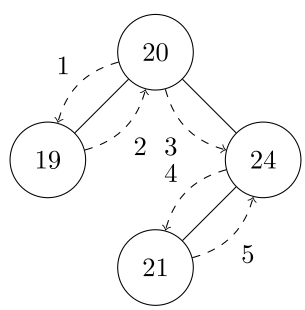


Figure 2.2: An example of a BST. Note that for each node, the left child subtree only contains nodes with a lower value, while the right child subtree only contains nodes with a higher value. The dashed arrows indicate the path that emerges when visiting each node using in-order traversal, resulting in the ordered listing of the node values: 19, 20, 21, 24.

2.1.3 Binary Heaps

A binary heap is a binary tree that meets the *heap property*, as explained by Liang [2013]. This property states that for each node v in a heap, having a left child and a right child a and b respectively:

$$\max(a, b) \leq v,$$

or, depending on the application:

$$\min(a, b) \geq v.$$

Intuitively this means that every node has a value greater or equal to its left child and right child. The nodes of a binary heap can be stored in an array, where the left and right child of a node at position i can be found at position $2i + 1$ and position $2i + 2$ respectively. An illustration of this concept has been included in Figure 2.3.

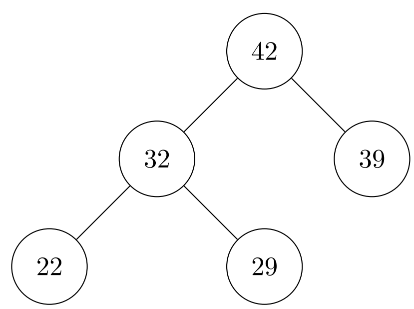


Figure 2.3: An example of a binary heap. Note that the left child and right child of each node has a smaller value than the node itself, i.e. $\max(a, b) \leq v$. The nodes in this heap can be stored as the array $[42, 32, 39, 22, 29]$.

2.1.4 Treaps

As mentioned before, treaps combine the features of BSTs and binary heaps. Concretely, this means that a treap has the property of order identical to that of the BST and the heap property. This is achieved by assigning *two* values to each node, a *key* and a *priority*. In the context of treaps, we can define a node as a tuple containing a key and priority a follows: (k, p). Given a treap having a node (v_k, v_p) having a left child (α_k, α_p) and a right child (β_k, β_p), both of which are roots of the sub-trees A and B respectively, then the following proposition must be true:

$$(\forall (a_k, a_p) \in A, \forall (b_k, b_p) \in B : a_k < v_k < b_k) \wedge (\max(\alpha_p, \beta_p) \leq v_p).$$

Intuitively this means that a tree constructed from the keys of each node in a treap must be a valid BST and a tree constructed from the priorities of each node in a treap must be a valid binary heap. An example of a treap is provided in Figure 2.4.

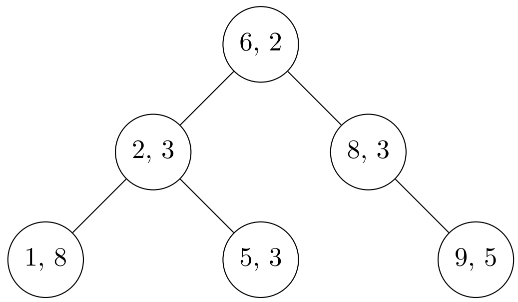


Figure 2.4: An example of a treap. Note that the key — the left part of each label — maintains the ordering property of a BST while the priority — the right part of each label — preserves the heap property, which is defined as $\min(a, b) \geq v$.

2.2 Cryptographic Techniques

This section describes the workings of the cryptographic techniques that are considered in this project. In general, these techniques are used to convert *plaintext* into *ciphertext* while still allowing to perform certain computations over the latter as if it where the prior. The goal of cryptography is to reduce the possibilities of reversing this process without the required prior knowledge, i.e. a key.

2.2.1 Order-Revealing Encryption

In order to gain a solid understanding of the concept of ORE, it is useful to be familiar with the related concept of Order-Preserving Encryption (OPE). Lewi and Wu [2016] describe an OPE as an encryption scheme that allows for the comparison of encrypted values. Being able to compare values means knowing the order of values. This is achieved by having the comparison operation result in an indication of the difference of one value to another, e.g. -1, 0 or 1 if value a is respectively smaller than, equal to or greater than value b. Lewi and Wu [2016] explain that OPE is a so-called Property-Preserving Encryption (PPE), an encryption scheme that allows for ciphertexts to reveal a specific property of the underlying plaintext. Unfortunately, research referenced by Lewi and Wu [2016] has shown that this mechanism within OPE also results in a significant amount of information leaks about the encrypted plaintexts.

The issue of information leakage in OPE is resolved in the definition of ORE, provided in Lewi and Wu [2016]. They state that, in contrast to OPE, ORE does not impose any constraints on ciphertexts — e.g. ordered numeric values — but requires that a comparison function is provided that is able to compute comparisons between ciphertexts. The ORE scheme consists of three algorithms: *ORE.Setup*, *ORE.Encrypt* and *ORE.Compare*. The (simplified) properties of the algorithms as described in Lewi and Wu [2016] and Chenette et al. [2015] are:

ORE.Setup() $\rightarrow k$ The algorithms returns a secret key k.

ORE.Encrypt$(k, m) \rightarrow c$ Given a secret key k and a plaintext input message m, the algorithms returns a ciphertext c.

ORE.Compare$(c_1, c_2) \rightarrow b$ Given two ciphertexts c_1 and c_2, the algorithm returns a value $b \in \{0, 1\}$.

Note that the scheme does not include a decryption algorithm. Chenette et al. [2015] explain that this is due to the generic nature of the ORE scheme. They argue that this functionality can be implemented using the available algorithms or by extending the encryption algorithm to be Chosen-Plaintext Attack (CPA) secure, meaning a symmetric encryption key is required.

2.2.2 Garbled Circuits

Boelter et al. [2016] describe GCs as a cryptographic technique that "encrypts" logical circuits such that the functionality is preserved. Due to the encrypts, the internal logic of the circuit cannot be evaluated in a meaningful way, allowing for critical information to be obscured. Furthermore, Boelter et al. [2016] provide an overview of the GC scheme, which consist of four algorithms: *GC.Garble*, *GC.Encode*, *GC.Eval* and *GC.Decode*. A graphical representation of the data flow when using a GC has been included in Figure 2.5. Next, the definitions of these algorithms will be presented as they have been given by Boelter et al. [2016]:

GC.Garble$(f) \rightarrow (F, e, d)$ Given a binary circuit f, the algorithm returns a garbled circuit F, encoding information e and decoding information d.

GC.Encode$(e, x) \rightarrow X$ Given encoding information e and plain input x, the algorithms returns a garbled input X, provided that x is a valid input for f.

GC.Eval$(F, X) \rightarrow Y$ Given a garbled circuit F and a garbled input X, the algorithm returns a garbled output Y.

GC.Decode$(d, Y) \rightarrow y$ Given decoding information d and garbled output Y, the algorithm returns plain output y.

