Universelle Verifizierung des Genfer E-Voting Systems

Bachelorthesis

Studiengang:	Informatik
Autor/in:	Christian Wenger
Betreuer/in:	Dr. Rolf Haenni
Auftraggeber/in:	Dr. Rolf Haenni
Experten:	Thomas Hofer
Datum:	14.01.19

Management Summary

Die fortan wachsende Digitalisierung macht auch keinen halt vor der Politik. So begann das Parlament im Jahr 2000 mit den Vorbereitungen für die elektronische Stimmabgabe kurz E-Voting. Doch ein solches System zu bauen ist ziemlich komplex. Denn es muss den aktuellen Sicherheitsanforderungen standhalten. Einer dieser Anforderungen ist die universelle Verifizierbarkeit. Das Ziel der Bachelor-Thesis war es, einen Verifier zu entwickeln, welcher diese Anforderung erfüllt.

Bei der universellen Verifizierbarkeit geht es darum, die Verlierer einer Abstimmung davon zu überzeugen, dass das Resultat korrekt ist. Bei der brieflichen Stimmabgabe geschieht dies durch eine mögliche Nachzählung der Stimmen. Im E-Voting ist dies leider nicht so einfach. Deshalb gibt es eine Applikation, genannt *Verifier*, welche von einer unabhängigen Instanz ausgeführt werden kann. Diese überprüft nun systematisch alle Abstimmungsdaten. Somit lässt sich feststellen, ob die Daten manipuliert wurden. Aber auch Softwarefehler lassen sich damit aufdecken. Eine einzelne solche Überprüfung wird als *Test* bezeichnet. Die Tests lassen sich in fünf Kategorien einordnen.

Bei der Kategorie Vollständigkeit wird überprüft, ob alle erforderlichen Daten vorhanden sind. Denn nur so kann man eine lückenlose Verifizierung gewährleisten. Jeder Test überprüft, ob ein Parameter vorhanden ist oder nicht.

Bei der Kategorie Integrität wird die Integrität der Parameter geprüft. Es wird also geprüft, ob die Parameter in sich schlüssig sind. Beispielweise wird geprüft, ob sie sich in den geforderten Wertebereichen befinden.

Bei der Kategorie Konsistenz wird geprüft, ob die Parameter zu den anderen konsistent sind, zum Beispiel zwei Vektoren, welche die gleiche Länge haben sollten. Nun wird geprüft, ob diese wirklich die gleiche Länge haben.

Bei der Kategorie Evidenz wird geprüft, ob die verschieden kryptographischen Beweise stimmen.
Bei der Kategorie Authentizität wird die Gültigkeit der Zertifikate und Signaturen überprüft.

Das Ziel der Arbeit war es, die bestehende Applikation „NextGenVote Visualization“ mit dem Verifier zu vervollständigen. Die Algorithmen für die Tests wurden in Python implementiert. Da bei dem System die Zertifikate und Signaturen fehlten, wurden „successful“, „failed“ und „skipped“ als Ausgabe der Tests definiert. Der Ausgabewert „skipped“ wurde verwendet, wenn keine Testdaten vorhanden waren.
Das Front-End wurde mit *Vue.js* umgesetzt. Dabei war ein Hauptziel den User möglichst genau zu informieren, was das Backend im Hintergrund macht. Dazu benötigt das Frontend eine Kommunikationsschnittstelle, welche mit *Socket.io* umgesetzt wurde.

Inhaltsverzeichnis

1 Einleitung

Das Internet ist heute aus unserem Leben nicht mehr wegzudenken. Wir kaufen online ein, holen von Google was wir wissen wollen und sogar unsere Bewerbungen können wir damit versenden. Dieser Fortschritt machte auch keinen Halt vor der Politik. So begann das Parlament im Jahr 2000 mit den Vorbereitungen für die elektronische Stimmabgabe kurz E-Voting [1, S. 1]. Doch ein solches System zu bauen ist ziemlich komplex. Denn es muss den aktuellen Sicherheitsanforderungen standhalten.
Einer dieser Anforderungen ist die universelle Verifizierbarkeit. Das Ziel der Bachelor-Thesis war es, einen Verifier zu entwickeln welcher diese Anforderung erfüllt.

1.1 Elektronische Stimmabgabe

Bei der Elektronischen Stimmabgabe in der Schweiz kann der Wähler mit seinem privaten Endgerät abstimmen oder eine Wahl tätigen. Dass die Stimmabgabe dabei in einer nicht behördlich kontrollierten Umgebung stattfindet, ist bereits durch die briefliche Stimmabgabe etabliert und akzeptiert [1, S. 3].
Zurzeit gibt es zwei vorhandene Systeme. Einmal jenes des Kanton Genf (CHVote) und dasjenige der Schweizerischen Post.

1.2 Verifizierbarkeit

Ein grosses Problem dieser Systeme ist die Verifizierbarkeit der einzelnen Schritte. Dabei ist zwischen individueller und universeller Verifizierbarkeit zu unterscheiden. Bei der individuellen Verifizierbarkeit kann der Wähler selbst durch Prüfcodes überprüfen, ob seine Stimme wirklich richtig abgegeben wurde.

Ziel der universellen Verifizierbarkeit ist es, dass ein Dritter die Wahlergebnisse mit einem sogenannten *Verifier* überprüfen kann. Somit lässt sich dann bestätigen, dass alle Abstimmungsdaten korrekt sind. Spätestens jetzt müsste also eine Manipulation an den Abstimmungsdaten erkannt werden. Die Anforderung der universellen Verifizierbarkeit, wurde von Bundesrat am 5. April 2017 gestellt [2, S. 4]

1.3 Verifier

Mit einem Verifier lässt sich nach der Abstimmung überprüfen, ob alle Abstimmungsdaten korrekt sind und somit keine Wahlmanipulation stattgefunden hat. Damit der Verifier dies beweisen kann, müssen alle Abstimmungsdaten bei jedem Schritt vom Erzeuger digital signiert werden. Ausserdem müssen bei einigen Schritten noch kryptographische Beweise erzeugt werden.

Im Folgenden wird die Definition der universellen Verifizierung für Internetwahlen [3, S. 12–13] verwendet. Der Verifier ist in fünf Schritten unterteilt. In im ersten Schritt wird die Vollständigkeit der Abstimmungsdaten geprüft. Der Verifier verfügt also über das Wissen, welche Daten vorhanden sein müssten und prüft diese Systematisch. Im Zweiten Schritt wird die Integrität der Abstimmungsdaten geprüft. Also, ob alle Abstimmungsdaten in sich korrekt sind. Zum Beispiel, ob sich die Abstimmungsdaten in den gewünschten Wertebereichen befinden oder nicht. Im dritten Schritt wird die Konsistenz der Abstimmungsdaten geprüft. Beispielsweise, ob zwei Vektoren, welche die gleich lang sein müssten, auch wirklich gleich lang sind. Im vierten Schritt wird die Evidenz geprüft. Dazu müssen alle Kryptographischen Beweise auf ihre Richtigkeit geprüft werden. Im fünften Schritt wird die Authentizität geprüft. Dazu werden alle Signaturen auf ihre Richtigkeit geprüft. Somit kann überprüft werden, ob die Abstimmungsdaten von der dazu befugten Person erzeugt wurden.

die einzelnen Überprüfungsschritte innerhalb dieser Kategorien werden im Folgenden als *Test* bezeichnet. Die Summe dieser Tests definieren dann den universelle Verifizierungsprozess. Der *Verifizierungsbericht* wird aus der Summe der erhaltenen Antworten dieser Tests gebildet. In der Folgende Abbildung ist diese grundlegende Funktionsweise schematisch dargestellt.

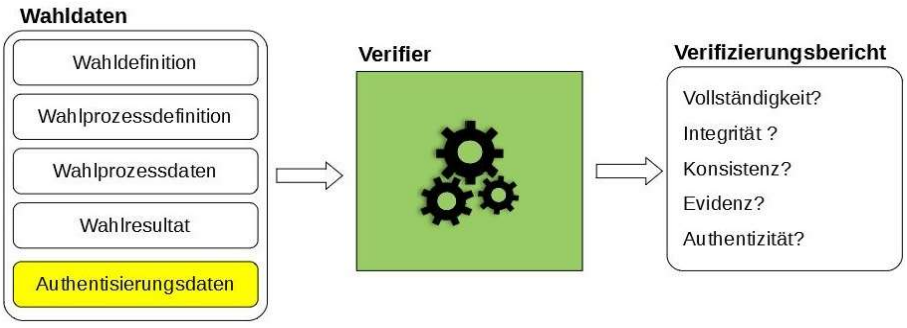


Abbildung 1: Funktionsweise des Verifiers In Ahnlehnung an [3, S. 13]

2 Aufgabenstellung

In der Projekt 2 Arbeit wurden alle Abstimmungsdaten systematisch aufgelistet und ein Katalog mit den erforderlichen Tests erstellt. In der Bachelor-Thesis galt es nun diese Tests zu implementieren und dessen Resultate in einem Verifizierungsbericht darzustellen.

Bachelorthesis-Aufgabe

Universelle Verifizierbarkeit des Genfer E-Voting Systems

ID	HNR1-1-18
Studierende	Christian Wenger
Betreuer	Dr. Rolf Haenni
Experten	Thomas Hofer
Verteidigung	Ort: BFH-TI Biel, Rolex-Gebäude, Höheweg 80, Saal N311. Datum: 24.01.2019 Zeit: 09.30-10.30
Aufgabe	Der Kanton Genf entwickelt zur Zeit eine vollständig überarbeitete Version ihres E-Voting-Systems, damit es den neuen Anforderungen der Bundeskanzlei entspricht. Die wichtigste neue Anforderung ist die universelle Verifizierbarkeit. Diese bedeutet, dass nach Abschluss eines Wahlgangs das ermittelete Wahlergebniss durch externe Personen verifiziert werden kann, wobei dazu ausschlisslich die während dem Wahlprozess publizierten Wahldaten benutzt werden können. Mittels verschiedensten kryptografischen Methoden wird dabei gewährleistet, dass das Stimmgeheimnis nicht gebrochen werden kann. In dieser Bachelort-Arbeit geht es darum, eine Software zu entwickeln, die eine solche universelle Verifizierung für das Genfer System durchführen kann. Als Basis für diese Arbeit gilt die öffentlich zugängliche Spezifikation des kryptografischen Protokolls, die von anderen Studierenen realisierte Visualisierungssoftware des Genfer Systems, sowie die in der Projektarbeit 2 erarbeitete Zusammenstellung der nötigen kryptografischen Tests.

Es wurde entschieden eine bestehend Bachelor-Thesis mit dem Verifier zu vervollständigen. Deshalb wurden die Tests in Python implementiert und für die Präsentation des Resultats wurde *Vue.js* verwendet.

3 Projektmanagement

In diesem Kapitel werden die verschiedenen Schritte der Projektplanung erläutert. Dies beinhaltet die Methodik, die Anforderungen sowie den Zeitplan.

Dass es sich hierbei um ein Ein-Mann-Projekt handelt, bietet sich eine agile Methodik an. Es wurde aber kein richtiges Model wie «SCRUM» gewählt, da dies den Rahmen dieses Projektes sprengen würde.
Es wurde entschied sich regelmässig zu Treffen und den Aktuellen Stand zu präsentieren, sowie die nächsten Schritte zu besprechen. In der Anfangsphase waren diese Treffen 1 Mal pro Woche und wurden dann auf alle 14 Tagen ausgeweitet.

3.1 Anforderungen

In einem ersten Schritt galt es den Rahmen der Applikation zu bestimmen. Dazu muss abgeklärt werden, was die Applikation können muss. Dies wurde vom Studierenden definiert und vom Betreuer genehmigt.

3.1.1 Allgemeine Anforderungen

ID	Beschreibung	Priorität	Status
A.1	Alle Teste von der Projektarbeit 2 sind nach Definition implementiert.	Muss	Erfüllt
A.2	Die bestehend Applikation zu Visualisierung des Genfer E-Voting System wird mit dem Verifier vervollständigt	Muss	Erfüllt
A.3	Eine Konsolenanwendung um die Teste lokal auszuführen wird entwickelt. Dazu ist eine Schnittstelle zur Bestehenden Applikation nötig.	Kann	Erfüllt
A.4	Die Applikation kann in Deutsch, Englisch und Französisch genutzt werden	Kann	Nicht Erfüllt
A.5	Ein Script für die Generierung der Abstimmungsdaten einer Volksabstimmung wird entwickelt.	Kann	Nicht Erfüllt

3.1.2 Verifier Visualisierung

ID	Beschreibung	Priorität	Status
A.6	Der Benutzer muss jederzeit nachvollziehen können, wie viel der Verifier schon getestet hat und bei welchem Test er gerade ist.	Muss	Erfüllt
A.7	Der Benutzer erhält am Ende der Tests einen Bericht, welche Tests erfolgreich waren und welche nicht.	Muss	Erfüllt
A.8	Im Bericht gibt es die Möglichkeit zu jedem Test detaillierte Informationen anzuzeigen	Muss	Erfüllt
A. 9	Der Bericht kann nach All, Successful, Failed und Skipped gefiltert werden	Kann	Erfüllt
A.10	Der Benutzer kann den Bericht als PDF exportieren	Kann	Nicht Erfüllt

3.2 Meistertask

Meistertask ist eine Web-Applikation, um seine Aufgaben zu organisieren. Es wurde dazu verwendet, um die Aufgaben des Projektplans in kleine Einheiten aufzuteilen. Es bietet einen Überblick über die getätigten Aufgaben und solche die noch anstehen. So wird die Gefahr gemindert, dass eine Aufgabe vergessen wird. Nach Möglichkeit wurde auch gerade ein Fälligkeitsdatum definiert. Somit wusste man immer welche Aufgabe bis wann erledigt werden musste. Gleichzeit wurde es als Protokoll für die regelmässigen Meetings mit dem Betreuer verwendet. Die nächsten Schritte wurden dann in Form von Aufgaben definiert.