2.2.3 Branch-Chained Garbled Circuits

In their master's thesis, Boelter et al. [2016] introduce the notion of *branch-chained garbled circuits*. These constructions are treaps where each node is a GC. When traversing the tree, starting at the root, the output of the GC at each node determines which of its two children will be the next node to visit. Considering the branch-chained GC as a scheme of its own, Boelter et al. [2016] define it to consist of three algorithms: *BCGC.Generate*, *BCGC.Encode* and *BCGC.Eval*. In Figure 2.6 a graphical representation is included of the data streams when using branch-chained GCs. Boelter et al. [2016] provide algorithms for the aforementioned methods, which are included in this report in a simplified form:

BCGC.Generate($\boldsymbol{f, e_1, e_2}$) $\rightarrow$ ($\boldsymbol{F, e}$) Given a boolean circuit f and encoding information e_1 and e_2, the algorithm returns a branch-chained garbled circuit F and encoding information e.

BCGC.Encode($\boldsymbol{e, x}$) $\rightarrow$ $\boldsymbol{X}$ Identical to the encode algorithm for GC schemes as presented in subsection 2.2.2.

BCGC.Eval($\boldsymbol{F, X_1}$) $\rightarrow$ ($\boldsymbol{b, X_2}$) Given a branch-chained garbled circuit F and a garbled input X_1, the algorithm returns a bit b indicating whether the next node will be the left or right child of F and a garbled input X_2 to be used as input for the evaluation of the next garbled circuit.

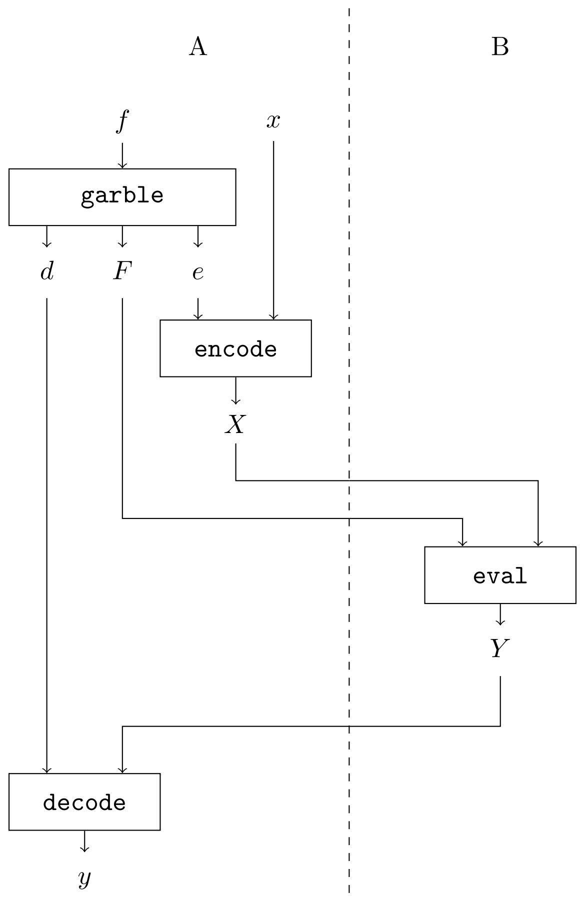


Figure 2.5: Graphical illustration of the data flow when using a GC, demonstrating how client B can be utilized to evaluate binary circuit f with input x without ever knowing about what these objects are due to the garbling and encoding applied by client A.

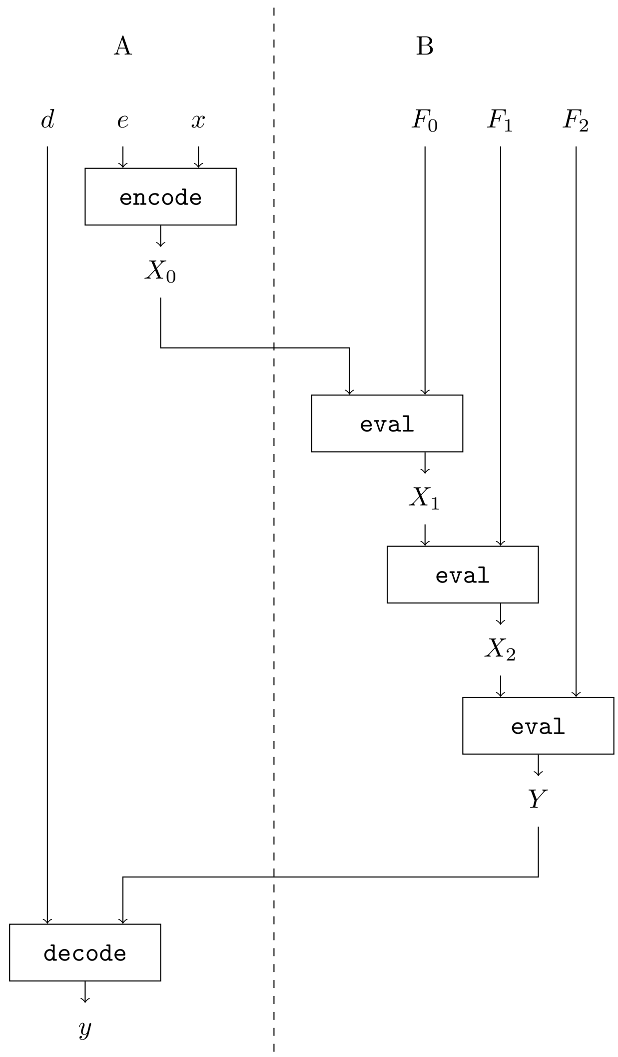


Figure 2.6: Graphical representation of the data flow when using branch-chained GC. Client A has provided client B with a collection of GCs earlier, generated as described in subsection 2.2.3. The output of each evaluation can be used to determine which GC to use next. In the context of this report, client B stores the GCs in a *treap* data structure.

Chapter 3

Activities

In this chapter the various activities are reported in chronological order. As explained in chapter 1 this starts out with analyzing the current state of the software. This analysis — in combination with the theory collected in chapter 2 — leads to an overview of the code that requires to be fixed or implemented.

3.1 Analysis of the Status Quo

In order to get a clear image of the work that still needs to be done, the current state of the software is analyzed. This is done by mapping the architecture, i.e. the components that make up the software and how they are related. Furthermore, the functionality of the software is tested using unit tests. Using these tests it can be determined whether the software meets the requirements that have been defined in chapter 2.

3.1.1 Client–Server Communication

As order queries are particularly useful to client–server communication, the initial code base provides ways to simulate such a use-case. To be able to use this software, it is important to understand how it is implemented, which component does what and how these components are related. The code for the client and server — `client.py` and `server.py` respectively — can be run in order to simulate the communication between a server and client in a real-life use-case. Both components utilize treap nodes to store data and communicate via *XML-RPC*. The process of inserting node into the server's database has been visualized in Figure 3.1. Different configurations can be used to store date either as plain text, ORE or GC. The client can be provided with a number that represents that amount of nodes to be inserted into the treap. Each node contains either plaintext or data encrypted with either ORE or GC, encapsulated in a wrapper class.