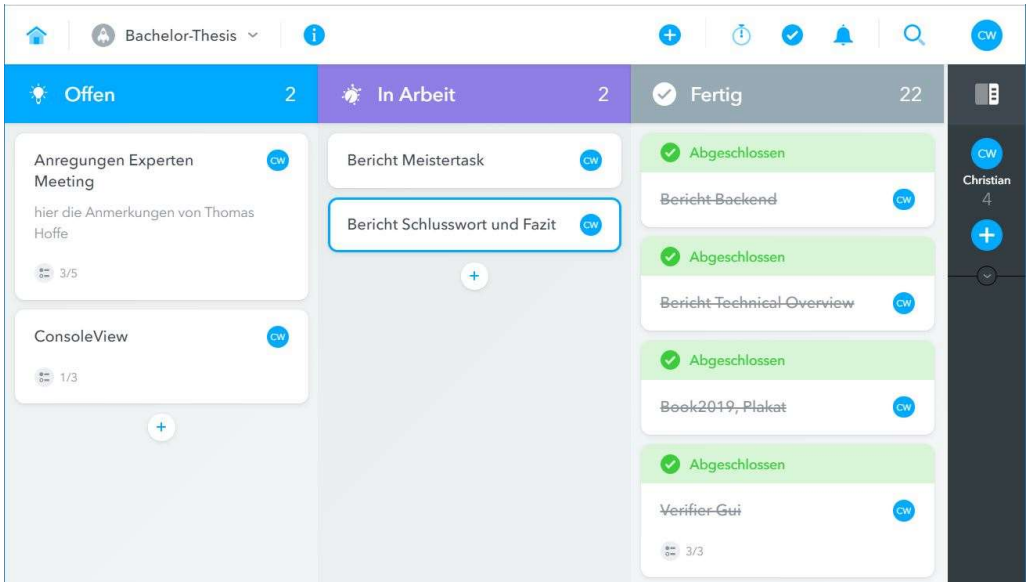


Abbildung 2: Beispiel Meistertask

3.3 Zeitplan und Meilensteine

Im nächsten Schritt wurde der Zeitplan und die Meilensteine definiert

Folgende Meilensteine wurden anhand der Anforderungen definiert

- M1: Projekt Initialisierung ist fertig / Es können nun Anpassungen an der bestehenden Applikation gemacht werden.
- M2: Implementation der Tests in Python ist abgeschlossen
- M3: Die GUI-Programmierung ist abgeschlossen
- M4: Das Backend ist mit dem Frontend verknüpft / Alle Muss-Kriterien sind erfüllt
- M5: Alle Kann-Kriterien wurden erfüllt
- M6: Dokumentation ist abgeschlossen


Projekt Aufgaben
38 39 40 41 M1 42 43 44 M2 45 46 47 M3 48 49 M4 50 51 52 M5 53 54 M6
Konzept und Planung
Anforderungen und Meilensteine definieren
soll
ist
Architektur und Prototyp(Mockup)
soll
ist
M1: Projekt Initialisieren
Bestehende Appliketion editieren können
soll
ist
M2: Implementation Python
Von jeder Kategorie mindestens ein Test Implementieren
soll
ist
Softwaredesing umsetzten
soll
ist
M3: UI-Programmierung
ConsoleView Programmierung
soll
ist
GUI Programmierung nach Prototyp
soll
ist
M4: Verknüpfung Backend/Frontend
Alle Teste in Python implementieren
soll
ist
Backend und Frontend verknüpfen / Alle Muss-Kriterien erfüllen
soll
ist
M5: Applikation erweitern
Kann-Kriterien erfülllen
soll
ist
M6: Dokumentation
Poster, Finaltag, Präsentation,Film
soll
ist
Dokumentation ergänzen
soll
ist

4 Die Applikation NextGenVote Visualization

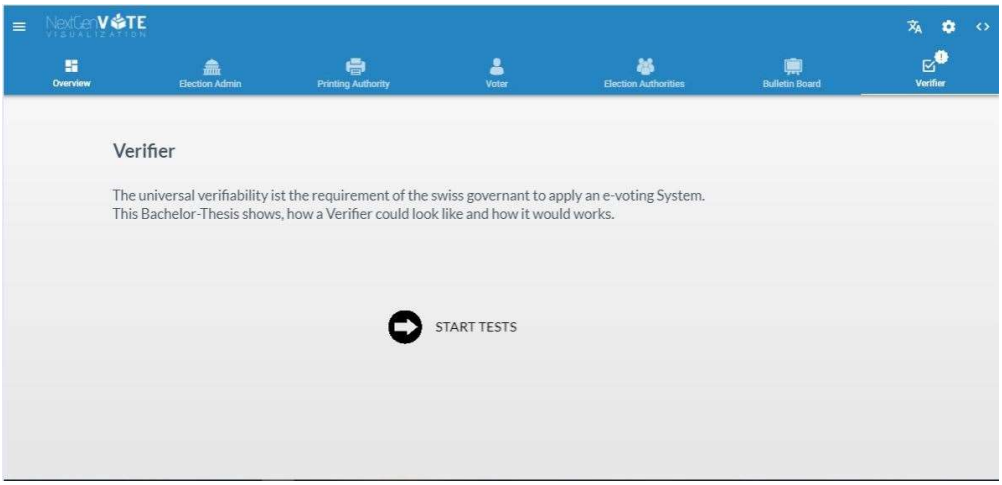
In einer früheren Bachelor-Thesis [4] wurde die Applikation «NextGenVote Visualization» entwickelt. Ziel war es das Genfer E-Voting Protokoll zu visualisieren. Somit lässt sich das Thema E-Voting besser fassen. Es wurde für jeden Teilnehmer des Protokolls eine Ansicht entwickelt. Folgende Teilnehmer werden im Protokoll verwendet:

- **Wahladministration:** Die Wahladministration ist für das Eröffnen einer Abstimmung und das Veröffentlichen des Resultats zuständig.
- **Druckerei:** Die Druckerei ist für das Drucken des Stimmausweis zuständig. Ein Stimmausweis enthält folgende Informationen:
  - **Voting Code:** Dieser wird für das abgeben der Stimme benötigt.
  - **Verfication Codes:** Jeder möglichen Auswahl, wird ein Code zugewiesen.
  - **Confirmation Code**: Wenn der Verfication Code mit meiner Auswahl übereinstimmt, bestätige wird dies mit dem Confirmation Code bestätigt.
  - **Finalization Code**: Um zu überprüfen ob der Confirmation Code korrekt übermittelt wurde, wird ein Finalization Code angezeigt.
- **Wahlautoritäten:** Die Wahlautoritäten sind für die Integrität und Privatsphäre der Stimmen verantwortlich. Sie generieren die Codes für die Druckerei. Am Ende des Wahlgangs werden die Stimmen gemischt und dann entschlüsselt. Beim Mischen wird der Wähler von seiner Stimme getrennt. Es ist somit nicht mehr möglich, nach dem Entschlüsseln von einer Stimme, auf den Wähler zu Schliessen. Somit wird das Stimmgeheimnis gewahrt. Das entschlüsseln kann nur stattfinden, wenn alle Autoritäten einverstanden sind.
- **Bulletin-Board:** Hier werden alle Daten abgelegt. Jedoch besitzt das Bulletin Bord einen öffentlichen Teil und je einen Teil nur für Wahladministration und Wahlautoritäten. In der Ansicht wurde der öffentliche Teil dargestellt. Der Verifier besitzt nun eine Schnittstelle, um auf diese Daten zuzugreifen.

Der Verifier ist nicht Teil des Protokolls. Er kommt erst zum Zug, wenn das Protokoll abgeschlossen ist. In der früheren Bachelor-Thesis wurden nur ein paar wenige Tests implementiert. In dieser Bachelor-Thesis wurde nun der Verifier vollständig implementiert.

5 Prototyp

In diesem Kapitel wird beschrieben wie die Applikation später einmal aussehen soll. Um das Backend programmieren zu können, musste klar sein, welche Funktionen die Applikation schlussendlich hat. Zu diesem Zweck wurde ein Prototyp des Frontends erstellt. In der ersten Ansicht wird beschrieben was das Ziel der Bachelor-Thesis ist und wozu ein Verifier nötig ist. Diese Ansicht wurde später wieder entfernt, da dies der Fluss der Applikation stört. Die Beschreibung der Bachelor-Thesis wurde in die About-Ansicht verschoben.



5.1 Fortschritt Visualisierung

Nach den Anforderungen muss ersichtlich sein, was der Verifier im Hintergrund gerade macht. Um dies aufzuzeigen wechseln die abgeschlossenen Kategorien ihre Farbe und auch ihre Intensität.

- **Rot**: Mindestens ein Test dieser Kategorie ist fehlgeschlagen.
- **Orange:** Mindestens ein Test konnte nicht durchgeführt werden, da keine Testdaten vorhanden waren.
- **Grün**: Alle Tests wurden erfolgreich abgeschlossen.
- **Intensität = 0%**: In dieser Kategorie wurde noch kein Test gestartet.
- **Intensität = 40%**: Der Verifier befinden sich nun in dieser Kategorie.
- **Intensität = 100%**: Diese Kategorie wurde abgeschlossen.

Abbildung 3: Jede Kategorie kann die Farbe Grün oder Rot und eine Intensität zwischen 0 und 100 Prozent haben

Um zu zeigen wie viel noch getestet werden muss wurde ein Fortschrittsbalken verwendet. Ausserdem wird immer angezeigt, welcher Test gerade durchgeführt wird.

Abbildung 4: Fortschrittsbalken mit aktuell durchzuführendem Test

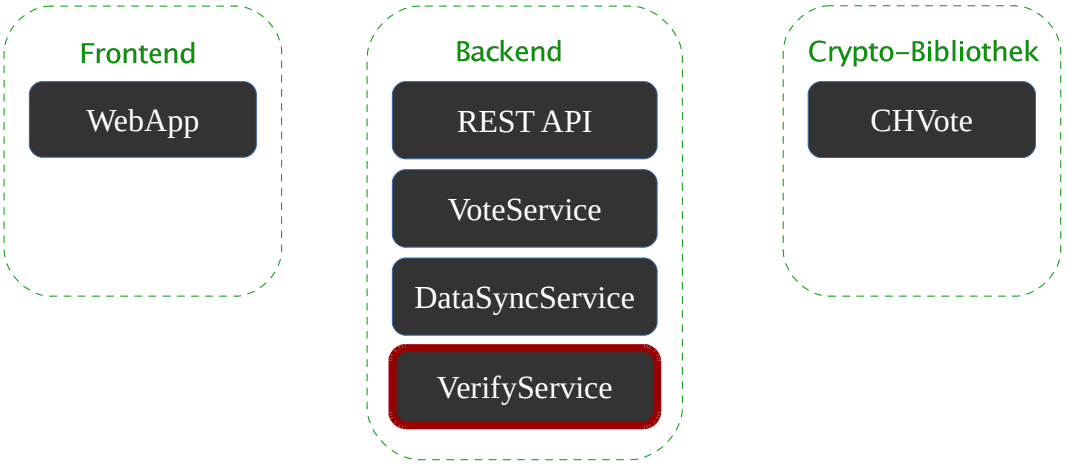
Wenn alle Tests durchgeführt wurden, kann man auf die gewünschte Kategorie klicken und bekommt dann einen Bericht mit allen durchgeführten Tests in dieser Kategorie und dessen Ergebnisse.

Abbildung 5: Resultat einer Kategorie. Die einzelenen Resultate können aufgeklapt werden.

Will man mehr über einen einzelnen Test wissen, klickt man auf den gewünschten Test und sieht dann alle Information die nötig sind, um den Test selbst nachzurechnen. Somit lässt sich jedes Ergebnis nachvollziehen. Auch hier wurden wieder die Farben grün und rot für erfolgreich und fehlgeschlagen verwendet. Zusätzlich lässt sich der Bericht nach erfolgreich und fehlgeschlagen filtern.

6 Technische Implementation

Durch die Entscheidung die Applikation der Bachelor-Thesis «Visualizing Geneva's Next Generation E-Voting System» mit dem Verifier zu vervollständigen, wird auch dessen Architektur verwendet.
Diese Kapitel beschreibt die technische Implementation der Applikation. Die **Zeichnung 1** gibt eine sehr abstrakte Übersicht der Architektur. Jede Komponente kann als Black-Box angesehen werden.



Zeichnung 1: Aus abstrakter Sicht besteht die Applikation aus 3 Hauptkomponenten: das Front-End (Web-Applikation), das Back-End, und die Crypto-Bibliothek, in Anlehnung an [4, S. 25]

- Das **Frontend** bietet die Hauptfunktionalitäten der Applikation. Hier wird der Prozess des Verifiers visualisiert. Man hat einen Überblick welche Kategorien schon abgeschlossen sind und welche nicht. Sind alle Tests durchgeführt, kann man sich durch die einzelnen Kategorien und Phasen klicken.
- Das **Backend** beinhaltet alle Tests, welche durchgeführt werden. Diese werden im `VerifyService` in einer Baumstruktur definiert. Um das Frontend immer auf dem neusten Stand zu halten, schickt das Backend laufend Updates.
- Die **Crypto-Bibliothek** ist das Resultat einer Projekt 2 Arbeit. Sie wurde um eine Komponente Verifier erweitert. Dort sind alle Algorithmen für die Tests in Form von Klassen definiert. Zusätzlich wurde in `Utils` noch eine Datei `VerifierHelper.py` erstellt, um redundanten Code auszulagern.

6.1 Technologie

Durch die Wahl die bestehend Applikation «NextGenVote Visualization» mit dem Verifier zu vervollständigen, wurde auch dessen Technologie gewählt. Jedoch wurde im Vorfeld schon das Wahlmodul Python besucht. Und in dem Vertiefungsmodul «Security 3» wurden schon erste Erfahrungen mit *Flask* gemacht. Auch soll *Vue.js* eine weniger steile Lernkurve als zum Beispiel *Angular* haben. Dies führte dann zum Entscheid, die bestehend Applikation zu vervollständigen.

Python kann sowohl prozedural, objektorientiert als auch funktional programmiert werden. *Prozedural* bedeuted, dass einfach eine Anweisung nach der anderen ausgeführt werden. Bei der *funktionalen* Programmierung, werden dann Funktionen definiert, um den gleichen Code an verschiedenen Stellen auszuführen. Bei der *objektorientierten* Programmierung werden sogenannte *Objekte* erstellt. Diese beinhalteten dann Daten und Funktionen. Python ist eine interpretierte Sprache. Das bedeutet, beim Starten wird der Code zuerst in Bytecode verwandelt und erst danach ausgeführt. Der Bytecod kann aber beim ersten ausführen zwischengespeichert werden und muss somit nicht noch einmal generiert werden. Meistens wird aber *CPython* als Interpreter genutzt. Es gibt aber auch zum Beispiel *MicroPython* welches sogar auf einem kleinen ARM Cortex-M4 läuft.

Flask ist ein Web Framework für Python. Es bietet eine schlanke Möglichkeit, um ein Rest-API zu implementieren. Dies wurde dazu verwendet, um Funktionen vom `Voteservice` aus dem Front-End aufzurufen.

Für das Front-End wurde wie schon erwähnt *Vue.js* verwende. *Vue.js* ist eine Clientseitiges Javascript Web Framework, um Single-Page Anwendungen zu schreiben. Es ist nach dem Entwursmuster *ModelView ViewModel (*MVVM) aufgebaut welches eine Variante von *Model View Controller (*MVC) ist. Es wurde sehr einfach gehalten und es reicht, wenn man Kenntnisse von Javascript und HTML hat, um *Vue.js* zu verstehen. Für das State Management wurde die Bibliothek *Vuex* verwendet. Dies bietet die Möglichkeit Daten zentral in einem Store zu speichern. In einem *Store* werden die Zustände der Applikation gespeichert. Ausserdem werden die registrierten Komponenten bei einer Änderung des Stores benachrichtigt. Dieser Store wurde zur Speicherung des Ergebnisses des Verifiers verwendet. Sobald zum Beispiel eine Kategorie einen fehlgeschlagenen Test enthält, wird dies mit roter Farbe angezeigt.

Um den Store im Front-End immer auf den neusten Stand zu halten wurde *Socket.io* verwendet. *Socket.io* ist eine einfache Art, um Daten über das Websocket-Protokoll auszutauschen. Sowohl *Flask* als auch *Vue.js* haben Bibliotheken welche *Socket.io* unterstützen. Auch für Python gibt es eine *Socket.io-client* Bibliothek. Diese wurde für die Konsolenanwendung verwendet.

6.2 Backend

Wie schon erwähnt wurde das Backend mit Python und *Flask* implementiert. Im ersten Schritt wurde die Architektur aufgebaut. Dies gestalte sich aber schwieriger als gedacht. Da die Teststruktur eigentlich ein Baum ist, war es logisch das Kompositionsmuster zu benutzten. Das *Kompositionsmuster* enthält 3 Komponenten. Die Komponente, das Blatt (von einem Baum abgeleitet) und Das Kompositum. Die Komponente ist eine abstrakte Basisklasse. Jede Klasse innerhalb der Hierarchie wird als Komponente behandelt.

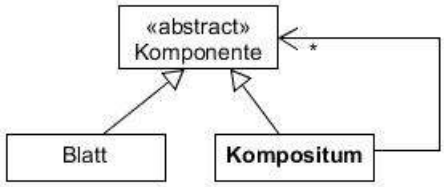


Abbildung 6: UML Kompositionsmuster

Wie man sieht, kann eine Kompositum eine oder mehrere Komponenten enthalten, wird aber selbst als Komponente behandelt. Konkret wurde als Basisklasse eine Klasse *Test* definiert. Als Blatt Komponente wurde die Klasse `SingleTest` definiere und als Kompositum wurde die Klasse `MultiTest` definiert. Damit war es aber noch nicht getan. Denn es gibt auch Tests, welche durch eine Liste von Daten iterieren, aber immer denselben Test durchführen. Für diesen Zweck wurde eine Klasse `IterationTest` definiere welche als `SingleTest` gilt. Doch dann war der `SingleTest` keine Blattkomponente mehr. Also wurde der `SingleTest` auch abstrakt definiert und es entstand für jeden Testalgorithmus eine neu Klasse. Im Klassendiagramm **Abbildung 10** ist dies durch `SampleOfSingleTest` dargestellt. Der Testalgorithmus wurde als Funktion mit dem Namen `runTest` implementiert. Mit dieser Architektur kann nun ein `Multitest` mit Namen `root` erstellt werden, welcher alle Tests enthält. Bei diesem Test kann nun die Funktion `runTest` aufgerufen werden und es werden automatisch bei allen Tests nacheinander die Funktion `runTest` aufgerufen. Dies geschieht in der Klasse `VerifyService`. Dort wird der Testbaum definiert. Sie enthält die Funktion *`verify`* um den Verifier zu starten. Leider sind die Testdaten beim definieren des Baums noch nicht vorhanden. Deshalb wurden sogenannte Schlüsselwörter eingeführt. Die Testdaten werden im *JSON-Format* geliefert. Das *JSON-Format* ist so definiert, dass alle Daten ein solches Schlüsselwort besitzen. In der Variable `keys` werden diese Schlüsselwörter definiert. Somit weiss der Test zur Laufzeit, wie er auf die Testdaten zugreifen kann. Bei den `IterationTests` sind die Testdaten Listen. Ein `IterationTest` Besitz aber ein `Test`, welcher mit jedem Element der Liste ausgeführt wird. Da der Test aber ein Schlüsselwort verlangt, um auf die Testdaten zuzugreifen, musste jedem Element einer Liste ein Schlüsselwort zugewiesen werden. Dazu wurde eine Funktion `prepareData` definiert, welche vor der Funktion `runTest` in der Funktion `verify` aufgerufen wird.

Doch in der Implementationsphase wurde bald klar, dass es auch für das Resultat eine Lösung brauchte. Es wurde also eine Klasse `TestResult` definiert. Doch erst später kam man zum Schluss, dass auch das Resultat eine Hierarchie benötigt. Also bekam jedes *`TestResult`* eine Liste von `TestResults` so konnten auch die Resultate von `Multi`- und `IterationTest` abgebildet werden. Nun mussten aber alle Resultate irgendwie zusammengefasst werden. Dafür wurde eine Klasse `Report` erstellt.

Ein anderes Problem war die Anforderung, dass die GUI jederzeit auf dem neusten Stand sein soll. Zu diesem Zweck wurde das *Beobachtermuster* eingesetzt. Es enthält im Wesentlichen zwei unterschiedliche Klassen. Ein Subjekt und ein Beobachter. Der Beobachter wird aber abstrakt definiert. Somit ist es später möglich verschiedene Beobachter zu haben, welche dasselbe Subjekt haben.

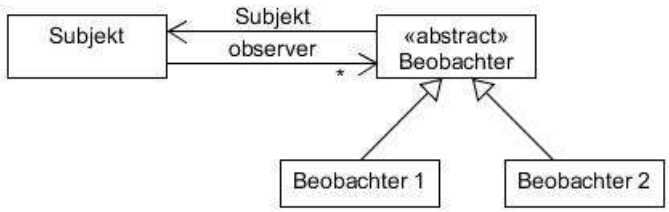


Abbildung 7: UML Beobachtermuster

Das Subjekt enthält eine Funktion `notify`, welches die Funktion `update` des jeweiligen Beobachters aufruft. Als Subjekt wurde die Klasse `Report` gewählt. Dann wurde für die Konsolen-Applikation und *Vue.js* Applikation je eine Klasse erstellt sowie eine Funktion `update` definiert. Mit der Zeit wurde diese Funktion aber sehr gross. Deshalb fiel der Entscheid, die Funktion in kleine Funktionen aufzuteilen. Wie schon erwähnt lässt sich in Python auch funktional programmieren. Das heisst es können auch Funktionen einer Variablen zugewiesen werden. Somit wurde am Ende der Klasse ein Klassenvariable `_functions` definiert, welche ein Dictonary von Namen und Funktion enthält. Der unterstrich wird in Python für *protected* variablen verwendet. Dies ist eine Variable welche nur in der Klasse und in deren Sub-Klassen verwendet werden darf. jedoch hindert dies niemand daran die Variable ausserhalb der Klasse trotzdem mit dem Unterstrich aufzurufen. Man muss dies aber absichtlich machen. Es hindert also nur daran, dass die Variable nicht aus Versehen von aussen aufgerufen wird.

```
_functions = {'testRunning': _testRunning ,
              'newProgress': _newProgress,
              'newResult': _newResult,
              'reportCreated': _reportCreated}
```

Abbildung 8: Variable `_functions` als Dictonary

Nun lässt sich in der Funktion *update* die gewünschte Funktion über den Namen wie folgt aufrufen. In Python werde die Codeblöcke nicht durch geschweifte Klammern wie in Java sondern durch einrücken definiert. Dabei ist es wichtig das immer vier Leerschläge pro Level eingerückt wird.

```
def update(self,state):
    func = self._functions[state]
    func(self)
```

Abbildung 9: Update Funktion der View-Klasse

Die Variable `state` enthält den Namen der gewünschten Funktionen. Nun wird aus dem Dictionary die passende Funktion geholt und in die Variable `func` geschrieben. Danach lässt sich die Funktion mit dem Parameter `self` in der Klammer aufrufen. Die Variable `self` ist eine Referenz auf des aufgerufenen Objekt.

In einem ersten Ansatz enthielt der Report eine Variable `currentTest`, welche beim Start des Tests über einen Setter neu gesetzt wurde und erhält dessen Resultat nach Abschluss. Somit konnte der Report bei einem neuen Status des Tests den Beobachter informieren. Dies führte dazu, dass der Report der Funktion `runTest` übergeben werden musste. Nach einiger Zeit stellte man fest, dass es einfacher wäre, wenn das `TestResult` über Statusänderungen der Tests informieren würde und der Report nur noch am ende der Funktion *`verify`* informiert, wenn er den Report erstellt hat. Somit musst der Report auch nicht mehr der Funktion `runTest` übergeben werden, sondern nur noch der Funktion `verify` .