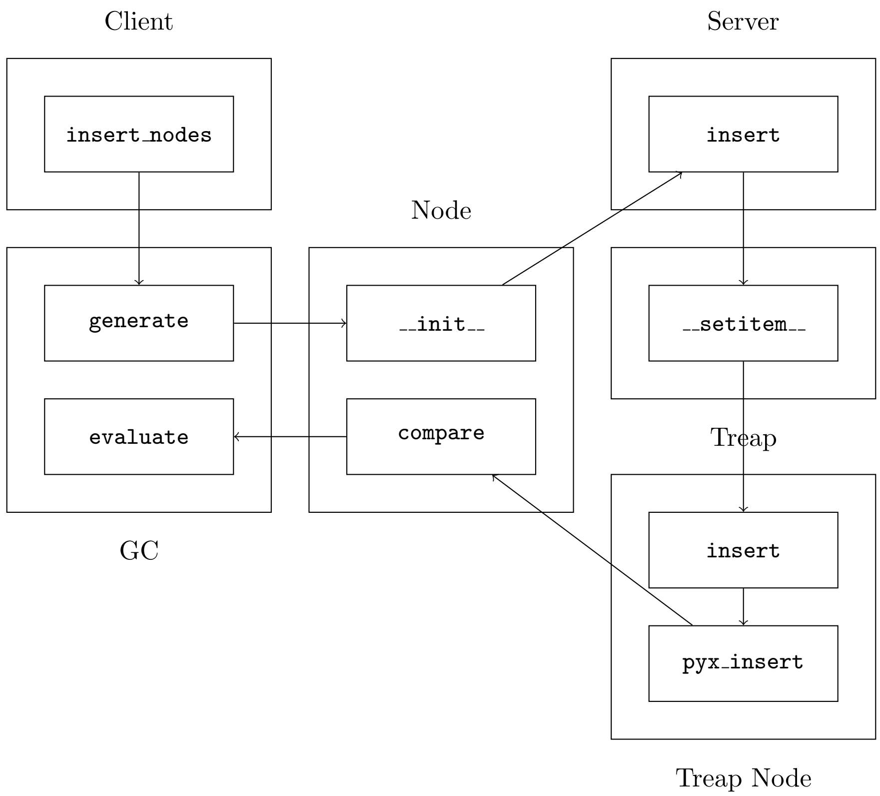


Figure 3.1: Illustration of the function sequence trigged when the client inserts nodes into the database, starting at `insert_nodes` and ending at the `evaluate` method. This visualization assumes that GC is used.

3.1.2 Unit Testing

In order to test whether the functionality of the implementations for the treap, ORE and GC are such as they have been defined in chapter 2, unit tests are implemented. These tests can be executed separately from the main programming and test the behavior of specific components of the software. Four unit test classes have been implemented. All the unit tests are implemented in Python and are included in Appendix B.

Executing the unit test with the initial version of the code results in four errors, all of which occur at tests that are part of `TestGC`. The complete results of these tests are included in section B.5. Upon further inspection of the code the cause is evident, as the implementation for the GC wrapper is incomplete. Therefore, in order to be able to compare the ORE and GC methods, this particular part of the software must be implemented in a next step.

3.2 Implementation

Due to the fact that initial software is not ready to be used for experimenting, several changes and additions have to be made. These changes are documented and their necessity is argued in this section.

3.2.1 GC Wrapper

As concluded in subsection 3.1.2, the initial version of the software lacked a completely implemented wrapper for the GC library. The signature of the initial version of the wrapper's `encode` method must be altered to allow two parameters for plaintext x and encoding information e, as is evident from the GC definition in chapter 2. The completed implementation of the GC wrapper has been included in section C.1 and has been verified to be correct using unit testing.

3.2.2 GC Communication

As a result of enabling actual GCs to be communicated between the client and server, a peculiar problem becomes apparent. Due to the specification of the `XML-RPC`, only 32-bit signed integers are supported, as seen in Winer [1999]. However, GC encoded data of values that exceed 32-bits, resulting in the error message `OverflowError: int exceeds XML-RPC limits`. In order to resolve this issue the decision has been made to convert every integer to a string in favor of implementing the communications using another protocol in order to safe time. Two methods, `pack` and `unpack` have been implemented to add this functionality and are included in section C.2. In order to distinguish between actual strings and *packed* integers, each converted integer is prefixed with a substring `__int__`.

3.2.3 GC Comparing

Taking in use the GC-base client–server communication has lead to the identification of another problem. Namely, the incorrect comparison of GCs, encapsulated in a node. As the code snippet in Listing 3.1 shows, each node relies of the `evaluate` method of the GC wrapper.

Listing 3.1: Code snippet of the `compare` method in the `Node` class.

```
def compare(self, b):
    if self.type == "ORE":
        return ORE.compare(
            self.value[0],
            b.value[1]
        )
    elif self.type == "GC":
        return GC.evaluate(
            self.value,
```

```
            b.value
        )
    elif self.type == "PLAIN":
        return PLAIN.compare(
            self.value,
            b.value
        )
```

As explained in chapter 2, the evaluation method of the GC is not a comparison function between two GCs, but an evaluation of a garbled input *by* the GC. This does not conform to the architecture of the programming, which assumes that the client provides a node containing a GC to be compared to another GC. The node class does not appear to be designed to encapsulate anything else than the wrapper class for ORE, GC or plaintext data, suggesting that the actual requirements for GCs have not been considered during any of the early stages of development. This is however necessary to communicate complete treaps in order to support the usage of GCs. Looking at the current implementation of the client-server communication it is evident that there is a major discrepancy between the requirements and available solution. Due to the significant size of the task of re-implementing the client-server communications, this is left for future researchers to implement.

Chapter 4

Discussion and Conclusion

Although the activities described in this report have lead to significant progression regarding the goal of the project, work still has to be done in order to be able to perform the experiments. This chapter will discuss the current state of the project and give insight into the changes that still need to be made in order to make the comparison tool ready for further research.

4.1 State of the Project and Product

The activities described in this document have lead to some crucial advancements of the product and the project. First and foremost, the product has been analyzed in its initial state. Documentation of the product and its usage was severely lacking, a problem that has been solved for a great part with this report. Furthermore, several components that where identified to be incomplete have been completed by either altering existing programming or implementing them completely. This allowed for experimenting with the software to further test its functionality. This ultimately lead to the problem of comparing GCs, as described in subsection 3.2.3. As has been explained in the aforementioned section, solving this problem requires significant changes to the code-base. Section 4.2 provides insight into these changes for future work.

4.2 Advice for Future Research and Development

The focus of future continuations of the project should be that of rewriting server programming. The mechanisms it contains to compare GCs should be redesigned, keeping in mind that the ultimate goal of the experiments is to compare the performance of the GC and ORE in combination with treap data structures. A major delaying factor during the project was that of a lack of documentation of both programming and concepts. Documentation conventions can be enforced using *lint* tools, e.g. *PyLint*. Correct functionality can be ensured using unit tests. Conventions for clean code can be taken from literature such as *Clean Code* by Robert C. Martin. Putting an effort in maintaining a high quality of code, documentation and functionality using the mentioned resources will ultimately result in an easier to manage product that can be handed over between researchers more easily. Many of these principles have been demonstrated in the source code provided in Appendix C.