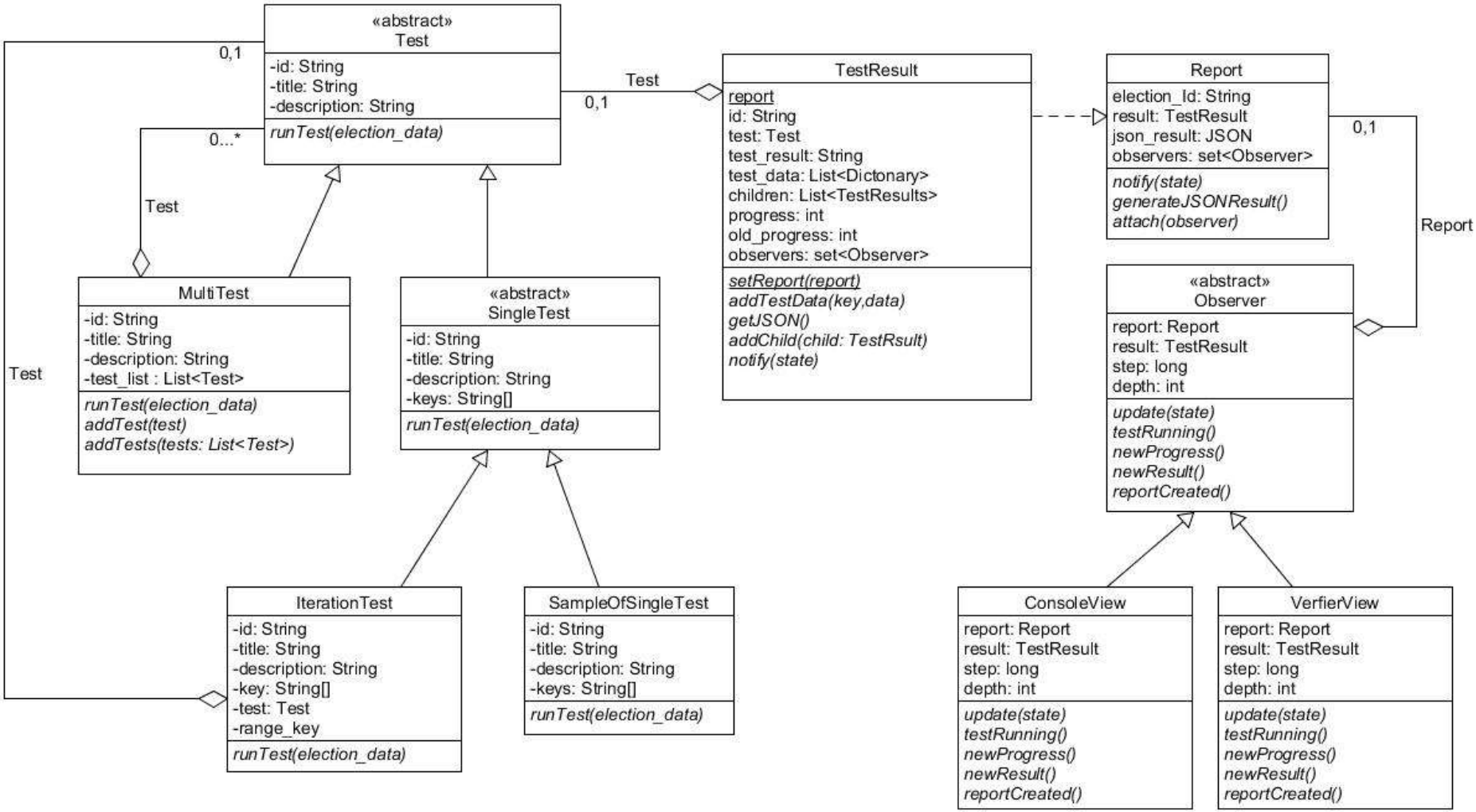


Abbildung 10: Klassendiagramm des Backends

6.2.1 VerifyService

Der VerifyService ist die Grundkomponente des Verifiers. Dort wurden alle Testalgorithmen importiert. Danach wurde ein Testbaum nach einem Testkatalog der Projektarbeit 2 implementiert. Jeder Test bekam eine eindeutige Identifikationsnummer, ein Titel, eine Beschreibung und wie schon erwähnt, die Schlüsselwörter, um auf die Testdaten zuzugreifen.

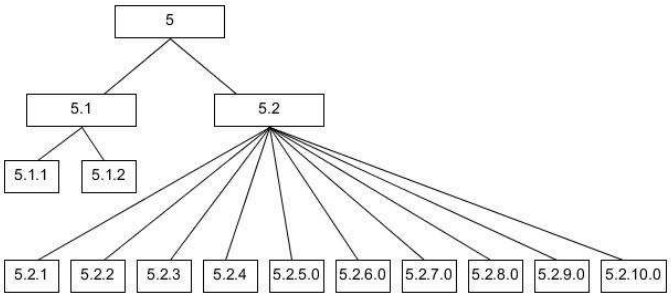


Abbildung 11: Testbaum von Authentizitätstests

Als Beispiel hier der Baum des Authentizitätstests. Auf der ersten Stufe ist die Kategorie als `MultiTest` implementiert. Auf zweiter Stufe die Unterkategorien auch *Phasen* gennant, auch als `MultiTest` implementiert. In diesem Fall ist die Erste Unterkategorie Zertifikate und die zweiten Signaturen. Ist die letzte Ziffer der Identifikationsnummer eine null, so handelt es sich um eine `IterationsTest`. Die letzte Ziffer wird dann bei jedem Durchgang hochgezählt. Bei alle anderen Blattkomponenten handelt es sich um `SingleTests`. Es kam aber auch vor das der `IterationTest` ein `MultiTest` enthielt. Dann wurde auch beim `MultiTest` die Identifikationsnummer hochgezählt. Die einzelnen Kategorien wurden dann an den Wurzelknoten gehängt, welcher dann auch ausgeführt wurde.

Beim Kreieren eines VerfyService-Objekts, wurde auch der gesamte Baum erstellt. Damit Dieser Baum nicht bei jedem Aufruf des Verifiers neu erstellt werden musste, wurde der `VerfyService` als *Singelton* implementiert. Bei diesem Entwurfsmuster wird sichergestellt, dass nur ein Objekt dieses Types vorhanden ist.

```
def __init__(self):
    """ Check if an instance is already created """
    if VerifyService.__instance != None:
        raise Exception("This class is a singleton!")
    else:
        VerifyService.__instance = self
        self.root_test = MultiTest("0:","Root Test","Test which conntains all Tests")
        self.setUpTests()
```

Abbildung 12: Funktion `intit` von `VerfyService`

Konkret wurde dafür eine Klassenvariable `__instance` erstellt und mit None initialisiert. In Python ist None auch ein Singelton, welches aber kein Wert und keine Funktionen hat. Nun wird in der Funktion `__init__` geprüft ob die Variable `__instance` von None verschieden ist. Ist dies nicht der Fall, wird `self` der Variable `__instance` zugewiesen. Danach wird ein neuer Wurzelknoten und mit `setUpTests` ein neuer Baum mit allen Tests erstellt. Falls `__instance` verschieden von None ist, wird ein Fehler ausgegeben.

Die Funktion `getInstance` gibt entweder eine Neues VerfyService-Objekt zurück oder falls `__instance` nicht None ist, wird `__instance` zurückgegeben.

Der doppelte Unterstrich bedeutet das die Variable privat ist. Das heisst, die Variable darf nur innerhalb der Klasse verändert oder deren Wert abgefragt werden. Um zu verhindern, das ausserhalb der Klasse auf diese Variable zugegriffen werden kann, verfügt Python über ein so genanntes *name mangeling*. Das heisst es setzt vor die Variable noch den Klassennamen. Die Variable `__instance` müsste also mit `_VerifyService.__instance` abgefragt werden. Dies sollte man aber auf keinen Fall tun. Denn dafür gibt es die Funktion `getInstance`.

```
@staticmethod
def getInstance():
    """ Static access method. """
    if VerifyService.__instance == None:
        VerifyService()
    return VerifyService.__instance
```

Abbildung 13: Funktion *`getInstace`* von `VerfyService`

6.2.2 Python Decorators

Beim Erstellen der Testklassen wurde festgestellt, dass die Funktion `runTest` am Anfang und am Schluss der Funktion immer die gleichen Codezeilen ausführen. Deshalb wurden diese Zeilen in sogenannte Decorators, ein weiteres Konzept von Python, ausgelagert. Es wurden zwei solche Decorators implementiert. Einer, für das Starten und Beenden der Tests, denn dann muss der Beobachter informiert werden. Ein anderer, um zu prüfen ob die Testdaten vorhanden sind oder nicht. Denn dann, musste der Test gar nicht ausgeführt werden und das Test Resultat ist direkt «skipped» Dies ist Beispielsweise bei den Signaturen der Fall. Denn diese sind nicht implementiert. Auch hier wird wieder der funktionale Ansatz gewählt. Denn eine Funktion kann nicht nur in Variable gespeichert werden, es kann auch Funktion in einer Funktion definiert und zurückgegeben werden. In der **Abbildung 14** wird eine Funktion `compose_greet_func` definiert, welche eine Funktion `get_message` zurückgibt. Die Funktion `compose_greet_func` wird nun mit dem Parameter «John» aufgerufen und in der Variable `greet` gespeichert. Der Output von `greet()` is dann «Hello there John!»

```
def compose_greet_func(name):
    def get_message():
        return "Hello there "+name+"!"

    return get_message

greet = compose_greet_func("John")
print(greet())

# Outputs: Hello there John!
```

Abbildung 14: Beispiel Funktion in Funktionen, Quelle: [6]

Übergibt man nun diesem Konzept eine Funktion, lässt sich vor oder nach der Funktion Code ausführen. Somit kann man vor der Funktion `runTest` dem Beobachter mitteilen, dass nun ein neuer Test gestartet wird und danach dem Beobachter das Resultat des Tests mitteilen. Ausserdem kann man so auch vor dem Start prüfen ob die benötigten Daten vorhanden sind.

In Abbildung 16 sieht man nun den ersten *Decorator*. Dieser erstellt ein neues `TestResult`. Dies bewirkt das der Beobachter über einen neun Test informiert wird. Damit der Test auf das Resultat in der Funktion `runTest` zugreifen kann, wird in diesem Test eine neue Instanzvariable `test_result` erstellt. In Python ist es nämlich möglich Instanzvariablen zur Laufzeit hinzuzufügen. Dies wäre zum Beispiel in Java nicht möglich. Danach wird die Funktion `func` ausgeführt. Die Variable `func` enthält die Funktion `runTest` des jeweiligen Tests. Zum Schluss wird der Rückgabewert der Funktion `func` in das TestResult geschrieben und dessen Variable `progress` auf 1 gesetzt. Dies bewirkt das der Beobachter darüber informiert wird, dass der Test nun abgeschlossen ist.

```
def test_run_decorate(func):
    def func_wrapper(self,election_data):
        result = TestResult(self,"empty Result",None)
        self.test_result = result
        res = func(self,election_data)
        result.test_result = res
        result.progress = 1
        return result
    return func_wrapper
```

Abbildung 16: Decorator ohne Überprüfung der Daten

Decorators können auch verschachtelt werden. Dies wurde beim zweiten Decorator in der **Abbildung 17** genutzt. Hier sieht man auch gerade wie ein Decorator aufgerufen wird. Python hat dafür eine Anotation. Oberhalb der Funktion wird mit @<Funktionsname> definiere welche Funktion aufgerufen werden soll. In diesem Fall die Funktion `test_run_decorate` der **Abbildung 16**. Dieser wird dann die Funktion `func_wrapper` übergeben. Die Funktion completness_decorate enthält in der Variable `func` die Funktion `runTest` des jeweiligen Tests. Diese wird nur aufgerufen, wenn die Testdaten vorhanden sind. Sonst wird «skipped» zurückgegeben.

```
def completness_decorate(func):
    @test_run_decorate
    def func_wrapper(self,election_data):
        com_test = completness_test(self,election_data)
        if com_test:
            return func(self,election_data)
        else:
            return "skipped"
    return func_wrapper
```

Abbildung 17: Decorated Decorator

6.2.3 Python Doctest

Für das Testing der einzelnen Algorithmen wurde Python Doctest verwendet. Es erlaubt gleichzeitig zu Dokumentieren und zu Testen. Python bietet mit Doctest eine sehr komfortable Art an, seinen Code zu testen. Der sogenannte *Docstring* bietet dabei die Basis. Einen *Docstring* wird mit 3 Anführungszeichen gestartet und wieder mit 3 Anführungszeichen beendet.

```
"""
>>> res = sct.runTest({'test':123})
>>> res.test_result
'successful'
>>> sct.test_result.test_data
[{'test': 123}]
>>> res = sct.runTest({'bla':123})
>>> res.test_result
'failed'
>>> sct.test_result.test_data
[]
"""
```

Abbildung 18: Beispiel Doctest

Mit den 3 spitzen Klammern wird eine Anweisung gestartet. Werden keine spitzen Klammern geschrieben, wird eine Ausgabewert erwartet. In unserm Beispiel wird als erstes `runTest` ausgeführt und ein `TestResult` in die Variable `res` geschrieben. Danach wird das Resultat vom `TestResult` abgefragt und das soll «successful» ergeben. So wird Zeile für Zeile getestet.
Damit ein Doctest gestartet wird, muss `doctest` importiert werden. Dann werden alle erforderlichen Module importiert. Und mit der Funktion `testmod` wird der Test gestartet. In der Funktion `testmod` gibt es die Möglichkeit ein globales Objekt für die Tests zu definiere. Für diesen Test wird ein `SingleCompletnessTest` Objekt benötigt, welches den Namen `sct` trägt. Dieses Objekt benötigt folgende Parameter. Eine eindeutige Nummer, ein Titel, eine Beschreibung und mindestens ein Schlüsselwort. Es können auch mehrere Schlüsselwörter angegeben werden. Dann wird mit dem ersten ein Dictionary geholt und mit dem nächsten den Wert mit dem dazugehörigen Schlüsselwort. Beispielsweise ist ein Ballot, welche eine Stimme enthält, ein Dictornary mit 3 Werten. Ruft man nun diese Datei in der Konsole auf, werden die Tests automatisch gestartet. Wenn keine Ausgabe erfolgt wurden alle Tests erfolgreich abgeschlossen. Ansonsten wird angezeigt welcher Test fehlgeschlagen ist.

```
if __name__ == '__main__':
    import doctest
    from chvote.verifier.TestResult import TestResult
    from app.verifier.Report import Report
    TestResult.setReport(Report("1"))
    doctest.testmod(extraglobs={'sct': SingleCompletnessTest("1.1","TEST","TEST",["test"])})
```

Abbildung 19: Main Funktion um Doctest zu starten

Beim einführen der Decorators funktionierten auf einmal die Doctests nicht mehr. Denn der Docstring wird nicht automatisch beim Decorator importiert. Zu diesem Zweck gibt es ein Packet `functools` mit einer Funktion `wraps`.

```
def test_run_decorate(func):
    @wraps(func)
    def func_wrapper(self,election_data):
        ...
```

Abbildung 20: wraps Decorator für Doctests

6.2.4 Verifier findet Softwarefehler

Beim Erstellen der Testalgorithmen und den dazugehörigen Doctests. Wurde ein Softwarefehler der vorherigen Arbeit gefunden. Als der `ConfProofIntegrityTest` implementiert war, wurde auch dieser mit einem Doctests getestet. Doch dieser Test schlug immer wieder Fehl. Es wurde festgestellt, dass ein Wert nicht im geforderten Wertebereich ist.

```
pi = self.test_data
res_pi_t = IsMemberOfGroupe(mpz(pi[0]),param.p_hat)
res_pi_s = int(pi[1]) in range(param.q_hat) # wasn't in q_hat
return 'successful' if res_pi_t and res_pi_s else 'failed'
```

Abbildung 21: Not in $Z_{\hat{q}}$

Der Wert `pi[1]` war nicht in $Z_{\hat{q}}$. Nach Spezifikation müsste dieser Wert bei Sicherheitslevel 3, nicht grösser sein als 8.31713535784724e+76 sein. Der Testwert war aber 1.3297466462711527e+77. Zusammen mit dem Betreuer wurde dann festgestellt. Das beim Erstellen der Bestätigungs-Beweise, ein Fehler unterlaufen ist.

```
# Changed by Christian Wenger 21.12.2018 was not in q_hat because the second brackets was missed
# s = w + c * (y + y_prime) % secparams.q_hat
s = (w + c * (y + y_prime)) % secparams.q_hat
```

Abbildung 22: Softwarefehler

Wie im Kommentar zu sehen wurde eine Klammer vergessen. Somit ist zwar der Wert $s * (y + y')$ in $Z_{\hat{q}}$, jedoch wurde dann noch w dazu addiert, was den s Wert wieder verändert.

6.3 Frontend

Wie schon erwähnt wurde das Frontend mit dem Javascript Web-Framework *Vue.js* umgesetzt. Da in der ersten Phase einen Prototyp erstellt wurde, war die Applikation schon ziemlich genau definiert. Dies wurde auch so weit wie möglich umgesetzt.

6.3.1 Komponente

Eine *Vue.js* Applikation besteht aus sogenannten Komponenten. Die Applikation ist eine Komponente, jede Seite ist eine Komponente und die Seiten enthalten meist auch noch Komponenten. Es ist auch möglich seine eigene Komponente zu definieren. Dies ermöglicht es wiederverwendbaren Code zu schreiben.

Wie man in der **Abbildung 23** sieht wurden im Frontend einige Python Klassen gespiegelt. Somit lässt sich dessen Resultat sehr komfortable anzeigen. Die `VerifierCategory` Komponente Ist für das Anzeigen der 5 Kategorie zuständige. Aus einer einzelnen Kategorie wird dann die Unterkategorie als Komponente `Tab` dargestellt. Die Komponente `Tab` ist Bestandteil des *Vuetify-Moduls*, welches wir später noch im Detail anschauen werden. In der `VerifierPage` wird dann in jedem `Tab` eine Komponente `ResultTree` gerendert. Die Komponente `ResultTree` zeigt dann je nach Art des Resultats ein `Single`-, `Multi`- oder `IterationResult` an. Die `Result` Komponente entspricht also einem Resultat des `SingelTest`.

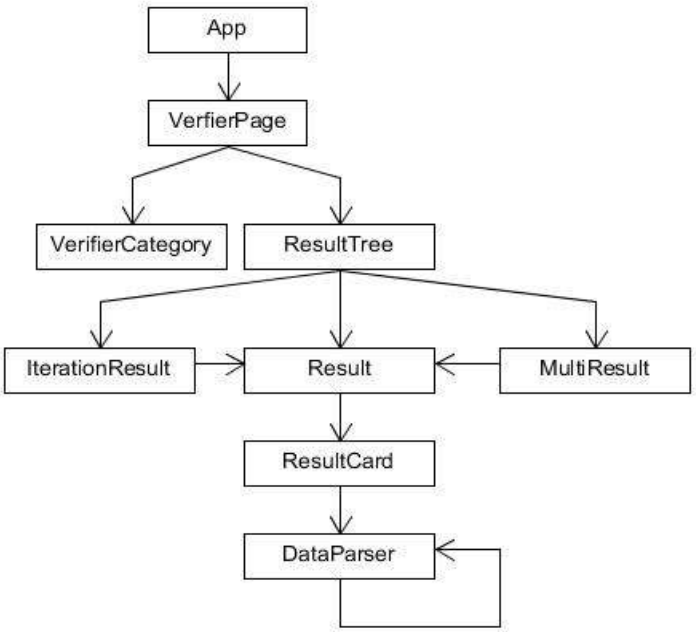


Abbildung 24: Überblick des Frontends

Vuetify ist eine Bibliothek mit vorgefertigten *Vue.js* Komponente nach dem Material Desing. Das Modul lässt sich mit `npm` installieren und muss dann in der `main.js` importiert werden. Nun kann von jeder Komponente aus auf die Komponenten von *Vuetify* zugegriffen werden. *Vuetify* untersütz das CSS-Framework *Bootrsap*. Dies bietet die Möglichkeit sehr einfach responsive Webseiten zu programmieren. Die Seite wird in 12 Spalten aufgeteilt. Je nach Grösse des Bildschirms, lässt sich dann bestimmen wie viele Spalten ein Element benutzten darf.
In *Vuetify* wurden die `div` Elementen mit verschieden CSS-Klassen, durch Komponente wie `v-layout` oder `v-flex` ersetzt. Folgende *Vuetify* Komponenten wurden genutzt [5].

- **Buttons:** Die `v-btn` Komponent ersetz das HTML-Element `button`, mit einem Button im Material Desing. Sie Wurde Beispielsweise für die Komponente `VerfierCategory` verwendet. Sie bietet verschiedene Eigenschaften, um beispielsweise Farbe oder Form der Komponente zu ändern. Wird zu Beispiel die Eigenschaft `fab` angegeben, lässt sich sehr einfach einen runden Button realisieren.
- **Cards:** Die `v-card` Komponente wurde in der ResultCard Komponente genutzt, um die Details der Komponente `Result` anzuzeigen. Sie eignet sich sehr gut, um viele Informationen übersichtlich anzuzeigen.
- **Data Iterator:** Die Komponente `v-data-iterator` wurde in der Komponente `InterationResult` verwende. Sie wurde genutzt, um sich durch den `IterationTest` klicken zu können. Links und rechts wurden eigen Pfeile hinzugefügt, um zum nächsten oder vorherigen Resultat zu kommen.

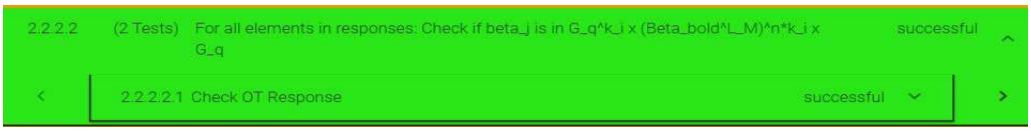


Abbildung 25:Die Komponente `IterationResult`

- **Expansion panels:** Die Komponente `v-expansion-panel` wurde in der Komponente `Result` genutzt. Sie besteht aus zwei Teilen, einem Header welcher immer sichtbar ist, und einem ausklappbaren Teil.
- **Progress:** Die `v-progress-linear` Komponente wurde in der Komponente `VerifierPage` genutzt, um den Fortschritt des Verifier anzuzeigen
- **Tabs:** Die Komponente `v-tab` wurde in der Komponente `VerifierPage` verwendet, um die Unterkategorie hinter Tabs zu verstecken. Somit kann nun zwischen den Tabs hin und her gewechselt werden. Dies bewirkt eine bessere Übersicht.

Eines der nützlichsten Features ist das Slot-Konzept in *Vue.js*. Damit lässt sich zur Laufzeit Komponenten in eine Komponente einfügen. Dies wurde Beispielsweise in der Komponente `Result` verwendet.

```
<template>
  <v-expansion-panel>
    <v-expansion-panel-content v-bind:style="{ 'background-color': getColor(result.value), border: '1.5px solid' }">
      <div slot="header">
        <v-layout row wrap>
          <v-flex mr-3 xs1 md1>{{result.id}}</v-flex>
          <v-flex xs1 md1 v-if="getTotalChildren(result) != 0">({{getTotalChildren(result)}} Tests)</v-flex>
          <v-flex xs8 md8>{{result.title}}</v-flex>
          <v-flex mx-3 text-xs-right>{{result.value}}</v-flex>
        </v-layout>
      </div>
      <slot></slot>
    </v-expansion-panel-content>
  </v-expansion-panel>
</template>
```

Abbildung 26: `Template` der `Result` Komponente

Dies ist das HTML Gerüst der Komponente. Wie man sieht wird hier ein `v-expansion-panel` gerendert. Alles was nicht im Slot Header definiert ist, wird erst beim Ausklappen angezeigt. Nun wurde aber noch explizit ein `slot` Element definiert. Dort wird alles angezeigt was innerhalb der Komponente `Result` definiert wurde. Hier sieht man auch die Eigenschaften der Komponenten in Orange. Mit `v-bind` oder kurz nur Doppelpunkt, lässt sich ein Wert an diese Eigenschaft binden. Verändert sich der Wert, so verändert sich auch der Wert dieser Eigenschaft.

```
<template>
  <v-flex>
    <v-flex v-for="result in results" :key="getId(result.id)">
      <result :result="result">
        <result-card v-if="getTotalChildren(result) === 0" :result="result"></result-card>
        <v-flex v-if="getTotalChildren(result) > 0">
          <iteration-result :result="result" v-if="getTotalChildren(result.children[0]) === 0"></iteration-result>
          <multi-result :result="result" v-if="getTotalChildren(result.children[0]) > 0"></multi-result>
        </v-flex>
      </result>
    </v-flex>
  </v-flex>
</template>
```

Abbildung 27: Template der ResultTree Komponente

Hier Sieht man, das innerhalb der Komponente `Result` entweder eine Komponente `ResultCard`, `IterationResult` oder `MultiResult` gerendert wird. Mit dem `v-if` lässt sich bestimmen ob eine Komponente gerendert werden soll oder nicht. Das Element `slot` wird also zur Laufzeit mit einer der Komponenten ausgetauscht.

Nebst dem Template können Komponente auch Daten, Eigenschaften und Methoden enthalten. Dies wird dann innerhalb der Script-Elemente definiert.

```
<script>
import getTotalChildrenMixin from '../../mixins/getTotalChildrenMixin.js'
export default{
  data: () => ({
    pagination: {
      rowsPerPage: 1,
      page: 1
    }
  }),
  mixins: [getTotalChildrenMixin],
  props: {
    result: {
      type: Object,
      required: true,
      default: {}
    }
  },
  methods: {
    next: function () {
      this.pagination.page++
    },
    previous: function () {
      this.pagination.page--
    }
  }
}
</script>
```

Abbildung 28: Javascript von IterationResult

Hier sieht man das Javascript der Komponente `IterationResult`. In *Vue.js* besteht das Javascript im Wesentlichen aus einem Objekt. Dieses kann folgende Attribute enthalten.

- **data:** Dieses Attribut enthält die Daten der Komponente. In diesem Beispiel ein Objekt `pagination`.
- **props**: Mit dem Attribut `props` werden Eigenschaft der Komponenten definiert, welche zur Laufzeit übergeben werden. In diesem Beispiel verlangt die Komponente eine Eigenschaft `result`.
- **methods:** Hier werden die Methoden definiert, um mit der Komponente zu interagieren. In diesem Beispiel `next` und `previous`, um zum nächsten oder vorherigen `Result` zu gelangen
- **mixins:** Das `mixins` Attribut ist etwas speziell. Damit lässt sich Teile des *Vue.js* Objektes importieren. Hier wird die Methode `getTotalChildren` über ein `mixin` eingefügt. Somit muss diese Methode nur einmal definiert werden und kann dann in allen Komponenten, welche die Methode benötigen, importiert werden.

```
export default {
  methods: {
    getTotalChildren: function (result) {
      return result.children.length
    }
  }
}
```

Abbildung 29: Beispiel `Mixins`

Nebst den oben erwähnten Attributen wurden noch zwei weitere verwendet.