In the current state of the software, an effort is made to unify the client-server communication for both ORE and GC. As defined in section 2.2, these methods have very different requirements, as ORE simply encrypts data in such a way that comparisons can still be made while GC encrypts both the data and the comparison logic. Thus, implementing a client and server for both methods individually will be worth the effort in order to prevent having to program exceptions for each in the current implementation.

4.3 Conclusion

As a result of the work done during this project, new insight has been gained into the requirements of the software that is needed to compare ORE and GC order query encryption methods. Some improvements have been made to the existing software, but more advancements must be made before experimentation can begin. However, the software has been analyzed and documented for the better part, allowing for easier adoption by other researchers in the future, who can work on solving the problems that have been identified.

Bibliography

Tobias Boelter, Rishabh Poddar, and Raluca Ada Popa. A Secure One-Roundtrip Index for Range Queries. Master's thesis, University of California, Berkely, CA, April 2016. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-7.html`.

Nathan Chenette, Kevin Lewi, Stephen A. Weis, and David J. Wu. Practical Order-Revealing Encryption with Limited Leakage. In Thomas Peyrin, editor, *Fast Software Encryption*, volume 9783 of *Lecture Notes in Computer Science*, pages 747–493, Berlin, Germany, March 2015. Springer. ISBN 978-3-662-52993-5. doi: 10.1007/978-3-662-52993-5_24.

M. W. Grim and A. T. Wiersma. Security of Systems and Networks. Master's thesis, University of Amsterdam, Amsterdam, NL, February 2017. URL `https://www.delaat.net/rp/2016-2017/p37/report.pdf`.

Kevin Lewi and David J. Wu. Order-Revealing Encryption: New Constructions, Applications, and Lower Bounds. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1167–1178, New York, NY, October 2016. ACM SIGSAC, ACM. ISBN 978-1-4503-4139-4. doi: 10.1145/2976749.2978376.

Y. Daniel Liang. *Introduction to Java Programming*. Pearson, Harlow, England, ninth edition, 2013. ISBN 978-0-273-77138-8.

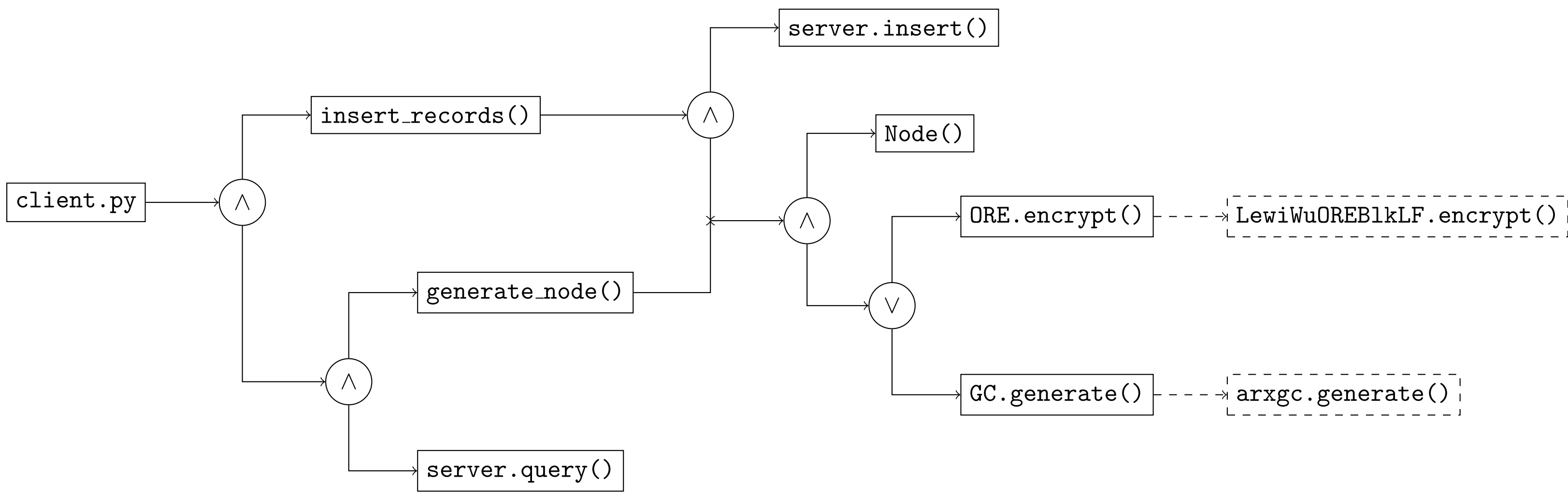
Dave Winer. XML-RPC Specification, June 1999. `http://xmlrpc.scripting.com/spec.html`, accessed on July 17th, 2019.

Appendix A

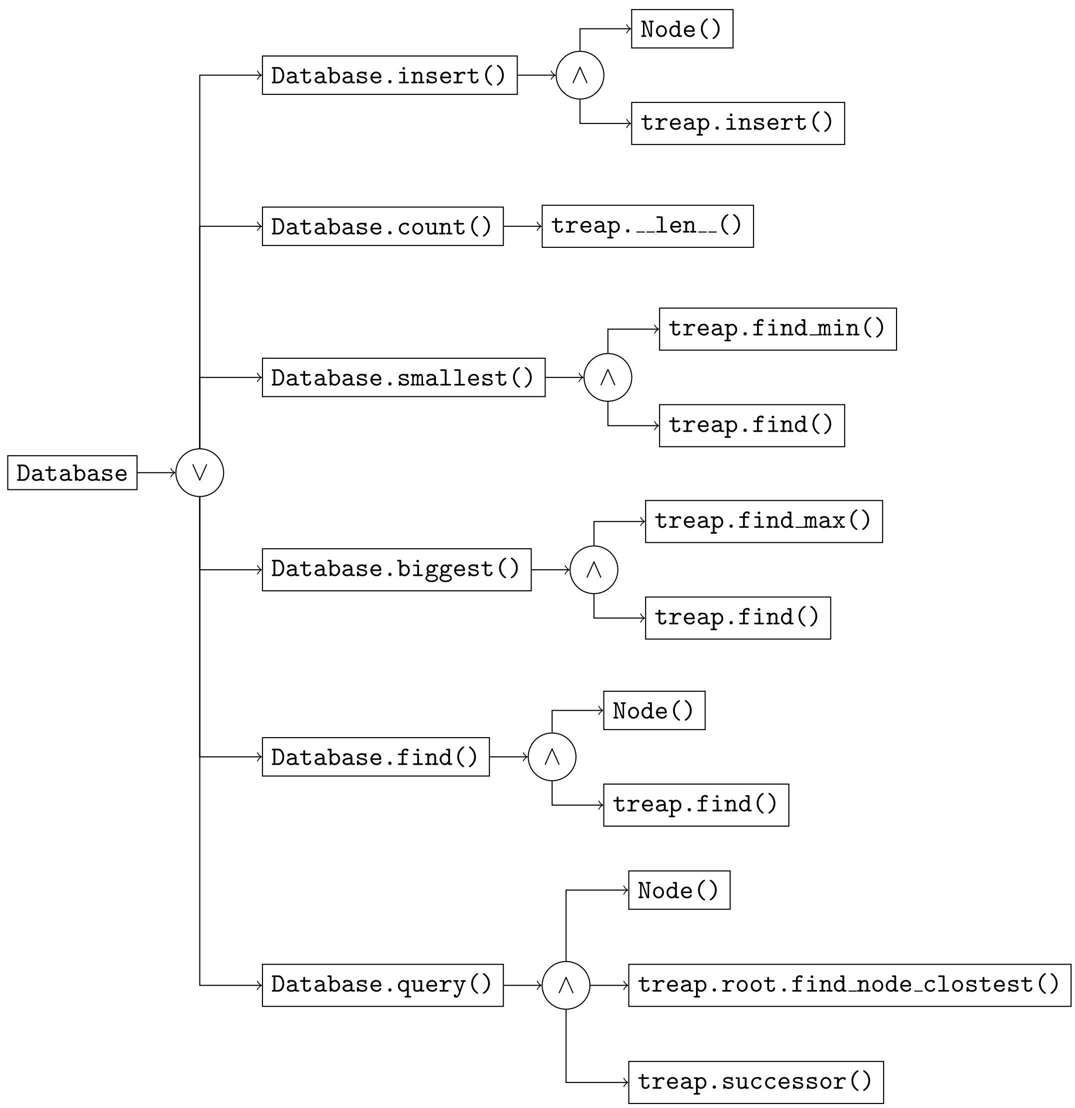
Functional Block Diagrams

This appendix contains several function block diagrams representing components of the code base presented at the start of the project. These diagrams provide an overview of the flow of execution that can occur upon calls to certain functions within the code. Each block represents such a function and operators indicate whether a specific set of blocks is called ($\wedge$) or if one or none of a specific set of blocks is called ($\vee$). The order of execution of a set of block is strictly speaking not indicated by these diagrams, but might be hinted at by the order in which they are listed from top to bottom. In some cases relations, blocks and operators are drawn with dashed lines, indicating that some logical details have been omitted in favor of readability.

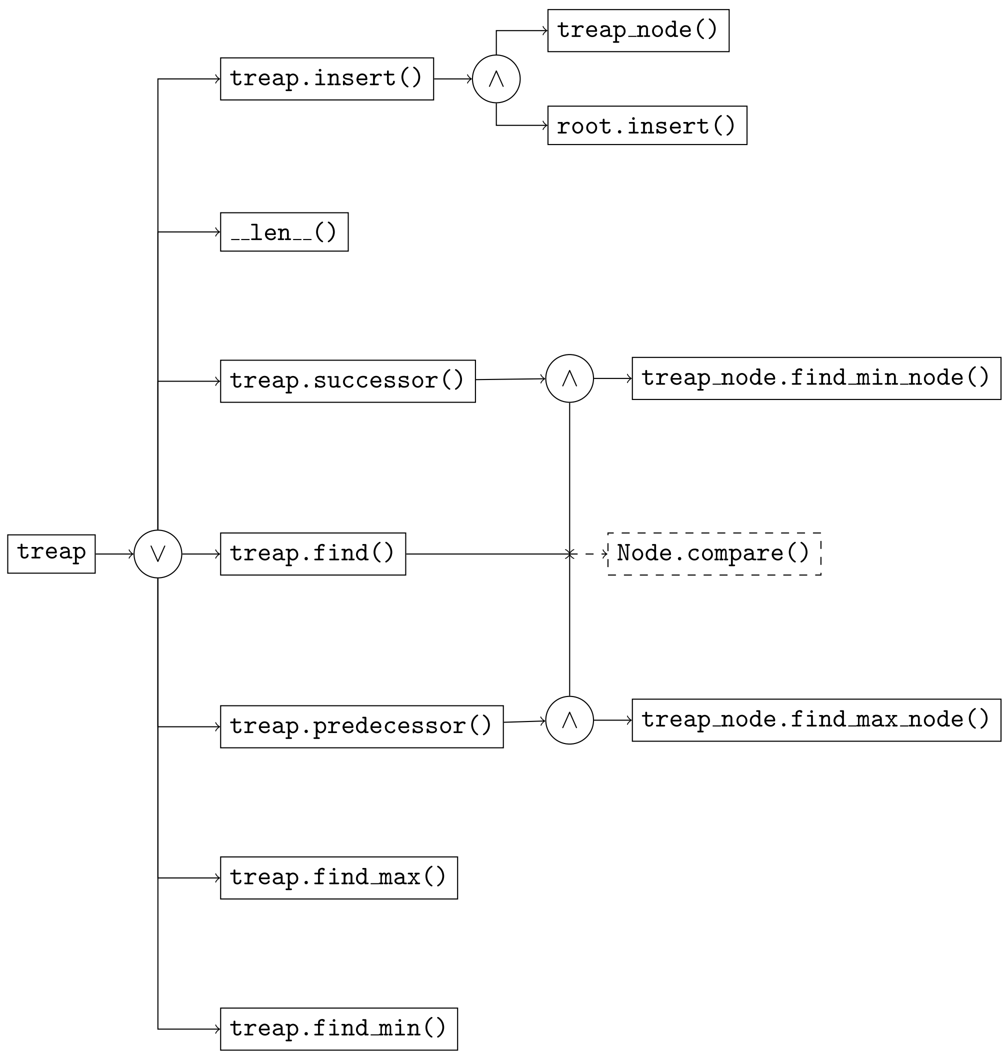
A.1 `client.py`


client.py
insert_records()
generate_node()
server.query()
server.insert()
Node()
ORE.encrypt()
LewiWuOREBlkLF.encrypt()
GC.generate()
arxgc.generate()