- **created:** Mit dem Attribut `created` lässt sich eine Funktion beim Erstellen der Komponente ausführen. Wie schon erwähnt, kam es vor, dass ein `IterationResult` eine `MultiResult` als Test enthielt. Nun wurde Bei erstellen der Komponente aus den Tests des `MultiResults` je ein `IterationResult` mit dessen Kindern erstellt.

```
created: function () {
  this.result.children[0].children.forEach((result) => {
    this.iterResult.push(this.createResult(result))
  })
}
```

Abbildung 30: Created Attribut von der Komponent Multiresult

- **Computed:** Im Attribut `computed` lassen sich auch Funktionen definiere. Jedoch im Gegensatz zu den `methods` werden `computed` Funktionen nur aufgerufen, wenn sich einer der Werte in der Funktion geändert hat.

```
computed: {
  getCategory: function () {
    return this.$store.state.Verifier.categories[this.category_id - 1]
  },
  ...
}
```

Abbildung 31: computed Attribut von der Komponente VerfierCategory

6.3.2 Rekursive Komponente

Komponente können auch rekursiv aufgerufen werden. Dies wurde in der Komponente `DataParser` genutzt.

```
<template>
  <v-flex>
    <v-flex v-if='isPrimitiv(value)'>
      <BigIntLabel v-if='isBigInt(value)' :mpzValue="value.toString()"></BigIntLabel>
      <ByteArrayLabel v-if='!isBigInt(value)' :value="value.toString()"></ByteArrayLabel>
    </v-flex>
    <v-flex v-else>
      <v-flex v-if='isArray(value)' v-for='(val) in value'>
        <v-layout row-wrap >
          <data-parser :value="val"></data-parser>
        </v-layout>
      </v-flex>
      <v-flex v-else v-for='(val,key) in value'>
        <v-layout row-wrap>
          {{key}}:<data-parser :value="val"></data-parser>
        </v-layout>
      </v-flex>
    </v-flex>
  </v-flex>
</template>
```

Abbildung 32: Template der DataParser Komponente

Die Komponente `DataParser` wurde genutzt, um die grossen Testdaten darzustellen. Leider wusste man nicht, ob das `value` einen primitiven Datentyp, wie Integer oder String ist, oder ob es sich um ein Array oder Objekt handelt. Wenn es sich nicht um einen primitiven Datentyp handelte, wurde die Komponente `data-parser` nochmal mit `val` aufgerufen. Der Wert `val` konnte aber auch wieder ein Array oder ein Objekt sein. Deshalb wurde die Komponente `data-parser` so lange aufgerufen, bis es sich beim `value` um einen primitiven Datentyp handelt.

6.3.3 Vuex

Vuex ist eine Bibibliothek für *Vue.js* und ermöglicht es eine Zentralen Datenspeicher zu haben. Diesen Datenspeicher wird in Vuex Store genannt. Dies wurde zur Kommunikation mit dem Backend verwendet. Immer wenn über *Socket.io* neue Daten gesendet werden, wird der Store aktualisiert. Die registrierten Komponenten werden dann über eine Änderung benachrichtigt und können darauf regieren. In **Abbildung 30** wird zum Beispiel immer, wenn sich die Kategorie ändert, ein neuer Wert geliefert. Dies funktioniert, weil die Werte als `computed` übergeben werden.

Da der Store solange besteht bis die Applikation neu gestartet wird, musste ein Startwert definiert werden. Denn der Store soll bei jedem Aufruf der Verfier-Seite, auf den Startwert zurückgesetzt werden. Zur besseren Übersicht wurde der Store in verschiedene Module unterteilt. Jeder Teilnehmer des E-Voting Protokolls wurde ein Modul im Store zugeteilt.

```
const getDefaultState = () => {
  return {
    categories: [
      {id: 1, title: 'Completness', state: 'idle', value: '', results: []},
      {id: 2, title: 'Integrity', state: 'idle', value: '', results: []},
      {id: 3, title: 'Consistency', state: 'idle', value: '', results: []},
      {id: 4, title: 'Evidence', state: 'idle', value: '', results: []},
      {id: 5, title: 'Authenticity', state: 'idle', value: '', results: []}
    ],
    currentResults: [],
    currentTest: '',
    progress: 0,
    completed: false
  }
}
```

Abbildung 33: Startwert für den Store in `Verifier.js`

```
const getDefaultState = () => {
  return {
    categories: [
      {id: 1, title: 'Completness', state: 'idle', value: '', results: []},
      {id: 2, title: 'Integrity', state: 'idle', value: '', results: []},
      {id: 3, title: 'Consistency', state: 'idle', value: '', results: []},
      {id: 4, title: 'Evidence', state: 'idle', value: '', results: []},
      {id: 5, title: 'Authenticity', state: 'idle', value: '', results: []}
    ],
    currentResults: [],
    currentTest: '',
    progress: 0,
    completed: false
  }
}
```

Abbildung 34: Startwert für den Store in `Verifier.js`

Folgende Elemente kann ein Store enthalten.

- **State:** Die Variable `state` enthält alle Daten des Moduls
- **mutations**: In der Variable `mutations` werden Funktion definiert, um die Daten des Moduls zu verändern. *Socket.io* kann in Vuex integriert werden. Sobald eine Funktion mit Präfix «SOCKET» definiert wurde, hört *Socket.io* automatisch auf einen Event mit dessen Namen. In diesem Fall auf `testRunning`

```
SOCKET_TESTRUNNING: (state, title) => {
  state.currentTest = title
  console.log('currentTest:', title)
},
```

Abbildung 35: Beispiel Socket.io Funktion, welche auf den Event testRunning hört.

- **getters:** Wenn die Werte, beim Abfragen, formatiert, aggregiert oder gefiltert werden müssen, ist es sinnvoll eine Funktion in der Variable `getters` dafür anzulegen. Dies wurde verwendet um die Resultate der Kategorie nach all `successful, failed` oder `skipped` zu filtern.

7 Schlussfolgerungen/Fazit

E-Voting ist und bleibt ein umstrittenes Thema. Nichtsdestotrotz denke ich, dass es früher oder später eingeführt wird. Denn wenn es gelingt eine Mehrheit der Bürger fürs Abstimmen über das Internet zu gewinnen, wird der Administrative Aufwand für den Staat um einiges abnehmen. Ausserdem denke ich, dass gerade die jüngeren Generationen eher Abstimmen würden, wenn dies übers Internet mögliche wäre. Denn dann würde man sich in einer gewohnten Umgebung aufhalten. Damit dies aber Wirklichkeit wird, muss zwingend nebst dem E-Voting System ein unabhängiger Verifier entwickelt werden. Denn damit wird die Wahrscheinlichkeit eines Wahlbetrugs so klein, dass man ein E-Voting System flächendeckend betreiben kann. Wie man in meiner Arbeit gesehen hat lässt sich damit sogar Softwarefehler aufdecken. Denn leider ist es eine Tatsache, dass Software fehlerhaft ist, egal wie gut der Programmiere auch sein mag. Mit dem Verifier hat man aber noch eine zusätzliche Möglichkeit, um das E-Voting System zu testen.

Nun möchte ich auf einige Aspekte der Bachelor-Thesis eingehen. Ein wichtiger Punkt, den ich gelernt habe ist, dass ein solider Softwaredesign das A und O ist, um erfolgreich eine Applikation zu entwickeln zu können. Leider entschied ich mich anfangs für den einfachsten und schnellsten Weg um möglichst schnell mit der Implementationsphase zu starten. Doch erst da wurde mir bewusst, welche Auswirkungen meine Entscheidungen hatten. Und so musst ich immer wieder meine Applikation neu designen. Dies hatte leider dann auch zur folgen, dass ich meinen Zeitplan nicht einhalten konnte. Schlussendlich fand ich aber dann zusammen mit den Betreuern eine Lösung, welche nun auch im Kapitel **6.2** erläutert wurde. Auch das Implantieren der einzelnen Tests nahm mehr Zeit in Anspruch als ich anfangs dachte. Denn die Testdaten mussten immer zuerst untersucht und bei Bedarf in die richtige Form gebracht werden. Gerade bei den Kryptographischen Beweisen musste sichergestellt werden, dass die Die Testaten den Richtigen Datentyp besitzen. Nicht selten wurden die Zahlen als Strings geliefert und mussten in `Multi-precison Integers (mpz)` umgewandelt werden. Oder es musste ein Beweisobjekt aus den richtigen Daten erstellt werden. Im Gensatz zu Denzer und Hänni [4, S. 47], war ich sehr zufrieden mit Python. Denn ich denke nicht, dass es mit einer Sprache wie Java möglich gewesen wäre, ein solch schlanken Verifier zu schreiben. Funktionale Konstrukte wie die Decorators wären in Java nicht möglich gewesen. Der Grund könnte darin liegen, dass ein Verifier ein viel kleineres Projekt ist, als ein Ganzes E-Voting System. Ich kann mir durchaus vorstellen, dass Python für grosse Projekte nicht geeignet ist. Durch das Entwickeln des Verfiers wuchs ich an Erfahrungen. Ich lernte wie mächtig die Sprache Python sein kann und konnte ein aktuelles Web-Framwork in vollen Zügen testen. Meiner Meinung nach hält *Vue.js* was es verspricht. Denn ich kam sehr schnell damit zurecht. Allerdings würde wohl ein erfahrener Programmierer, einiges anders machen als ich. Dieses Projekt gab mir einen ersten Einblick in die Zukunft des E-Votings. Ich hoffe ich konnte ein kleiner Teil beitragen, um E-Voting in Zukunft einführen zu können.

Literaturverzeichnis

[1] „E-Voting.pdf“, 14-Nov-2018. [Online]. Verfügbar unter: https://www.admin.ch/gov/de/start/dokumentation/dossiers/E-Voting.html.

[2] René Lenzin, „Faktenblatt_DE.pdf“, Juli 2018.

[3] E. Dubuis, R. Haenni, R. E. Koenig, und P. Locher, „Definition der universellen Verifizierung für Internetwahlen“.

[4] K. Häni und Y. Denzer, „Visualizing Geneva’s Next Generation E-Voting System“.

[5] „Vue.js Material Component Framework — Vuetify.js“. [Online]. Verfügbar unter: https://vuetifyjs.com/en/. [Zugegriffen: 02-Jan-2019].

[6] A. Farhat, „A guide to Python’s function decorators“, *The Code Ship*, 06-Jan-2014. [Online]. Verfügbar unter: https://www.thecodeship.com/patterns/guide-to-python-function-decorators//. [zugegriffen: 04-Jan-2019].

Anhang A

In diesem Kapitel werden alle Daten aufgeführt welche später für die Tests im Testkatalog verwendet werden. Jeder, zu prüfende Parameter verfügt über ein eigenes Zeichen, eine Beschreibung und einen Wertebereich. Auch hier diente die Spezifikation als Grundlage.

1. Vordefinierte Parameter

Die nachfolgenden Daten sind nicht Teil des Bulletin Boards. Sie spielen für die Sicherheit des Systems jedoch eine entscheidende Rolle. Sie müssen auch nicht für jeden Wahlgang neu gewählt werden. Entscheidend sind die drei Parameter σ, τ und ε.
Der Parameter σ definiert wie viel Rechenpower nötig ist, damit ein Angreifer in polynomialer Zeit die Privatsphäre einer Stimme brechen könnte.
Der Parameter τ definiert wie viel Rechenpower nötig ist, damit ein Angreifer in gleicher Weise die Integrität der Stimme brechen könnte.
Der Parameter ε definiert die Wahrscheinlichkeit, dass eine Manipulation am System von einem aufrichtigen Teilnehmer entdeckt wird. Alle anderen Parameter können von diesen dreien abgeleitet werden. Zusätzlich habe ich noch die nötigen Zertifikate und dessen öffentliche Schlüssel für die Signaturen in dieser Tabelle aufgeführt. Denn auch diese müssen nicht bei jedem Wahlgang neu erstellt werden

Parameter	Description	Range
p	Modulo of encryption group $\mathbb{G}_q$	$p \in \mathbb{S}$
g, h	Independent generators of $\mathbb{G}_q$	$g, h \in \mathbb{G}_q \setminus \{1\}$
$\hat{p}$	Modulo of identification group $\mathbb{G}_{\hat{q}}$	$\hat{p} \in \mathbb{P}$
τ	Minimal integrity (bits)	$\tau \in \{4, 80, 112, 128\}$
σ	Minimal privacy (bits)	$\sigma \in \{4, 80, 112, 128\}$
λ	Security level	$\lambda \in \{0, 1, 2, 3\}$
ε	Deterrence factor	$\varepsilon \in \{0.99, 0.999, 0.9999, 0.99999\}$
ℓ	Hash lenght (bits)	$\ell \in \{8, 160, 224, 256\}$
L	Hash lenght (bytes)	$L \in \{1, 20, 28, 32\}$
L_M	Length of OT messages (bytes)	$L_M \in \{2, 40, 56, 64\}$
L_F	Length of finalization codes F_i (bytes)	$L_F \in \{1, 2, 3\}$
$\hat{q}$	Order of $\mathbb{G}_{\hat{q}}$	$\|\|\hat{q}\|\| \geqslant 2\tau$
$\hat{g}$	Generator of $\mathbb{G}_{\hat{q}}$	$\hat{g} \in \mathbb{G}_{\hat{q}} \setminus \{1\}$
p'	Modulo of prime field $\mathbb{Z}'_p$	$\|\|p'\|\| \geqslant 2\tau$
$\hat{q}_x$	Upper bound of secret voting credential x	$\|\|\hat{q}_x\|\| \geqslant 2\tau$
$\hat{q}_y$	Upper bound of secret confirmation credential y	$\|\|\hat{q}_y\|\| \geqslant 2\tau$
n_{max}	Maximal number of candidates	$n_{max} \geqslant 2$
(pk_{Admin}, C_{Admin})	Public key and certificate of election administrator	$pk_{Admin} \in \mathbb{G}_q$, $C_{Admin} \in$ X.509
$((pk_{Auth_j}, C_{Auth_j}))$	Public key and certificate of election authority j	$pk_{Auth_j} \in \mathbb{G}_q$, $C_{Auth_j} \in$ X.509, $j \geqslant 1$

2. Das Bulletin Board

Die nachfolgenden Daten sind als öffentlich zu betrachten. Wie schon erwähnt kann jedoch definiert werden, wer auf welche Daten Zugriff bekommt. Wie der Namen schon ahnen lässt, kann man sich das Bulletin Board wie eine Pinnwand vorstellen. Jedoch dürfen nur die Autoritäten, Administratoren und Wähler schreibend darauf zugreifen. Bei allen andern ist es zwingend nötig das sie nur Leserechte haben.

Einer der wichtigsten Parameter ist die ID des Wahlgangs. Denn nur damit lässt sich sicherstellen, dass es keine Vermischungen der Wahlgang Daten gibt. Denn ohne könnte man denn Daten nicht ansehen zu welchem Wahlgang sie gehören. Deshalb muss diese ID mit jeder Nachricht mitgeschickt werden. Ausserdem müssen alle Nachrichten von den Autoritäten und Administratoren signiert werden. So kann man später noch feststellen wer die Nachricht geschickt hat.

Zur Übersicht sind die Daten den drei Phasen der Spezifikation zugeordnet. Diese sind „Vor der Wahl“ (Pre-Election), „Während der Wahl“ (Election) und „Nach der Wahl“ (Post-Election).

Um die Länge von Matrizen darzustellen wird folgende Definition verwendet:

$$\text{For } X = (x_{ij})_{n \times m}, \text{ then } |X| = (n, m)$$

2.1 Vor der Wahl

Parameter	Description	Range
U	Unique election event identifier to protect for election event mismatch.	$U \in \mathbf{A}^*_{\mathbf{ucs}}$
$\mathbf{n} = (n_j)$	Number of candidates in each election	$n_j \geqslant 2,\ \lvert\mathbf{n}\rvert \geqslant 1$
$\mathbf{c} = (C_i)$	Candidate description	$C_i \in \mathbf{A}^*_{\mathbf{ucs}},\ \lvert\mathbf{c}\rvert \geqslant 2$
$\mathbf{k} = (k_j)$	Number of selection in each election	$k_j \geqslant 2,\ \lvert\mathbf{k}\rvert \geqslant 1$
$\mathbf{v} = (V_i)$	Voter descriptions	$V_i \in \mathbf{A}^*_{\mathbf{ucs}},\ \lvert\mathbf{v}\rvert \geqslant 0$
$\mathbf{w} = (\omega_i)$	Assigned counting circles	$\omega_i \geqslant 1, \lvert\mathbf{w}\rvert \geqslant 0$
$\mathbf{E} = (e_{ij})$	Eligibility matrix	$e_{ij} \in \mathbb{B},\ \lvert\mathbf{E}\rvert \geqslant (0, 1)$
$\hat{\boldsymbol{D}} = (\hat{\boldsymbol{d}}_j)$	Public credentials of all voters	$\lvert\hat{\boldsymbol{d}}\rvert \geqslant 0,\ \lvert\hat{\boldsymbol{D}}\rvert \geqslant 1,$
$\mathbf{pk} = (pk_j)$	Public key for encryption	$pk_j \in \mathbb{G}_q,\ \lvert\mathbf{pk}\rvert \geqslant 1$
σ_1^{param}	Signature of full election parameters	$\sigma_1^{param} \in \mathbb{B}^\ell \times \mathbb{Z}_q$
σ_2^{param}	Signature of part of election params	$\sigma_2^{param} \in \mathbb{B}^\ell \times \mathbb{Z}_q$
σ_3^{param}	Signature of other part of params	$\sigma_3^{param} \in \mathbb{B}^\ell \times \mathbb{Z}_q$
$\mathbf{s}_{prep} = (\sigma_j^{prep})$	Signatures of public credentials	$\sigma^{prep} \in \mathbb{B}^\ell \times \mathbb{Z}_q,\ \lvert\mathbf{s}_{prep}\rvert \geqslant 1$
$\mathbf{s}_{kgen} = (\sigma_j^{kgen})$	Signatures of public keys	$\sigma^{kgen} \in \mathbb{B}^\ell \times \mathbb{Z}_q,\ \lvert\mathbf{s}_{kgen}\rvert \geqslant 1$

2.2 Während der Wahl

In dieser Phase werden 4 Listen benötigt. Dabei gilt zu beachten, dass diese unvollständig sein können. Eine Aufgabe besteht also darin, die Gültigen an Hand der ID des Abstimmenden zu finden und zu kontrollieren, ob der Abstimmende in allen Listen genau einen gültigen Eintrag hat. Alle anderen Einträge, dürfen nicht gezählt werden.

Parameter	Description	Range
$\mathbf{A} = \langle(v, \alpha)\rangle$	List of ballots	$\lvert\mathbf{A}\rvert \geqslant 0$
v	Voter id	$v \geqslant 0$
$\alpha = (\hat{x}_v, \mathbf{a}, \pi_\alpha)$	Ballot	$\alpha \in \mathbb{Z}_{\hat{q}} \times (\mathbb{G}_q \times \mathbb{G}_q)^{k'_v} \times ((\mathbb{G}_{\hat{q}} \times \mathbb{G}_q^2) \times (\mathbb{Z}_{\hat{q}} \times \mathbb{G}_q \times \mathbb{Z}_q))$
$\hat{x}_v$	Voter's public credentials	$\hat{x}_v \in \mathbb{G}_{\hat{q}}$
$\mathbf{a} = (a_j)$	Encrypted selection of a voter	$a_j \in \mathbb{G}_q^2$, $\lvert\mathbf{a}\rvert \geqslant 0$
π_α	Proof of validity of ballot	$\pi_\alpha \in (\mathbb{G}_{\hat{q}} \times \mathbb{G}_q^2) \times (\mathbb{Z}_{\hat{q}} \times \mathbb{G}_q \times \mathbb{Z}_q)$
$\mathbf{B} = \langle(v, \beta_j, \sigma_{vj}^{cast})\rangle$	List of OT responses and signature	$\lvert\mathbf{B}\rvert \geqslant 0$
β_j	OT response	$\beta_j \in \mathbb{G}_q^{k'_v} \times (\mathcal{B}^{L_M})^{nk'_v} \times \mathbb{G}_q$
σ_{vj}^{cast}	Signature of OT response	$\sigma_{vj}^{cast} \in \mathbb{B}^\ell \times \mathbb{Z}_q$
$\mathbf{C} = \langle(v, \gamma)\rangle$	Confirmation list	$\lvert\mathbf{C}\rvert \geqslant 0$
$\gamma = (\hat{y}_v, \pi_\beta)$	Confirmation	$\gamma \in \mathbb{G}_{\hat{q}} \times (\mathbb{G}_q \times \mathbb{Z}_{\hat{q}})$
$\hat{y}_v$	Confirmation credential	$\hat{y}_v \in \mathbb{G}_{\hat{q}}$
π_β	Proof knowledge of $y + y'$	$\pi_\beta \in \mathbb{G}_{\hat{q}} \times \mathbb{Z}_{\hat{q}}$
$\mathbf{D} = \langle(v, \delta_j, \sigma_{vj}^{\text{conf}})\rangle$	List of finalization and signature	$\lvert\mathbf{D}\rvert \geqslant 0$
δ_j	Finalizations	$\delta_j \in \mathcal{B}^{L_F} \times \mathbb{Z}_q^2$
$\sigma_{vj}^{\text{conf}}$	Signatures of finalizations	$\sigma_{vj}^{\text{conf}} \in \mathbb{B}^\ell \times \mathbb{Z}_q$

2.3 Nach der Wahl

Parameter	Description	Range
$\mathbf{E}' = (\mathbf{e}_j)$	Mixed and re-encrypted ballot lists	$\mathbf{e}_j \in (\mathbb{G}_q^2)^N,\ N \geqslant 0,\ \lvert\mathbf{E}'\rvert \geqslant 1$
$\boldsymbol{\pi} = (\pi_j)$	Shuffle proofs	$\pi_j \in (\mathbb{G}_q^3 \times \mathbb{G}_q^2 \times \mathbb{G}_q^N) \times (\mathbb{Z}_q^4 \times \mathbb{Z}_q^N \times \mathbb{Z}_q^N) \times \mathbb{G}_q^N \times \mathbb{G}_q^N$ $N \geqslant 0, \lvert\boldsymbol{\pi}\rvert \geqslant 1$
$\mathbf{B}' = (\mathbf{b}'_j)$	Partial decrypted ballot lists	$\mathbf{e}_j \in (\mathbb{G}_q^2)^N,\ N \geqslant 0,\ \lvert\mathbf{B}'\rvert \geqslant 1$
$\boldsymbol{\pi}' = (\pi'_j)$	Decryption proofs	$\pi'_j \in (\mathbb{G}_q \times \mathbb{G}_q^N) \times \mathbb{Z}_q$ $N \geqslant 0, \lvert\boldsymbol{\pi}'\rvert \geqslant 1$
$\boldsymbol{V} = (v_{ij})$	Election result matrix	$v_{ij} \in \mathbb{B}, v_{ij} = \{\ 1 \text{ if vote } i \text{ contains } j\ \}$ $\lvert\mathbf{V}\rvert \geqslant (0, 2)$
$\boldsymbol{W} = (\omega_{ij})$	Counting circle matrix	$w_{ij} \in \mathbb{B}, \omega_{ij} = \{\ 1 \text{ if } i \text{ is assigned to } j\ \}$ $\lvert\mathbf{W}\rvert \geqslant (0, 1)$
σ^{tally}	Signature of tallying result	$\sigma^{tally} \in \mathbb{B}^\ell \times \mathbb{Z}_q$
$\mathbf{s}_{mix} = (\sigma_j^{mix})$	Signatures of mixed re-encryptions	$\sigma_j^{mix} \in \mathbb{B}^\ell \times \mathbb{Z}_q,\ \lvert\mathbf{s}_{mix}\rvert \geqslant 1$
$\mathbf{s}_{dec} = (\sigma_j^{dec})$	Signatures of partial decryptions	$\sigma_j^{dec} \in \mathbb{B}^\ell \times \mathbb{Z}_q,\ \lvert\mathbf{s}_{dec}\rvert \geqslant 1$

Anhang B

Aus all den zu prüfenden Daten galt es nun einen Testkatalog zu erstellen. Dieser ist in fünf Kategorien unterteilt, welche später im Detail erklärt werden. Das Endergebnis des Verifiers ist also die Summe aus allen fünf Ergebnissen der jeweiligen Kategorie.

Später in der Bachelor-Thesis muss ersichtlich sein. Welcher Test welchem Programm Code entspricht. Aus diesem Grund wurde jedem Test eine Eindeutige Nummer zugewiesen. Damit man diese Später im Programm Code verwenden kann.

Wie auch schon beim Bulletin Board wurden die Tests auch hier teilweise den 3 Phasen der Spezifikation zugewiesen. Diese ermöglicht eine bessere Übersicht über die Tests.

Um die Integritäts- und Konsistenz Tests durchführen zu können, müssen noch folgende Parameter definiert werden.

- $\omega = max(\mathbf{w})$
- $t = |\mathbf{n}|$
- $N_E = |\mathbf{v}|$
- $s = |\hat{\boldsymbol{D}}|$
- $n = \sum_{j=1}^{t} n_j$
- $N = |\mathbf{e}_1|$
- $k = \sum_{j=1}^{t} k_j$

1. Vollständigkeit

In diesem Teil wird überprüft, ob alle erforderlichen Daten vorhanden sind. Denn nur so kann man eine lückenlose Verifizierung gewährleisten. Es werden hier also einfach nochmal alle Daten aufgeführt. Jeder Test überprüft ob ein Parameter vorhanden ist oder nicht. Der Output ist also entweder wahr oder falsch.

1.1 Vor der Wahl

1.1.1 Check for election identifier U

1.1.2 Check for vector of number of candidates in each election $\mathbf{n}$

1.1.3 Check for vector of candidate description $\mathbf{c}$

1.1.4 Check for vector of number of selection in each election $\mathbf{k}$

1.1.5 Check for vector of voter description $\mathbf{v}$

1.1.6 Check for vector of assigned counting circles $\mathbf{w}$

1.1.7 Check for eligibility matrix $\mathbf{E}$

1.1.8 Check for vector of public voter credentials $\hat{\mathbf{D}}$

1.1.9 Check for vector of public keys of Authorities $\mathbf{pk}$

1.1.10 Check for signature of full election parameters σ_1^{param}

1.1.11 Check for signature of part of election parameters σ_2^{param}

1.1.12 Check for signature of other part of election parameters σ_3^{param}

1.1.13 Check for vector of signature of public credentials $\mathbf{s}_{prep}$

1.1.14 Check for vector of signature of public keys $\mathbf{s}_{kgen}$

1.2 Während der Wahl

1.2.1 Check for ballot list $\langle(v, \alpha)\rangle$

1.2.2 Ballot Tests $\mathbf{A}$

1.2.2.1 For all elements check for voter ID ballot v

1.2.2.2 For all elements check ballot α

1.2.3 Check for OT-response list $\langle(v, \beta_j, \sigma_{ij}^{cast})\rangle$

1.2.4 Responses Tests $\mathbf{B}$

1.2.4.1 For all elements check for voter ID v

1.2.4.2 For all elements check for OT-response β_j

1.2.4.3 For all elements check for signature σ_{ij}^{cast}

1.2.5 Check for confirmation list $\langle(v, \gamma)\rangle$

1.2.6 Confirmation Tests $\mathbf{C}$

1.2.6.1 For all elements check for voter ID v

1.2.6.2 For all elements check for confirmation γ

1.2.7 Check for finalization list $\langle(v, \delta_j, \sigma_{ij}^{cast})\rangle$

1.2.8 Finalization Tests $\mathbf{D}$

1.2.8.1 For all elements check for voter ID v

1.2.8.2 For all elements check for finalization δ_j

1.2.8.3 For all elements check for signature σ_{ij}^{cast}

1.3 Nach der Wahl

1.3.1 Check for mixed and re-encrypted Ballot lists $\mathbf{E}'$

1.3.2 Check for vector of Shuffle Proofs $\boldsymbol{\pi}$

1.3.3 Check for vector of partial decrypted Ballot lists $\mathbf{B}'$

1.3.4 Check for vector of decryption Proofs $\boldsymbol{\pi}'$

1.3.5 Check for election result matrix $\mathbf{V}$

1.3.6 Check for counting circle matrix $\mathbf{E}$

1.3.7 Check for signature of tallying result σ^{tally}

1.3.8 Check for vector of signatures of mixed re-encryptions $\mathbf{s}_{mix}$

1.3.9 Check for vector of signatures of partial decryption's $\mathbf{s}_{dec}$

2.1 Integrität

In diesem Teil wird die Integrität der Parameter geprüft. Es wird also geprüft ob die Parameter in sich schlüssig sind. Beispielweise wird geprüft, ob sie sich im geforderten Wertebereich befinden. Der Output ist wieder entweder wahr oder falsch.