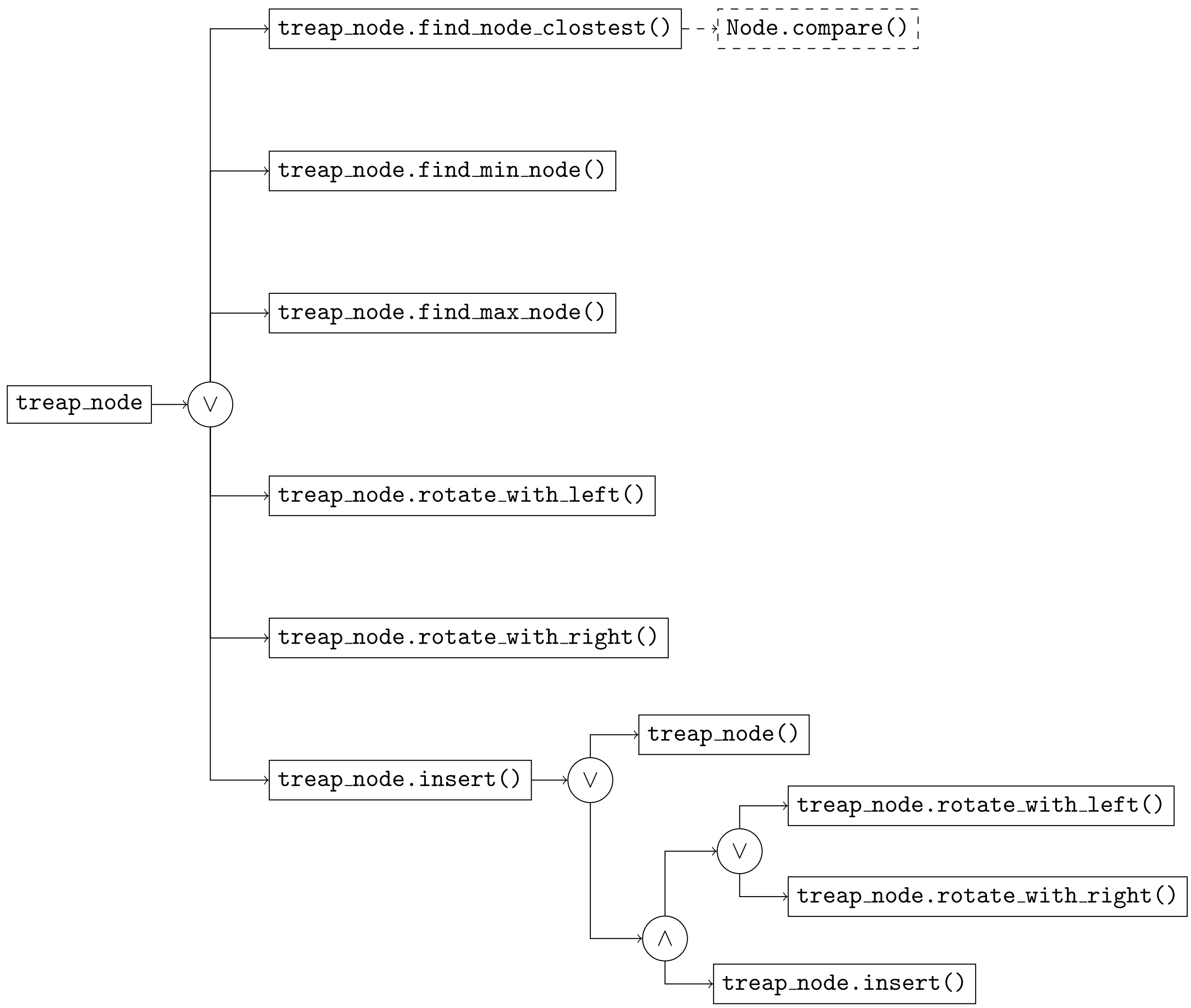
A.2 server.py: Database



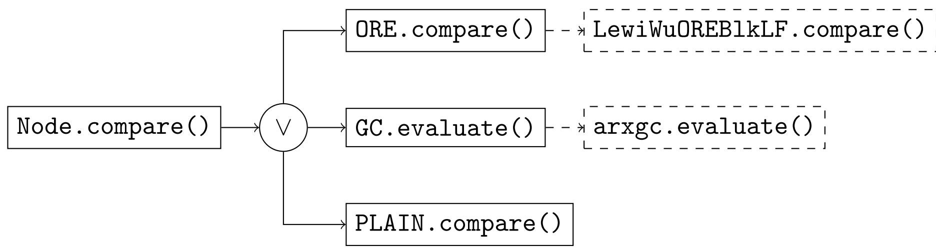
A.3 treap.py: treap



A.4 treap.py: treap_node



A.5 `node.py`: `Node`



Appendix B

Unit Tests

In this appendix, the source code for the unit test can be found. These tests are implemented as described in subsection 3.1.2. In order to run these tests, all dependencies of the *OrderQ* software must be installed as described on the relevant Git repository[1]. Furthermore, the `unittest` Python module must be installed. All tests can be executed at once with the command `python -m unittest discover -s test`, given that it is executed from the root directory of the *OrderQ* project.

B.1 `TestGC`

Listing B.1: `test_gc.py`

```python
"""
Test the functionality of the GC database implementation.
"""

from unittest import TestCase

# Have PyLint ignore the abundance of public methods,
# as this is in the very nature of a unit test.
#
# pylint: disable=too-many-public-methods
from arx import GC


class TestGC(TestCase):
    """
    Test the functionality of the GC database. More
    specifically test the custom wrapper for the GC
```

[1] `https://github.com/fturkmen/Private_OrderQ`

```
    library.
    """

    def setUp(self):
        """
        Set up the GC database.
        """
        self.gc = GC()

    def test_generate(self):
        """
        Assert that the database returns *something*
        when generating a garbled circuit.
        """
        self.assertIsNotNone(self.gc.generate(58))

    def test_encode(self):
        """
        Assert that the database returns *something*
        when encoding a value.
        """
        self.assertIsNotNone(
            self.gc.encode(99, self.gc.generate(45))
        )

    def test_evaluate_eq(self):
        """
        Assert that the database returns 0 when
        comparing a value to an equal value.
        """
        e = self.gc.generate(34)
        x = GC.evaluate(
                e,
                self.gc.encode(43, e)
            )
        self.assertEqual(
            0,
            GC.evaluate(
                e,
                self.gc.encode(43, e)
            )["result"]
        )

    def test_evaluate_lt(self):
        """
        Assert that the database returns 1 when
```

```
        comparing a value to a greater value.
        """
        e = self.gc.generate(92)
        self.assertEqual(
            1,
            GC.evaluate(
                e,
                self.gc.encode(86, e)
            )["result"]
        )

    def test_evaluate_gt(self):
        """
        Assert that the database returns 0 when
        comparing a value to a lesser value.
        """
        e = self.gc.generate(50)
        self.assertEqual(
            0,
            GC.evaluate(
                e,
                self.gc.encode(87, e)
            )["result"]
        )
```

B.2 TestORE

Listing B.2: test_ore.py

```
"""
Test the functionality of the ORE database
implementation.
"""

from unittest import TestCase

from ore import ORE


# Have PyLint ignore the abundance of public methods,
# as this is in the very nature of a unit test.
#
# pylint: disable=too-many-public-methods
class TestORE(TestCase):
    """
    Test the functionality of the ORE database. More
```

```
    specifically test the custom wrapper for the ORE
    library.
    """

    def setUp(self):
        """
        Set up the ORE database.
        """
        self.ore = ORE()

    def test_encrypt(self):
        """
        Assert that the database returns *something*
        when encrypting a value.
        """
        self.assertIsNotNone(
            self.ore.encrypt(98)
        )

    def test_compare_eq(self):
        """
        Assert that the database returns 0 when
        comparing a value to an equal value.
        """
        self.assertEqual(
            0, ORE.compare(
                self.ore.encrypt(18)[0],
                self.ore.encrypt(18)[1]
            )
        )

    def test_compare_lt(self):
        """
        Assert that the database returns -1 when
        comparing a value to a greater value.
        """
        self.assertEqual(
            -1, ORE.compare(
                self.ore.encrypt(29)[0],
                self.ore.encrypt(77)[1]
            )
        )

    def test_compare_gt(self):
        """
        Assert that the database returns 1 when
```

```
        comparing a value to a lesser value.
        """
        self.assertEqual(
            1, ORE.compare(
                self.ore.encrypt(63)[0],
                self.ore.encrypt(61)[1]
            )
        )
```

B.3 TestPlain

Listing B.3: test_plain.py

```
"""
Test the functionality of the plain database
implementation.
"""

from unittest import TestCase

from plain import PLAIN


# Have PyLint ignore the abundance of public methods,
# as this is in the very nature of a unit test.
#
# pylint: disable=too-many-public-methods
class TestPlain(TestCase):
    """
    Test the functionality of the plain database, which
    is a benchmark for the ORE and GC.
    """

    def test_compare_eq(self):
        """
        Assert that the database returns 0 a value to an
        equal value.
        """
        self.assertEqual(
            0, PLAIN.compare(87, 87)
        )

    def test_compare_lt(self):
        """
        Assert that the database returns -1 when
        comparing a value to a greater value.
```

```python
        """
        self.assertEqual(
            -1, PLAIN.compare(82, 90)
        )

    def test_compare_gt(self):
        """
        Assert that the database returns 1 when
        comparing a value to a lesser value.
        """
        self.assertEqual(
            1, PLAIN.compare(64, 12)
        )
```

B.4 TestTreap

Listing B.4: `test_treap.py`

```python
"""
Test the functionality of the treap implementation.
"""

from unittest import TestCase

import treap

# Example nodes available for the test case, taken from
# the example treap in Michaël's report, structured as
# illustrated in Fig. 1, Fig. 2 and Fig. 3, displaying
# either the order index, keys or priorities
# respectively. Each node is a tuple containing the node
# data in the following order: key, value, priority,
# depth and distance from the left in nodes.
#
#  +-----------+  +-----------+  +-----------+
#  |     4     |  |     6     |  |     2     |
#  |    / \    |  |    / \    |  |    / \    |
#  |   2   5   |  |   2   8   |  |   3   3   |
#  |  / \   \  |  |  / \   \  |  |  / \   \  |
#  | 1   3   6 |  | 1   5   9 |  | 8   3   5 |
#  +-----------+  +-----------+  +-----------+
#     Fig. 1         Fig. 2         Fig. 3

NODE_1 = (1, 'eggs', 8, 2, 0)
NODE_2 = (2, 'ham', 3, 1, 0)
NODE_3 = (5, 'bacon', 3, 2, 1)
```

```python
NODE_4 = (6, 'spam', 2, 0, 0)
NODE_5 = (8, 'sausage', 3, 1, 1)
NODE_6 = (9, 'beans', 5, 2, 3)

NODE_MIN = NODE_1
NODE_MAX = NODE_6

NODE = NODE_1

NODES = [
    NODE_4,
    NODE_2,
    NODE_1,
    NODE_3,
    NODE_5,
    NODE_6
]


NODES_ORDERED = [
    NODE_1,
    NODE_2,
    NODE_3,
    NODE_4,
    NODE_5,
    NODE_6
]


# Have PyLint ignore the abundance of public methods,
# as this is in the very nature of a unit test.
#
# pylint: disable=too-many-public-methods
class TestTreap(TestCase):
    """
    Test the functionality of the treap, using the nodes
    of the treap example in Michaël's report as test
    cases where needed.
    """

    def setUp(self):
        """
        Set up the treap to be the subject of the
        testing.
        """

```

```
        # Have PyLint ignore treap not being callable,
        # as this is an incorrect assumption.
        #
        # pylint: disable=not-callable
        self.treap = treap.treap()
        for key, value, priority, _, _ in NODES:
            self.treap.insert(key, value, priority)

    # Have PyLint ignore the casing of this method,
    # in favor of the casing conventions of the unittest
    # module.
    #
    # pylint: disable=invalid-name
    def assertNodeEqual(self, key, node):
        """
        A shorthand function that asserts equality in
        the specified key and the key of the specified
        node.

        :param key:
            The key of a node.
        :param node:
            A node having a key.
        """
        self.assertEqual(
            key, node.key
        )

    def test_find_node(self):
        """
        Assert that an inserted node can be found in the
        treap by its key.
        """
        key, value, _, _, _ = NODE
        node = self.treap.find_node(key)
        self.assertListEqual(
            [
                key,
                value
            ], [
                node.key,
                node.value
            ]
        )

    def test_insert(self):
```

```
        """
        Assert that a node is contained in the treap
        after insertion.
        """
        key, _, _, _, _ = NODE
        self.assertIn(key, self.treap)

    def test_remove(self):
        """
        Assert that a node is not contained in the treap
        after removal.
        """
        key, _, _, _, _ = NODE
        self.treap.remove(key)
        self.assertNotIn(key, self.treap)

    def test_remove_min(self):
        """
        Assert that the min node is not contained in the
        treap after min removal.
        """
        key, _, _, _, _ = NODE_MIN
        self.treap.remove_min()
        self.assertNotIn(key, self.treap)

    def test_remove_max(self):
        """
        Assert that the max node is not contained in the
        treap after max removal.
        """
        key, _, _, _, _ = NODE_MAX
        self.treap.remove_max()
        self.assertNotIn(key, self.treap)

    def test_get_key(self):
        """
        Assert that the key of an inserted node can be
        found in the treap.
        """
        key, _, _, _, _ = NODE
        self.assertEqual(
            key, self.treap.get_key(key)
        )

    def test_find(self):
        """
```

```
        Assert that an inserted node can be found in the
        treap.
        """
        key, value, _, _, _ = NODE
        self.assertEqual(
            value, self.treap.find(key)
        )

    def test_find_min(self):
        """
        Assert that the min node can be found in the
        treap.
        """
        key, _, _, _, _ = NODE_MIN
        self.assertEqual(
            key, self.treap.find_min()
        )

    def test_find_max(self):
        """
        Assert that the max node can be found in the
        treap.
        """
        key, _, _, _, _ = NODE_MAX
        self.assertEqual(
            key, self.treap.find_max()
        )

    def test_predecessor(self):
        """
        Assert that the predecessor of a node can be
        found.
        """
        key_1, _, _, _, _ = NODE_1
        key_2, _, _, _, _ = NODE_2
        self.assertNodeEqual(
            key_1, self.treap.predecessor(
                self.treap.find_node(key_2)
            )
        )

    def test_successor(self):
        """
        Assert that the successor of a node can be found.
        """
        key_1, _, _, _, _ = NODE_1
```

```python
        key_2, _, _, _, _ = NODE_2
        self.assertNodeEqual(
            key_2, self.treap.successor(
                self.treap.find_node(key_1)
            )
        )

    def test_inorder_traversal(self):
        """
        Assert that the treap is traversed in the
        expected order.
        """
        nodes = iter(NODES_ORDERED)

        def visit(visit_key, visit_value):
            next_key, next_value, _, _, _ = next(nodes)
            self.assertListEqual(
                [
                    next_key,
                    next_value
                ], [
                    visit_key,
                    visit_value
                ]
            )

        self.treap.inorder_traversal(visit)

    def test_detailed_inorder_traversal(self):
        """
        Assert that the treap is traversed in the
        expected order.
        """
        nodes = iter(NODES_ORDERED)

        def visit(
                node,
                visit_key,
                visit_value,
                visit_depth,
                visit_from_left
        ):
            next_key, next_value, _, \
                next_depth, next_from_left = next(nodes)
            self.assertNodeEqual(next_key, node)
            self.assertListEqual(
```

```
                [
                    next_key,
                    next_value,
                    next_depth,
                    next_from_left
                ], [
                    visit_key,
                    visit_value,
                    visit_depth,
                    visit_from_left
                ]
            )

        self.treap.detailed_inorder_traversal(visit)

    def test_check_tree_invariant_preserved(self):
        """
        Assert that the tree invariant is preserved,
        i.e. a < v < b.
        """
        self.assertTrue(
            self.treap.check_tree_invariant()
        )

    def test_check_heap_invariant(self):
        """
        Assert that the heap invariant is preserved,
        i.e. a < v > b.
        """
        self.assertTrue(
            self.treap.check_heap_invariant()
        )

    def test_depth(self):
        """
        Assert that the depth of the treap is correct.
        """
        self.assertEqual(
            3, self.treap.depth()
        )

    def test_iterkeys(self):
        """
        Assert that the keys in the treap can be
        iterated.
        """
```

```
        keys = [k for k, _, _, _, _ in NODES]
        for key in self.treap.iterkeys():
            self.assertIn(key, keys)
            keys.remove(key)

    def test_keys(self):
        """
        Assume that this functionality is identical to
        treap.iterkeys().
        """
        self.test_iterkeys()

    def test_iterator(self):
        """
        Assume that this functionality is identical to
        treap.iterkeys().
        """
        self.test_iterkeys()

    def test_itervalues(self):
        """
        Assert that the values in the treap can be
        iterated.
        """
        values = [v for _, v, _, _, _ in NODES]
        for value in self.treap.itervalues():
            self.assertIn(value, values)
            values.remove(value)

    def test_values(self):
        """
        Assume that this functionality is identical to
        treap.itervalues().
        """
        self.test_itervalues()

    def test_iteritems(self):
        """
        Assert that the node in the treap can be
        iterated.
        """
        items = [(k, v) for k, v, _, _, _ in NODES]
        for key, value in self.treap.iteritems():
            self.assertIn((key, value), items)
            items.remove((key, value))
```

```
    def test_items(self):
        """
        Assume that this functionality is identical to
        treap.iteritems().
        """
        self.test_iteritems()

    def test_reverse_iterator(self):
        """
        Assert that the nodes in the treap can be
        iterated in reverse.
        """
        keys = reversed([
            key for key, _, _, _, _ in NODES_ORDERED
        ])
        for a, b in zip(
            keys, self.treap.reverse_iterator()
        ):
            self.assertEqual(a, b)
```

B.5 Results

Listing B.5: The results of executing the unit tests on the initial version of the code.

```
EEEE...............................
======================================================================
ERROR: test_encode (test_gc.TestGC)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "[PROJECT ROOT]/test/test_gc.py", line 40, in
      test_encode
    self.gc.encode(99, self.gc.generate(45))
TypeError: encode() takes 2 positional arguments but 3
    were given

======================================================================
ERROR: test_evaluate_eq (test_gc.TestGC)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "[PROJECT ROOT]/test/test_gc.py", line 53, in
      test_evaluate_eq
    self.gc.encode(43, e)
TypeError: encode() takes 2 positional arguments but 3
    were given

```

```
======================================================================
ERROR: test_evaluate_gt (test_gc.TestGC)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "[PROJECT ROOT]/test/test_gc.py", line 81, in test_evaluate_gt
    self.gc.encode(87, e)
TypeError: encode() takes 2 positional arguments but 3 were given

======================================================================
ERROR: test_evaluate_lt (test_gc.TestGC)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "[PROJECT ROOT]/test/test_gc.py", line 67, in test_evaluate_lt
    self.gc.encode(86, e)
TypeError: encode() takes 2 positional arguments but 3 were given

----------------------------------------------------------------------
Ran 36 tests in 0.003s

FAILED (errors=4)
```

Appendix C

Implementation

C.1 GC Wrapper

This section contains the completed implementation for the GC wrapper. Note that the initial version of this class did not contain any functionality and had an incorrect signature for the `encode` method.

Listing C.1: Completed implementation of the GC wrapper in `arx.py`.

```python
import arxgc


class GC:

    # def __init__(self, adict=None):
    #     """
    #     Convert a dictionary to a class
    #     @param :adict Dictionary
    #     """
    #     if adict is not None:
    #         self.__dict__.update(adict)
    #         for k, v in adict.items():
    #             if isinstance(v, dict):
    #                 self.__dict__[k] = GC(v)
    #
    #
    # def get_object(adict):
    #     """
    #     Convert a dictionary to a class
    #     @param :adict Dictionary
    #     @return :class:Struct
    #     """
    #     return GC(adict)
```

```


    def generate(self, f):
        """
        Given boolean circuit f, return encoding
        information e.
        """
        return arxgc.generate(f)

    def encode(self, x, e):
        """
        Given plaintext x and encoding information e,
        return garbled input X (i.e. extracted labels).
        """
        return arxgc.encode(x, e)

    @staticmethod
    def evaluate(e, X):
        """
        Given encoding information e and garbled
        input X (extracted labels) for an input x, return
            evaluation.
        """
        return arxgc.evaluate(
            e["gc"],
            e["input_labels"],
            e["output_labels"],
            e["arx_table_zero_labels"],
            e["arx_table_one_labels"],
            X
        )
```

C.2 Pack & Unpack

Listing C.2: Implementation of the function to convert all integers in GC data to string and vice versa.

```
def pack(value):
    """
    Recursively convert any ints in the specified
    collection to strings as their value may exceed
    32 bits. Each string has a prefix for easy
    unpacking.
    """
    if type(value) is int:
```

```
        return "__int__" + str(value)
    elif type(value) in (list, tuple):
        return list([pack(v) for v in value])
    elif type(value) is dict:
        return {k: pack(v) for k, v in value.items()}
    else:
        return value


def unpack(value):
    """
    Recursively unpack the specified value by casting
    strings starting with a specific prefix to
    integers.
    """
    if type(value) is str \
            and value.startswith("__int__"):
        return int(value[7:])
    elif type(value) in (list, tuple):
        return tuple([unpack(v) for v in value])
    elif type(value) is dict:
        return {k: unpack(v) for k, v in value.items()}
    else:
        return value
```