2.1 Vor der Wahl

2.1.1 Check if $U \in A^*_{\mathbf{ucs}}$

2.1.2 Check if $\omega \geqslant 1$

2.1.3 Check if $t \geqslant 1$

2.1.4 Check if $N_E \geqslant 0$

2.1.5 Check if $s \geqslant 1$

2.1.6 For all $j \in \{1, ..., t\}$, check if $n_j \geqslant 2$

2.1.7 For all $j \in \{1, ..., t\}$, check if $k_j \geqslant 1$

2.1.8 EligibilityMatrix Tests

2.1.8.1 For all $j \in \{1, ..., t\}$, $i \in \{1, ..., N_E\}$, check if $e_{ij} \in \mathbb{B}$

2.1.8.2 Check if $\sum_{j=1}^{t} e_{ij} \geqslant 1$

2.1.9 For all $i \in \{1, ..., n\}$, check if $C_i \in A^*_{\mathbf{ucs}}$

2.1.10 For all $i \in \{1, ..., N_E\}$, check if $V_i \in A^*_{\mathbf{ucs}}$

2.1.11 For all $i \in \{1, ..., N_E\}$, check if $\omega_i \in \{1, ..., \omega\}$

2.1.12 For all $j \in \{1, ..., s\}$, $i \in \{1, ..., N_E\}$, check if $\hat{d}_{ij} = (\hat{x}_{ij}, \hat{y}_{ij})$

2.1.13 For all $j \in \{1, ..., s\}$, check if $pk_j \in \mathbb{G}_q$

2.1.14 Check if $\sigma_1^{param} \in \mathbb{B} \times \mathbb{Z}_q$

2.1.15 Check if $\sigma_2^{param} \in \mathbb{B} \times \mathbb{Z}_q$

2.1.16 Check if $\sigma_3^{param} \in \mathbb{B} \times \mathbb{Z}_q$

2.1.17 For all $j \in \{1, ..., s\}$, check if $\sigma_j^{prep} \in \mathbb{B} \times \mathbb{Z}_q$

2.1.18 For all $j \in \{1, ..., s\}$, check if $\sigma_j^{kgen} \in \mathbb{B} \times \mathbb{Z}_q$

2.2 Während der Wahl

2.2.1 Ballot Tests $\mathbf{A}$

2.2.1.1 For all elements check if $v \in \{0, ..., N_E\}$

2.2.1.2 For all elements in α, check if $\hat{x}_v \in \mathbb{G}_{\hat{q}}$

2.2.1.3 For all elements in α, check if $a_j = (a_{j,1}, a_{j,2}) \in \mathbb{G}_q^2$

2.2.1.4 For all elements in α, check if $\pi_\alpha \in (\mathbb{G}_{\hat{q}} \times \mathbb{G}_q^2) \times (\mathbb{Z}_{\hat{q}} \times \mathbb{G}_q \times \mathbb{Z}_q)$

2.2.2 Response Tests $\mathbf{B}$

2.2.2.1 For all elements check if $v \in \{0, ..., N_E\}$

2.2.2.2 For all elements check if $\beta_j \in \mathbb{G}_q^{k'_v} \times (\mathcal{B}^{L_M})^{nk'_v} \times \mathbb{G}_q$

2.2.2.3 For all elements check if $\sigma_{vj}^{cast} \in \mathbb{B}^\ell \times \mathbb{Z}_q$

2.2.3 Confirmation Tests $\mathbf{C}$

2.2.3.1 For all elements if $v \in \{0, ..., N_E\}$

2.2.3.2 For all elements in γ, check if $\hat{y}_v \in \mathbb{G}_{\hat{q}}$

2.2.3.3 For all elements in γ, check if $\pi_\beta \in \mathbb{G}_{\hat{q}} \times \mathbb{Z}_{\hat{q}}$

2.2.4 Finalization Tests $\mathbf{D}$

2.2.4.1 For all elements check if $v \in \{0, ..., N_E\}$

2.2.4.2 For all elements check if $\delta_j \in \mathcal{B}^{L_F} \times \mathbb{Z}_q^2$

2.2.4.3 For all elements check if $\sigma_{vj}^{\text{conf}} \in \mathbb{B}^\ell \times \mathbb{Z}_q$

2.3 Nach der Wahl

2.3.1 For all $j \in \{1, ..., s\}$ check if $\mathbf{e}_j \in (\mathbb{G}_q^2)^{N*}$

2.3.2 For all $j \in \{1, ..., s\}$ check i$^{\pi_j \in (\mathbb{G}_q^3 \times \mathbb{G}_q^2 \times \mathbb{G}_q^N) \times (\mathbb{Z}_q^4 \times \mathbb{Z}_q^N \times \mathbb{Z}_q^N) \times \mathbb{G}_q^N \times \mathbb{G}_q^N}$

2.3.3 For all $j \in \{1, ..., s\}$ check if $\mathbf{b}'_j \in \mathbb{G}_q^N$

2.3.4 For all $j \in \{1, ..., s\}$ check if $\pi'_j \in (\mathbb{G}_q \times \mathbb{G}_q^N) \times \mathbb{Z}_q$

2.3.5 For all $j \in \{1, ..., s\}$, $i \in \{1, ..., N\}$ check if $v_{ij} \in \mathbb{B}$

2.3.6 For all $j \in \{1, ..., s\}$, $i \in \{1, ..., N\}$ check if $\omega_{ij} \in \mathbb{B}$

2.3.7 For all $j \in \{1, ..., s\}$ check if $\sigma_j^{mix} \in \mathbb{B} \times \mathbb{Z}_q$

2.3.8 For all $j \in \{1, ..., s\}$ check if $\sigma_j^{dec} \in \mathbb{B} \times \mathbb{Z}_q$

2.3.9 Check if $\sigma^{tally} \in \mathbb{B} \times \mathbb{Z}_q$

3. Konsistenz

In diesem Teil wird geprüft ob die Parameter zu den anderen konsistent sind. Wir haben zu Beispiel zwei Vektoren, welche die gleiche Länge haben sollten. Nun wird geprüft ob diese wirklich die gleiche Länge haben. Auch hier ist der Output wahr oder falsch.

3.1 Vor der Wahl

3.1.1 Check if $|\mathbf{n}| = t$

3.1.2 Check if $|\mathbf{k}| = t$

3.1.3 Check if $|\mathbf{E}| = N_E$

3.1.4 For all $i \in \{1, ..., N_E\}$ check if $|\mathbf{e}_i| = t$

3.1.5 Check if $|\mathbf{c}| = n$

3.1.6 Check if $|\mathbf{v}| = N_E$

3.1.7 Check if $|\mathbf{w}| = N_E$

3.1.8 Check if $|\hat{\mathbf{D}}| = s$

3.1.9 For all $j \in \{1, ..., s\}$, check $|\hat{\mathbf{d}}_j| = N_E$

3.1.10 Check if $|\mathbf{pk}| = s$

3.1.11 Check if$^{p_{n+w}\prod_{j=1}^{k} p_{n-j+1}} < p$

3.2 Während der Wahl

3.2.1 For all elements in $\mathbf{A}$, for all α, check if $|\mathbf{a}| = k$

3.2.2 Response Tests $\mathbf{B}$

3.2.2.1 For all elements, for $\boldsymbol{\beta}_v = \{\beta_{v,1}, ..., \beta_{v,s}\}$, check if $|\boldsymbol{\beta_v}| = s$

3.2.2.2 For all elements, for $\mathbf{S}_{cast_v} = \{\sigma_{v,1}^{cast}, ...\sigma_{v,s}^{cast}\}$,

check if $|\mathbf{S}_{cast_v}| = s$

3.2.3 Finalization Tests $\mathbf{D}$

3.2.3.1 For all elements, for $\boldsymbol{\delta}_v = \{\delta_{v,1}, ..., \delta_{v,s}\}$, check if $|\boldsymbol{\delta_v}| = s$

3.2.3.2 For all elements, for $\mathbf{S}_{\text{conf}_v} = \{\sigma_{v,1}^{\text{conf}}, ...\sigma_{v,s}^{\text{conf}}\}$,

check if $|\mathbf{S}_{\text{conf}_v}| = s$

3.3 Nach der Wahl

3.3.1 Check if $|\mathbf{E}'| = s$

3.3.2 For all $j \in \{1, ..., s\}$, check if $|\mathbf{e}'_j| = N$

3.3.3 Check if $|\boldsymbol{\pi}| = s$

3.3.4 Check if $|\mathbf{B}'| = s$

3.3.5 For all $j \in \{1, ..., s\}$, check if $|\mathbf{b}'_j| = N$

3.3.6 Check if $|\boldsymbol{\pi}'| = s$

3.3.7 Check if $|\mathbf{V}| = N$

3.3.8 For all $i \in \{1, ..., N\}$, check if $|\mathbf{v}_i| = n$

3.3.9 Check if $|\mathbf{W}| = (N, \omega)$

3.3.10 For all $i \in \{1, ..., N\}$, check if $|\omega_i| = \omega$

3.3.11 Check if $|\mathbf{s}_{mix}| = s$

3.3.12 Check if $|\mathbf{s}_{dec}| = s$

4. Evidenz

In diesem Teil wird geprüft, ob die verschieden kryptographischen Beweise stimmen. Dazu wird eine Verifizierungsfunktion aufgerufen, die wahr oder falsch zurückgibt.
Es handelt sich hierbei um sogenannte Nicht-Interaktive Zero-Knowledge-Beweise. Sie dienen dazu, zu beweisen, dass man Kenntnis von gewissen Parameter oder Schlüsseln hat.Beispiel „proof of confirmation“ : Hier wir der Beweis erbracht, dass man sowohl das öffentliche „confirmation credential“ und auch das private „vote validity credential“ kennt.

4.1 Proof Tests

4.1.1 For all elements in α, check proof of validity of ballot π_α

4.1.2 For all elements in $\mathbf{C}$, check proof of confirmation π_β

4.1.3 For all $j \in \{1, ..., s\}$, check shuffle proof π_j

4.1.4 For all $j \in \{1, ..., s\}$, check decryption proof π'_j

5. Authentizität

In diesem Teil wird die Gültigkeit der Zertifikate und Signaturen überprüft.
Auch hier wird eine Verifizierungsfunktion aufgerufen, welche wiederum wahr oder falsch zurückgibt. Jedes Zertifikat besitzt einen Gültigkeitszeitraum sowie einen öffentlichen Schlüssel. Mit dem öffentlichen Schlüssel kann dann die Signatur des Zertifikates überprüft werden.

5.1 Zertifikate

5.1.1 Check validity of Certificate of election administrator C_{Admin}

5.1.2 For all $j \in \{1, ..., s\}$, check validity of the Certificates of authorities C_{Auth_j}

5.2 Signaturen

5.2.1 Check signature of full election parameters σ_1^{param}

5.2.2 Check signature of part of election parameters σ_2^{param}

5.2.3 Check signature of other part of election parameters σ_3^{param}

5.2.4 Check signature of tallying result σ^{tally}

5.2.5 For all $j \in \{1, ..., s\}$, check signature of public credentials σ_j^{prep}

5.2.6 For all $j \in \{1, ..., s\}$, check signature of public keys σ_j^{kgen}

5.2.7 For all elements in $\mathbf{B}$, for $j \in \{1, ..., s\}$, $i \in \{1, ..., N\}$ check σ_{ij}^{cast}

5.2.8 For all elements in $\mathbf{D}$, for $j \in \{1, ..., s\}$,$i \in \{1, ..., N\}$ check $\sigma_{ij}^{\text{conf}}$

5.2.9 For all $j \in \{1, ..., s\}$, signatures of mixed re-encryption's check σ_j^{mix}

5.2.10 For all $j \in \{1, ..., s\}$, check signatures of partial decryption's σ_j^{dec}

Selbständigkeitserklärung

Ich bestätige, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der im Literaturverzeichnis angegebenen Quellen und Hilfsmittel angefertigt habe. Sämtliche Textstellen, die nicht von mir stammen, sind als Zitate gekennzeichnet und mit dem genauen Hinweis auf ihre Herkunft versehen.

Ort, Datum:

Unterschrift: