Cloud-Assisted Access Control for the Internet of Things

Viktor Bubanja
University of Canterbury
Christchurch, New Zealand
vbu19@uclive.ac.nz

Clémentine Gritti
University of Canterbury
Christchurch, New Zealand
clementine.gritti@canterbury.ac.nz

ABSTRACT

The Internet of Things (IoT) involves providing everyday devices with a means to identify themselves and communicate with each other over the Internet. It is a rapidly growing technology and has already brought significant benefits to a range of domains, such as smart cities, smart factories, wearables, e-health, and self-driving cars. However, the increase of data transfer and inter-connection between devices also brings an added security risk. As more personal devices are connected to the internet, privacy and security concerns must be considered. Furthermore, the potential for large numbers of unsecure IoT devices being coordinated to achieve detrimental cyberattacks is a real concern and there have been a growing number of these attacks in recent years. As the number of connected IoT devices grows each year, the frequency and severity of these attacks grow. It is clear from the number of IoT related cyberattacks that primitive authentication techniques are not suitable for the IoT. It is therefore vital to find new ways to secure the IoT, and in particular, to achieve secure authentication to prevent malicious unauthenticated IoT devices causing damage within an IoT network.

CHARIOT, an authentication protocol for the IoT, proposes a potential solution.

Within CHARIOT, IoT devices are authenticated based on attributes they possess which are kept secret by utilising digital signatures. Since digital signature generation is computationally intensive, the computations are offloaded to a powerful cloud server which performs the computations without gaining any knowledge about the private attributes.

This paper describes a prototype implementation for CHARIOT. The prototype allowed for the protocol to be successfully validated by performing time benchmarks that showed that the protocol is suitable for a real IoT environment. Furthermore, due to the successful implementation of the prototype, it can be extended to form further prototypes, and finally, a fully operational authentication system for a real IoT environment.

ACM Reference Format:
Viktor Bubanja and Clémentine Gritti. 2020. Cloud-Assisted Access Control for the Internet of Things. In *Proceedings of SENG402 - Software Engineering Research Project (SENG402).* ACM, New York, NY, USA, 13 pages.

SENG402, 2019, University of Canterbury, New Zealand

1 INTRODUCTION

The Internet of Things (IoT) is an emerging technology that is allowing communication between an increasing number of devices over the Internet. This communication is enabling information gathering and analysis on a level previously impossible. The number of connected IoT devices is rapidly increasing and has surpassed 50 billion in 2020 [1]. The benefits of IoT are widespread across a range of industries such as agriculture, manufacturing, retail, and healthcare.

However, with this increase in inter-connectivity and data sharing, there is also added security risk [2]. With IoT, data can be shared between devices that would previously never have been able to communicate, potentially exposing private information. Furthermore, the massive amounts of weak and unsecure devices now connected to the internet bring the potential for large-scale cyberattacks that coordinate many infected IoT devices to achieve detrimental results. Authentication within IoT has added challenges due to the restricted computational power and storage capacity of IoT devices, such as actuators, sensors, and wearables. CHARIOT [3] is an innovative authentication protocol that proposes a solution to these problems. By utilising a powerful cloud server, computationally expensive access control operations are offloaded from the less powerful IoT devices.

To use CHARIOT in a real IoT environment, a functional prototype must first be built which can be tested and time benchmarked to determine the protocol's scalability. Furthermore, the system parameters must be experimentally evaluated in order to determine suitable values.

In Section 2, the objectives of the project and the context to the problem are provided. In Section 3, existing work related to CHARIOT is explored. Section 4 describes the design of the solution. Implementation details and the software development process are described in Section 5. Section 6 contains an evaluation of the implementation and a discussion of the results obtained. Finally, Section 7 contains conclusions from the project and Section 8 contains potential future work that may follow. Additional material is provided in Section 9 including images related to how the project was organised, how the source code was stored and versioned, as well as some examples of unit tests.

2 BACKGROUND AND OBJECTIVES

The sole supervisor of the project was Clémentine Gritti: one of the authors of CHARIOT and a University of Canterbury staff member. The project was completed individually with no other team members or third-parties. This was the first iteration of the project so there was no existing solution to build upon.

The aim of the project was to build a prototype implementation of CHARIOT. CHARIOT is currently a theoretical authentication

protocol; to use it in a real IoT setting, a prototype must first be built which can be tested and evaluated. Time benchmarking is a key form of evaluation which must be performed to determine the scalability and suitability of CHARIOT within a realistic IoT environment. Furthermore, time benchmarking is needed to find suitable value ranges for the system parameters which define what environments CHARIOT may be suitable for. Once successfully validated and evaluated, the prototype may be further extended in the future to form further prototype iterations, and finally, a fully functional and commercial authentication system for the IoT.

3 RELATED WORK

Recent surveys of security and privacy of IoT [4], [5] have identified the most important directions for future research to be authentication and privacy, computational complexity, communication overhead, and verification techniques.

In addition to traditional access control systems, such as discretionary access control (DAC) and mandatory access control (MAC), attribute-based access control has recently attracted significant interest [6]. Unlike traditional systems, attribute-based access control is considered a "next-generation" model since it provides dynamic, context-aware access control to resources based on various attributes. Traditional access control solutions are not suitable for IoT devices with limited computing power requiring access to IoT platforms.

Gritti *et. al.* [3] have proposed an attribute-based scheme for controlled access of IoT devices to an IoT platform. Their work includes a comprehensive description of novel techniques that enable authentication of IoT devices to an IoT platform without compromising the privacy of the devices. Below, we discuss the existing work related to CHARIOT and mention the shortcomings that CHARIOT overcomes.

Attribute-based signatures (ABS), introduced in [7], allow a party to sign a message with a predicate that is satisfied by attributes from an authority. The signature reveals only the fact that a single user, who has some set of attributes satisfying the predicate, has signed the message. The signature does not disclose the attributes that were used to satisfy the predicate or any identifying information about the signer. ABS are useful in a variety of applications, such as anonymous authentication and attribute-based messaging.

There have been a number of works that built upon this idea, including [8], [9]. These papers contain constructions that provide a high-level of security and are proved secure against the random-oracle assumption and the standard computational Diffie-Hellman assumption (these are assumptions that can be used as the basis of security proofs). However, the high level of security also results in a lack of efficiency which is vital for practical applications. Previous ABS schemes contained signatures that grew linearly in size with the number of attributes specified by the attribute authority, resulting in poor performance. Herranz *et al.* [10] proposed an ABS scheme with constant-size signatures and a threshold policy resulting in an increase in efficiency. Threshold policies allow the attribute authority to define the minimum number of specified attributes a user must hold to be authenticated. The solution proposed by Herranz *et al.* [10] utilises dummy attributes however (attributes whose only purpose is to pad the specified attributes to maintain a constant number of total attributes), resulting in computational and storage overhead. Susilo *et al.* [11] improve upon the work of Herranz *et al.* [10] by proposing a new scheme that does not rely on dummy attributes and has increased performance.

Securely outsourcing expensive computations is explored in [12], [13], [14] which involve outsourcing sequence comparison, linear algebra, and linear programming computations respectively. However, these cannot be directly applied to signature generation. Schemes which utilise servers to specifically aid signature generation are proposed in [15], [16]; however, they do not allow access control management based on user's credentials.

4 DESIGN

4.1 Overview of CHARIOT

The CHARIOT protocol consists of four entities that run six algorithms between them.

4.1.1 *Entities.*

(1) The IoT device to be authenticated
(2) The IoT platform responsible for authenticating devices
(3) The untrusted powerful cloud server which performs computationally intensive mathematical operations on behalf of the IoT device
(4) The trusted attribute authority which is responsible for generating secret keys and initialising the protocol

4.1.2 *Algorithms.*

(1) *Setup* (run by the trusted attribute authority): Initialises the public parameters within the protocol and the master secret key stored by the trusted attribute authority
(2) *KeyGen* (run by the trusted attribute authority): Generates secret keys for the cloud server, IoT device, and IoT platform
(3) *Request* (run by the IoT device): Hashes the attributes embedded in the access policy
(4) $Sign_{out}$ (run by the cloud server): Performs computationally intensive digital signature generation for the IoT device
(5) *Sign* (run by the IoT device): Finalises the digital signature of the IoT device
(6) *Verify* (run by the IoT platform): Decides whether or not the IoT device should be authenticated based on its signature

A diagram of the CHARIOT protocol is shown in Figure 1.

The various design decisions required to implement the protocol are discussed below including preliminary information.

4.2 Preliminaries

Several algebraic structures are needed for the implementation of CHARIOT. Here we provide their brief definitions; more details are given in the abstract algebra textbooks [17], [18].

4.2.1 *Group.*

A group is a set G combined with a binary operation $\oplus$ that satisfies the following properties:

(1) The group is closed under the operation: that is, the operation assigns to each ordered pair (a, b) of elements in G an element in G denoted $a \oplus b$.
(2) The group contains an identity element 0, such that for all elements a in G, $a \oplus 0 = 0 \oplus a = a$.

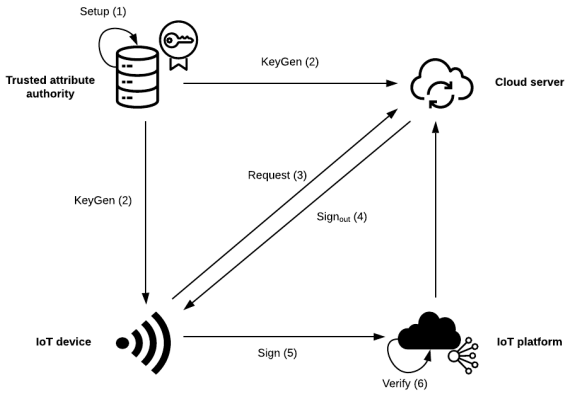

Figure 1: Overview of CHARIOT architecture

(3) The group contains inverses: for each element a in G, there is an element b in G, such that $a \oplus b = b \oplus a = 0$.
(4) The operation is associative: that is, for all elements a, b, c in G it is valid that $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.

If a group has the property that $a \oplus b = b \oplus a$ for every pair of elements a, b in G, the group is called *abelian*.

4.2.2 *Field.*

A field is a set F together with two binary operations $\oplus$ (addition) and $\odot$ (multiplication), that satisfy the following requirements:

(1) Identities: there are two elements 0 and 1 in F such that for all elements a in F, $a \oplus 0 = a$ and $a \odot 1 = a$.
(2) Additive inverse: for every element a in F, there exists b in F such that $a \oplus b = 0$.
(3) Multiplicative inverse: for every element a in F, there exists b in F such that $a \odot b = 1$.
(4) Both operations are associative, that is for all elements a, b, c in F it is valid that $(a \oplus b) \oplus c = a \oplus (b \oplus c)$, and $(a \odot b) \odot c = a \odot (b \odot c)$.
(5) Commutativity of addition and multiplication: $a \oplus b = b \oplus a$ and $a \odot b = b \odot a$ for every pair of elements a, b in F.
(6) Distributivity of multiplication over addition, that is for all elements a, b, c in F, $a \odot (b \oplus c) = a \odot b \oplus a \odot c$.

4.2.3 *Bilinear map.*

A bilinear map is a function that takes an element from a group G_1 and an element from a group G_2 and outputs an element from another group G_T. If the map is symmetric, then G_1 and G_2 are the same group.

CHARIOT relies on an efficiently computable (symmetric) bilinear map, $e : G_1 \times G_2 \rightarrow G_T$. The specific groups and bilinear map are not specified by CHARIOT and can vary between implementations. A decision was made to use elliptic curves for the groups and elliptic curve pairings for the bilinear map. The map takes two points on an elliptic curve and outputs a point on the same curve. Below is a description of elliptic curve cryptography and the motivation behind this decision.

4.3 Elliptic Curve Cryptography

The goal of cryptography is to conceal information from an adversary that might have access to the communication channel between two parties. Before the mid-1970s, all cipher systems used symmetric key algorithms, that is the same secret key was used by the sender and the receiver. The key in every such system had to be exchanged securely between the communicating parties before the system could be used. This requirement quickly becomes overwhelming with an increase in the number of communicating parties or when keys are frequently changed. Diffie-Hellman key exchange was a groundbreaking protocol introduced to enable the secure exchange of keys over a public channel [19]. Their method was followed shortly afterwards by RSA [20], which is nowadays widely used for data encryption of e-mail and other internet transactions. All public cryptographic systems employ trapdoor functions; such functions are easy to compute one way but are difficult to invert. In the case of RSA, the trapdoor function involves calculating the multiplication of two prime numbers; computing the inverse operation, that is finding the prime factors of a composite number, is a highly time-consuming operation. The most efficient algorithm for integer factorization is the general number field sieve for which the running time is sub-exponential [21] (caveat: this applies to classical computers; for a quantum computer there is much more efficient algorithm [22], but large scale quantum computers capable of this task have not yet been built). As the computing power available to adversaries increases, the size of the keys needs to grow. This is not suitable for IoT devices with limited storage capacity and computational power; therefore, RSA is not an ideal system for the IoT environment. Elliptic curves provide much more efficient trapdoor functions and are used widely throughout various internet protocols, such as encryption over SSL/TLS/HTTPS. The same level of security of the RSA algorithm with a 2048-bit prime is achieved with elliptic curve cryptography (ECC) with a 224-bit prime [23]. Below we provide a brief description of the basic elements of ECC [24], [25].

An elliptic curve over real numbers is defined as a plane curve of the form:

$$y^2 = x^3 + ax + b, \tag{1}$$

where $a, b \in \mathbb{R}$. The curve is defined to be smooth when its graph has no cusps, self-intersections or isolated points. For the above elliptic curve, this is satisfied when its discriminant $\Delta = -16(4a^3 + 27b^2) \neq 0$. For $\Delta < 0$, the graph has one, otherwise two components. A property of the set of points on an elliptic curve is that together with a point O 'at infinity' they form an abelian group. The point 'at infinity' is defined as the additive identity, that is $P + O = P$. The addition of two points, P and Q, is defined by drawing a line through the points and reflecting the intersection between the line and the elliptic curve over the x axis. An example of addition is shown in Figure 2.

The algebraic expressions for the coordinates of $P + Q$, corresponding to the above geometric construction, when $P \neq Q$ are

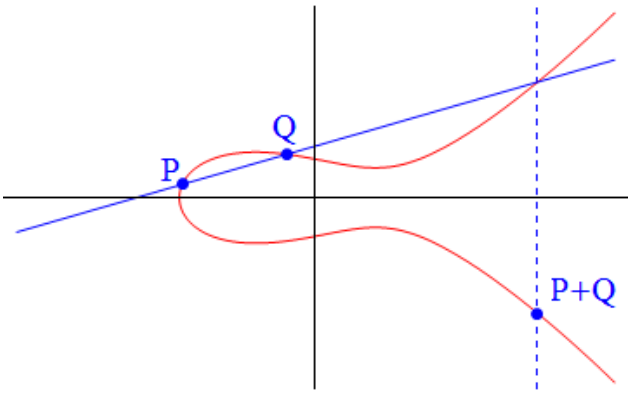

Figure 2: Elliptic curve addition on a curve with a=-3, b=5.

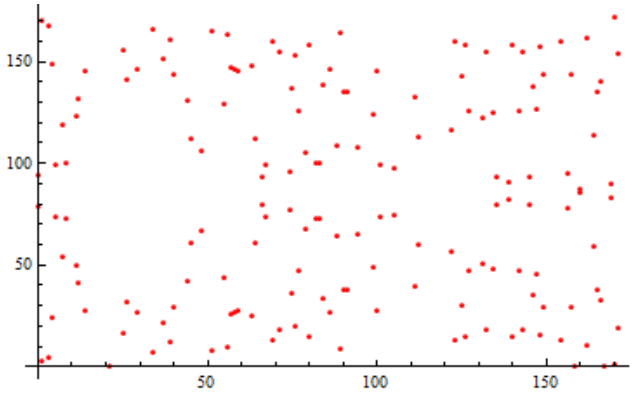

Figure 3: Elliptic curve modulo p with a=-5, b=13, p=173.

given by,

$$x_{P+Q} = \left(\frac{y_P - y_Q}{x_P - x_Q}\right)^2 - x_P - x_Q \tag{2}$$
$$y_{P+Q} = -y_P + \frac{y_P - y_Q}{x_P - x_Q}(x_P - x_{P+Q}).$$

When $P = Q$, we have,

$$x_{2P} = \left(\frac{3x_P^2 + a}{2y_P}\right)^2 - 2x_P \tag{3}$$
$$y_{2P} = -y_P + \frac{3x_P^2 + a}{2y_P}(x_P - x_{P+Q}).$$

The notation nP is used to denote P added to itself n times if n is positive, and otherwise - P added to itself $|n|$ times[1]. Computing nP, by using the double and add algorithm is $O(\log n)$. For a given P and nP, finding n is known as the logarithm problem. There is a variant of the logarithm problem, called the discrete logarithm problem, when considering a finite field. If we reduce the domain of an elliptic curve to be a finite field, scalar multiplication remains easy, while the discrete logarithm becomes a hard problem. This is called the elliptic curve discrete logarithm problem (ECDLP) and is the basis of security of ECC. Currently, no known algorithm solves this problem on a classical computer in polynomial time.

For this reason, instead of the real field, elliptic curves are defined over a finite field for cryptographic use. The most popular choice is a Galois field, $GF(p)$, where all arithmetic operations are performed modulo a prime p. In this case, the elliptic curve is defined as,

$$\{(x, y) \in GF(p)^2 | y^2 \equiv x^3 + ax + b (\mathrm{mod} p), \\ 4a^3 + 27b^2 \not\equiv 0 (\mathrm{mod} p)\} \cup \{\mathcal{O}\}, \tag{4}$$

where $p > 3$, $\mathcal{O}$ is still the point at infinity, and $a, b \in GF(p)$. An example of a curve over $GF(127)$ is shown in Figure 3. The definitions for calculations of point additions are the same as algebraically defined above for the case of a real field, except that all the operations are now performed modulo p. Division x/y in a finite field is defined as $x \cdot y^{-1}$, where y^{-1} is the multiplicative inverse of y, i.e. $y \cdot y^{-1} \equiv 1 (\mathrm{mod} p)$. For every point P, we have that $nP + mP = (n + m)P$, i.e. multiples of P are closed under addition.

[1]Alternative notation for nP is P^n which we use in other parts of the text

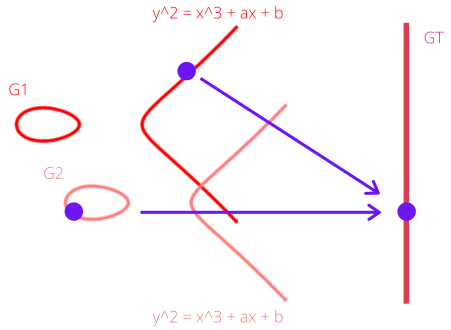

Figure 4: Elliptic curve pairing

Furthermore, the multiples of P form a cyclic subgroup of the group formed by the elliptic curve. The point P is called a base point or a generator of its cyclic subgroup.

In practice, instead of low order groups (groups with few elements), such as the curve shown in Figure 3, elliptic curve groups with high order are used (large prime p).

Elliptic curve pairing, shown in Figure 4, involves defining a function that takes two points from an elliptic curve (or from two different elliptic curves) and outputs an element from another group. This function is a map $e : G_1 \times G_2 \rightarrow G_T$, where G_1 and G_2 are elliptic curves, and G_T is another group. The map is bilinear if for any two element elements $g_1 \in G_1$ and $g_2 \in G_2$, and integers a, b:

$$e(g_1^a, g_2^b) = e(g_1, g_2)^{ab}, \tag{5}$$

The map e is called admissible bilinear map if $e(g_1, g_2)$ generates G_T and e is efficiently computable. Bilinear maps give cyclic groups additional properties that are useful for identity-based encryption [26, 27].

Elliptic curve pairings were therefore chosen to form the basis of all the mathematical operations within the implementation of CHARIOT. This meant that all operations were performed as elliptic curve arithmetic and the bilinear map took two points from an elliptic curve and output a point from the same curve.

4.4 Hash functions

One of the parameters of CHARIOT is a hash function, H, which must be collision-resistant (highly unlikely for two different inputs to hash to the same output). Unlike most hash functions, H must also be an extendable-output function (XOF), i.e. the output length (number of bits) is adjustable. Several hash functions were considered, including SHAKE128, SHAKE256 and BLAKE2. BLAKE2 performs faster than SHAKE128 and SHAKE256 [28] and all three functions have sufficient security needed for CHARIOT. Therefore, the BLAKE2 function was selected.

Another hash function within CHARIOT is τ, a hash-based message authentication code (HMAC). HMACs include a cryptographic hash function and a chosen secret key. The difficulty in selecting the hash function is that it must output values within the finite field of integers defined by p, a value defined by the elliptic curve we choose to use. The finite field of integers defined by p refers to integers modulo p, i.e. 0, 1, ..., $p-1$ ($\mathbb{Z}_p$). All calculations within the protocol are performed modulo p. If we simply set the output of the hash function to be the modulo of p of the output, the function cannot be guaranteed to be collision-resistant. We must therefore choose a hash function and an elliptic curve where the p value of the elliptic curve has more bits than the output of the hash function. Python has an in-built HMAC library that allows us to specify a hash function. SHA-256 was chosen as the hash function as it outputs 256-bit values and the elliptic curves provided by Charm have either a 512 or 1024-bit base field, i.e. p is at least 512-bit, ensuring collision-resistance.

4.5 Language and Framework

The code was written in Python with the aid of Charm[2], a framework for rapid cryptographic prototyping. There are a range of valuable cryptographic libraries, such as Stanford's Pairing-Based Cryptography Library[3] and MIRACL[4] which could be used to implement CHARIOT; however, many of them are not designed with usability in mind with the aim of enabling quick development of prototype systems. Charm performs intensive mathematical operations in native C modules to maximise performance while allowing prototypes to be written in Python to allow higher readability. Furthermore, Charm has an extensive range of example cryptographic schemes implemented by researchers that can be used for reference. Therefore, Charm was chosen as the cryptographic framework used for the implementation of the CHARIOT prototype.

5 IMPLEMENTATION

The algorithms within CHARIOT (Setup, KeyGen, Request, Sign_{out}, Sign, and Verify) were all implemented successfully. Since the arithmetic within CHARIOT occurs within multiple different groups with different operators, care had to be taken to ensure that the correct operations were being performed throughout the computations. An example calculation is given below.

[2]Charm, a framework for rapid cryptographic prototyping: http://charm-crypto.io
[3]The Pairing-Based Cryptography Library, a free C library for implementing pairing-based cryptosystems: Phttps://crypto.stanford.edu/pbc/
[4]MIRACL, an open source SDK for elliptic curve cryptography: https://github.com/miracl

```python
class Vector:
    def __init__(self, elements: list):
        self.elements = elements

    # Element-wise multiplication
    def dot(self, v2):
        assert len(self.elements) == len(v2.elements)
        return Vector([i * j for i, j in zip(self.elements, v2.elements)])

    # Element-wise exponentiation
    def exp(self, exponent):
        return Vector([v ** exponent for v in self.elements])

    def __getitem__(self, key):
        return self.elements[key]

    def __eq__(self, other):
        return self.elements == other.elements
```

Figure 5: Source code for the Vector class

$$e(T_2', v_{n-s+t}^{-1}) \cdot e(T_1, h^{\alpha F_S(\gamma)}) = e(u \cdot g^{\beta_1}, h_{s-t}). \tag{6}$$

where each of the symbols are values calculated or defined previously within CHARIOT.

Here, $h, g \in \mathbb{G}$, while $\alpha, \beta_1, \gamma \in_R \mathbb{Z}_p^*$. In the expression $u \cdot g^{\beta_1}$, all the operations are performed in the group $\mathbb{G}$, while in calculating the exponent $\alpha F_S(\gamma) = \alpha \prod_{at \in S}(\gamma + \tau(K, at)) = \alpha \sum_{i=0}^{s-t} \gamma^i b_i$, ($b_{s-t} = 1$) , all the operations are performed in the field Z_p. Operations performed within the $\mathbb{G}$ group are done with elliptic curve arithmetic whereas operations within Z_p are performed as addition and multiplication modulo p. Using Charm, the Python syntax is the same for operators in different groups and the functionality is determined by the type of the object that the operation is being applied to. To ensure that the correct objects were being used at every stage of the computations, unit tests were written that verify the correct type of the involved objects. Following test-driven development with this form of tests helped to identify bugs caused by incorrect object types within complex pieces of code. The tests that were written throughout development were mostly limited to simply verifying object types due to the limitation on the ability to test which is discussed in Section 5.3.

5.1 Object-Oriented Programming

Due to the complexity of the calculations, it was important to focus on code reusability and code cleanliness to minimise the potential for bugs. To achieve this, object-oriented programming was used. Helper classes were defined to store repeated functionality. For example, element-wise multiplication and exponentiation of vectors are used frequently throughout the protocol so a vector helper class was defined to store this logic. The source code for this class is shown in Figure 5.

A similar recurring calculation was the computation of Groth-Sahai commitments, which are defined as an element-wise multiplication of vectors:

$$\vec{C}_x = (1, 1, x)^T \cdot (\vec{g}_1)^{r_1} \cdot (\vec{g}_2)^{r_2},$$

where $x \in \mathbb{G}$, $r_1, r_2 \in \mathbb{Z}_p$, and $\vec{g}_i \in \mathbb{G}^3$.

Since the individual values within the above equation must be remembered, Groth-Sahai commitments were implemented as classes that contain both the values within the equation and the logic for performing the calculation. The code for the commitment class is shown in Figure 6.

```python
class Commitment:
    def __init__(self, r_theta: int, s_theta: int, theta: int, g1: Vector, g2: Vector):
        self.r_theta = r_theta
        self.s_theta = s_theta
        self.g1 = g1
        self.g2 = g2
        self.theta = theta
        self.value = self.calculate()

    def calculate(self) -> Vector:
        return Vector([self.g1[0] ** self.r_theta,
                       self.g2[1] ** self.s_theta,
                       self.theta * (self.g1[2] ** self.r_theta) * (self.g2[2] ** self.s_theta)])

    def __getitem__(self, key):
        return self.calculate()[key]
```

Figure 6: Source code for the Commitment class

```python
@dataclass
class PublicParams:
    attribute_universe: list
    n: int
    g: int
    h: int
    u: int
    vi: list
    hi: list
    g1: Vector
    g2: Vector
    g3: list
```

Figure 7: Source code for the PublicParams data class

CHARIOT contains many large tuples containing numerous values. To store this information, Python data classes were used. Data classes simplify the code for complex classes by providing built-in basic functionality for instantiating, printing, and comparing objects. For example, the public parameters within CHARIOT are given by: $params = (\lambda, \mathcal{P}, n, p, \mathbb{G}, \mathbb{G}_T, e, g, h, u, \{v_i\}_{i\in[0,n]}, \{h_i\}_{i\in[0,n]}, \vec{g}_1, \vec{g}_2, \{\vec{g}_{3,i}\}_{i\in[0,k]}, H, \tau)$. The corresponding Python class is shown in Figure 7.

Since there are a large number of such classes within the implementation, the amount of boilerplate code was significantly reduced.

5.2 Algorithm for converting a polynomial from factored form to expanded form

One of the necessary computations within CHARIOT is finding the coefficients of an expanded form polynomial given its factored form. This is a particularly expensive computation and was the cause of the key results of the evaluation of the prototype discussed in Section 7.

The following general formula was derived:

$$(x+a_1)(x+a_2)...(x+a_n) = x^n + (a_1+a_2+...+a_n)x^{n-1} + (a_1a_2+a_1a_3+...+a_{n-1}a_n)x^{n-2} + ... + (a_1a_2...a_i+...+a_{n-i+1}a_{n-i+2}...a_n)x^{n-i} + ... + a_1a_2...a_n \quad (7)$$

i.e. the coefficient of x^{n-i} in the expanded form is the sum of all i-length combinations of the values $a_1, a_2, ...a_i$ from the factored form. The source code that was written to find the coefficients of an expanded polynomial is shown in Figure 8.

5.3 Testing

Testing the implementation had unique challenges. Since each of the algorithms performed a series of complex computations which involved random numbers, it was impossible to verify the exact output of each algorithm. Furthermore, trivially testing the output of the individual algorithms would involve duplicating the mathematics within the unit tests and would therefore accomplish little.

```python
def get_polynomial_coefficients(numbers) -> list:
    coefficients = []
    for i in range(len(numbers), 0, -1):
        total = 0
        for combination in combinations(numbers, i):
            total += reduce(operator.mul, combination, 1)
        coefficients.append(total)
    return coefficients
```

Figure 8: Source code for generating the coefficients of a polynomial given its zeros

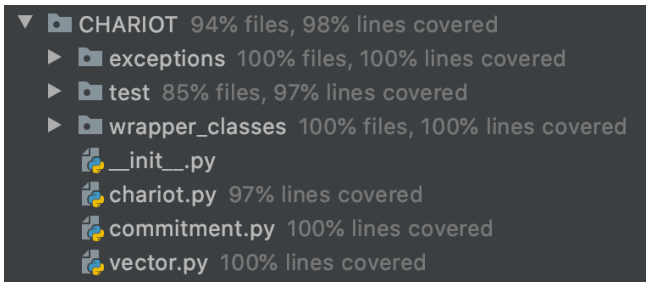

Figure 9: Unit test coverage

Therefore, until the entire protocol was implemented, testing was limited to verifying correct object types and testing of functions outside the core algorithms (for example, the function that finds polynomial coefficients). Once the entire protocol was implemented, it could be tested by passing in parameters that were expected to result in successful authentication or not and verifying the output. An example set of inputs is given below.

```
device_attributes = [1, 2, 3]

threshold = 2
policy_attributes = [1, 2, 3, 4]
```

These inputs specify that the IoT device must have at least 2 of the attributes specified by the policy to be authenticated which is the case here so the device should be authenticated. Unit tests with a range of inputs were used to validate the functionality of the implementation. A total of 36 automated unit tests were written. The tests achieved 97% line coverage as shown in Figure 9.

Since the program has no user interface and only one core module, forms of testing such as manual testing and integration testing were not appropriate and thus the prototype's functionality was tested solely with unit testing.

5.4 Project Management

5.4.1 *Research.*

CHARIOT is a theoretical cryptographic protocol that requires comprehensive validation before being able to be used in a practical environment. Validation research involves asking questions about something that does not exist yet: the implementation of the solution. Prototypes are therefore used to model a simplified version of the solution and can be evaluated to gain knowledge about the

validity of the final solution. Prototypes are often built iteratively with increasing complexity and evaluated in more and more realistic environments. Within this project, the first prototype iteration of CHARIOT was designed, implemented, and evaluated. The research method that was used was an instance of the design science cycle [29] (a regulative cycle). A regulative cycle involves designing a solution, validating the design, implementing the solution, then evaluating the solution. The solution design involved making the series of design decisions discussed within Section 4. These decisions were validated through discussions with the supervisor, Clémentine Gritti, where pros and cons of different solutions were explored and final design decisions were made. Implementing the solution is what consumed most of the time spent on the project and involved building the prototype using the design decisions. After implementation was complete, the prototype was evaluated with testing and time benchmarking. The effect of changing various system parameters within the implementation was measured to evaluate the protocol. Initially, all system parameters were experimented with, then more thorough experiments were conducted on the parameters that were found to have the greatest effect on the protocol's performance. The evaluation provided useful insight into the protocol's performance and scalability; this is discussed in detail in Section 6. Due to the prototypes successful implementation, it can be extended in further iterations of prototype implementations as validation research continues.

5.4.2 Software development.
Since the CHARIOT protocol is fixed and there are no changing user requirements or market conditions, the requirements of the project were strict, well understood, and highly unlikely to change. The timeline of the project was fixed by the deadlines throughout the year. The nature of the functionality of each of the six algorithms meant it was difficult to test their functionality until all of them were completed and thus the amount of testing that could be performed throughout development was limited. Since the project had fixed requirements, a firm timeline, and a limited ability to perform testing during development, each of the stages of development needed to be performed separately. Therefore, waterfall was used as the software development methodology [30].

To organise completion of tasks throughout the project, a kanban board was used. Tasks were organised by research, code, evaluation, and report related tasks. A prioritised backlog was created each week and the completed tasks were archived at the end of the week. Although a kanban board was used, the methodology of kanban, such as focusing on optimising the cycle time of tasks and removing bottlenecks, was not followed. The software tool used for the kanban board was Notion[5].

Git was used as the version control system and the codebase was stored in a repository on Github[6]. The Git Feature Branch Workflow [31] was used to work on individual features on separate branches within the repository. Since there was no team involved in the development, the main benefit of this strategy was the ability to experiment with different implementations without risking jeopardising the main prototype.

[5]Notion, a collaboration tool: https://www.notion.so
[6]GitHub, an online platform for hosting software projects with using Git: https://github.com/

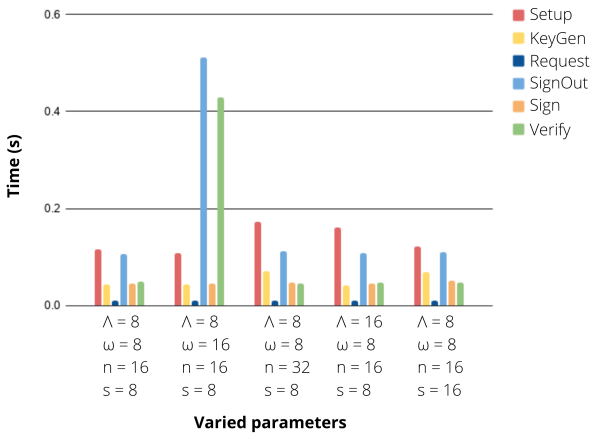

Figure 10: Time benchmarks of each algorithm within CHARIOT with varied parameters

6 EVALUATION AND DISCUSSION

6.1 Benchmarking

In order to evaluate the prototype implementation, a set of time benchmarks were performed to determine the suitability of the protocol in an IoT environment. The input parameters were varied during the benchmark process to determine how the prototype performed under various conditions. The parameters that were experimented with include:

- Security parameter, λ: This defines the length in bits of the hashed message. Since the message itself is public, the hash function does not need to be cryptographic but rather collision-resistant and thus λ must be sufficiently large to avoid collisions. It is assumed that the IoT devices have a limited dictionary of messages they can sign and thus relatively small values of λ can be chosen.
- Attribute set, Ω: This is the set of attributes contained by the IoT device. The size of the attribute set is referred to as ω.
- Policy, S: This is the set of attributes that defines which attributes the IoT device must have to be authenticated. The size of the policy is referred to as s.
- Upper bound of policy sizes, n: This defines the maximum number of attributes that policies can contain.

The execution time of each of the algorithms within the protocol was measured for varied parameter values to determine their individual performance under different conditions. The benchmarks results are shown in Figure 10. The initial parameter values were based on the example configuration within the original CHARIOT paper [3]. When varying the parameters, each value was doubled so that the performance impact could be compared between all parameters. All benchmarking was performed on a machine with a 2.7 GHz Dual-Core Intel Core i5 processor. More accurate benchmarking would require installing and running the program on separate machines that resemble the entities within the protocol (this is discussed in Section 8).

The time complexities of each of the algorithms are given below.

- Setup was found to scale linearly with the upper bound of policy sizes, n and the security parameter, λ.
- KeyGen was found to scale linearly with the upper bound of policy sizes, n, and the number of attributes on the IoT device, ω.
- Request was found to scale linearly with the number of attributes in the policy, s.
- Sign_{out} was found to scale exponentially with the number of attributes in the policy, s.
- Sign was found to scale linearly with the security parameter, λ.
- Verify was found to scale exponentially with the number of attributes in the policy, s.

6.1.1 *$Sign_{out}$ and Verify.*

The key finding was that $Sign_{Out}$ and Verify scale exponentially with the number of attributes in the policy.

The cause of the time complexity growth is the computation of polynomial coefficients given the list of its zeros (i.e. the conversion of a polynomial from factored form to expanded form). This requires finding all possible combinations of the values within the factored form of every length between 1 and the degree of the polynomial, each of the values within these combinations must then be multiplied together. This requires the following number of computations:

$$t(n) = \sum_{i=1}^{n} \binom{n}{i} i \tag{8}$$

To determine the time complexity of the algorithm, the equation for the number of computations can be simplified as follows:

$$\begin{aligned} t(n) &= \sum_{i=1}^{n} \binom{n}{i} i \\ &= \sum_{i=1}^{n} \frac{n!}{(n-i)!i!} i \\ &= \sum_{i=1}^{n} \frac{n!}{(n-i)!(i-1)!} \\ &= \frac{n!2^{n-1}}{\Gamma(n)} = \frac{n!2^{n-1}}{(n-1)!} = n2^{n-1} \end{aligned} \tag{9}$$

The time complexity is therefore:

$$O(2^n) \tag{10}$$

It is important to note that Sign_{out} is intended to run on a powerful cloud server and Verify on the IoT platform. Both of these machines are more powerful than the machine used for benchmarking and can be upscaled to mitigate the inefficiency of the algorithms. However, with an exponential time complexity, even upscaling has limited effects and larger policies quickly become impractical to process.

Further benchmarking was performed to determine the performance of Sign_{out} and Verify for various policy sizes to find the range of possible policy sizes where the algorithms complete within a reasonable time frame. The results are shown in Figure 11.

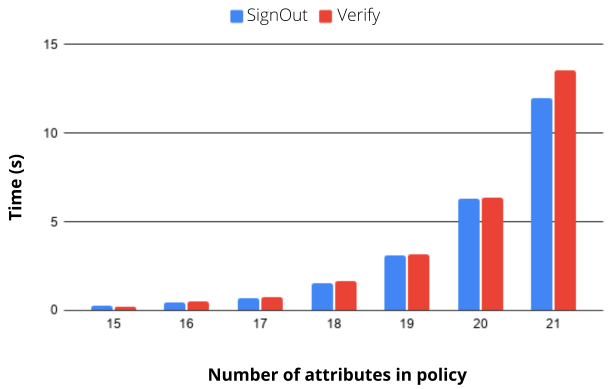

Figure 11: Time benchmarks of Sign_{out} and Verify algorithms for different policy sizes

It was observed that both algorithms completed in less than 1 second for up to 17 attributes. The time taken for the algorithms increased steeply for larger policy sizes and exceeded 10 seconds for policies with 21 attributes. This indicates the approximate maximum number of attributes the protocol can reasonably handle. The example configuration in the original CHARIOT paper [3] used a policy with 15 attributes. Furthermore, attribute policies of size 10 were chosen by Ambrosin et al. [32] in their experiment to determine the feasability of attribute-based encryption within a realistic IoT healthcare environment. A maximum size of 15 attributes is therefore reasonable and does not prevent CHARIOT from being used in an IoT environment with a large number of IoT devices. Furthermore, there are various potential strategies for overcoming this limitation within large systems. One such example is segregating devices into separated groups with unique group IDs. Specific group IDs can then be specified as attributes in policies to limit the number of devices affected by the policy and effectively decrease the size of the system.

6.1.2 *Request and Sign.*

Another key finding was that Request and Sign, the algorithms that run on the IoT device, were both efficient and scaled linearly with input parameters. This is vital since IoT devices are characteristically constrained in their computational power. The execution time of Request remained below 10 milliseconds for up to 500 attributes on the IoT device and was measured to execute in 0.0004 seconds for 15 attributes (a more realistic number of attributes). The execution time of Sign remained below 80 milliseconds for up to $\lambda = 512$ (maximum possible value) and was measured to execute in 0.0569 seconds for $\lambda = 40$ (a more realistic value). It is important to note that these benchmarks were performed on a machine that is significantly more powerful that a typical IoT device and do not form comprehensive benchmarks. However, they provide a valuable early indication of the performance, and especially the scalability (which does not depend on the machine used), of these algorithms.

6.2 Results Summary

Overall, given the efficiency of the algorithms that run on the computationally constrained IoT devices and the reasonable value range that the remaining algorithms remain performant for, the protocol was evaluated to be suitable for a realistic IoT environment. The limit on the number of attributes in the policies (approximately 15 depending on the power of the cloud server and IoT platform) was found to be sufficiently large to allow for fine-grained policies that can accurately authenticate specific subsets of devices from a large system.

6.3 Evaluation of Method

The inability to test the functionality of each algorithm individually was the cause of a significant amount of time spent debugging. By the time the entire protocol was implemented and could be tested, a significant amount of code was written which made locating bugs challenging.

The end-to-end tests that were written once the implementation was complete involved verifying the equalities within CHARIOT. An example of one of these equalities is:

$$e(T_1, H_S) = e(u, h_{s-t}) \cdot e(T_2, v_{n-s+t}) \tag{11}$$

where $T_1, H_S, u, h_{s-t}, T_2$, and v_{n-s+t} are calculated or defined earlier within the protocol. The tests initially did not pass (the equality did not hold) indicating there were bugs present.

To debug the implementation, the above equality was proved which was then used to write tests that verified each of the individual equalities within the proof. The proof is as follows:

$$\begin{aligned}
&e(T_2', v_{n-s+t}^{-1}) \cdot e(T_1, h^{\alpha F_S(h)}) = \\
&e\left(h^{(r-\beta_2)\gamma^n} \cdot \prod_{i=0}^{s-t-1} \left(h^{r\gamma^{i+n-s+t}}\right)^{b_i}, g^{-\frac{\alpha}{\gamma^{n-s+t}}}\right) \\
&\cdot e(g^{\frac{r}{F_{\Omega_S}(\gamma)}}, h^{\alpha F_S(\gamma)}) = \\
&e\left(h^{(r-\beta_2)\gamma^{s-t}} \cdot \prod_{i=0}^{s-t-1} \left(\left(h^{r\gamma^i}\right)^{\gamma^{n-s+t}}\right)^{b_i}, g^{-\frac{\alpha}{\gamma^{n-s+t}}}\right) \\
&\cdot e(g, h)^{\alpha r F_{S\setminus\Omega_S}(\gamma)} = \\
&e\left(h^{r\gamma^{s-t}} \cdot h^{-\beta_2\gamma^{s-t}} \cdot \prod_{i=0}^{s-t-1} \left(h^{r\gamma^i}\right)^{b_i}, g^{-\alpha}\right) \\
&\cdot e(g, h)^{\alpha r F_{S\setminus\Omega_S}(\gamma)} = \\
&e\left(h^{-\beta_2\gamma^{s-t}} \cdot \prod_{i=0}^{s-t} \left(h^{r\gamma^i}\right)^{b_i}, g^{-\alpha}\right) \cdot e(g, h)^{\alpha r F_{S\setminus\Omega_S}(\gamma)} = \\
&e(h, g)^{\alpha\beta_2\gamma^{s-t}} \cdot e(h, g)^{-\alpha r F_{S\setminus\Omega_S}(\gamma)} \cdot e(g, h)^{\alpha r F_{S\setminus\Omega_S}(\gamma)} = \\
&e(g^{\beta_2}, h^{\alpha\gamma^{s-t}}) = e(g^{\beta} \cdot g^{\beta_1}, h^{\alpha(s-t)}) = e(u \cdot g^{\beta_1}, h_{s-t}).
\end{aligned}$$

where the property of symmetric bilinear maps that states that $e(g, h) = e(h, g)$ was used. This method greatly helped in narrowing down the locations of bugs within the code.

However, to further decrease the time spent debugging, greater effort could have been made to enable testing at smaller intervals throughout development. To avoid simply duplicating mathematical operations in the unit tests, the best way to verify the correct functionality of the individual algorithms is to perform equality checks (such as the equality check in equation 11). Since all the equality checks within CHARIOT were already used for testing and debugging, this would have required deriving further equality checks using the properties of symmetric bilinear maps. Although this too would have been time consuming, it would have likely overall decreased the time spent on development.

7 CONCLUSION

A prototype implementation of CHARIOT was successfully built and evaluated by performing time benchmarking for a range of system parameters. The results indicated that CHARIOT successfully runs efficiently for IoT devices and is performant for realistic parameter values. However, benchmarking also revealed that the protocol is limited by the number of attributes that can be specified in the policies due to an algorithm with an exponential time complexity. The limit for the number of attributes was determined to be sufficiently large for this not to be a problem for the majority of systems, although it may be a problem for systems with massive amounts of IoT devices that require highly fine-grained access policies. It is also important to note that the algorithm runs on powerful machines, namely the cloud server and the IoT platform. These machines can be upscaled to mitigate the effects of the inefficient algorithm; however, with an exponential time complexity, there is limited potential for reducing the algorithm's run-time. Overall, given its performance for typical system parameter values, CHARIOT was determined to be suitable for a realistic IoT environment.

8 FUTURE WORK

Given the correct and robust functionality of the prototype, it can be further extended to form further prototype iterations. The successful implementation of the individual algorithms within the protocol will greatly speed up the process of building later prototypes. To improve the existing solution, it would be worthwhile to investigate more efficient methods for generating polynomial coefficients from polynomial zeros to replace the algorithm described in Section 5.2. This algorithm has exponential time complexity and is the main bottleneck for the implementation.

The next stage for the implementation is extending the system to run on web servers hosting the cloud server and the IoT platform and separate machines for the trusted attribute authority and IoT device where communication is performed over the Internet. The system then needs to be extended so that, instead of running procedurally, the web servers hosting the IoT platform and cloud server are listening for requests from the IoT device. The Charm framework will also eventually need to be replaced since it is designed for prototyping and therefore has sacrificed some performance for usability.

At each new iteration of the implementation, new benchmarks need to be performed that approximate the performance of the final product with increasing accuracy. Realistic devices need to be used for evaluation, such as specifically configured virtual machines or actual web servers and hardware devices for the IoT device (such as a Rasberry Pi).

Once the protocol has been sufficiently validated, the implementation can be extended further until the solution is a commercially viable product used in a real IoT environment.

REFERENCES

[1] Muhammad Burhan, Rana Asif Rehman, Byung-Seo Kim, and Bilal Khan. 2018. Iot elements, layered architectures and security issues: a comprehensive survey. *Sensors*, 18, (August 2018). DOI: 10.3390/s18092796.

[2] Syed Rizvi, Andrew Kurtz, Joseph Pfeffer, and Mohammad Rizvi. 2018. Securing the internet of things (iot): a security taxonomy for iot. In (August 2018), 163–168. DOI: 10.1109/TrustCom/BigDataSE.2018.00034.

[3] Clémentine Gritti, Melek Önen, and Refik Molva. 2018. Chariot: cloud-assisted access control for the internet of things. In (August 2018), 1–6. DOI: 10.1109/PST.2018.8514217.

[4] M. A. Ferrag, L. A. Maglaras, H Janicke, J. Jiang, and L. Shu. 2017. Authentication protocols for internet of things: a comprehensive survey. *Security and Communication Networks*, 1–41. DOI: 2017/6562953.

[5] T. Gebremichael, P. I. Ledwabe, M. H. Eldefrawi, G. P. Hancke, N. Pereira, M. Gidlund, and J. Akerberg. 2020. Security and privacy in the industrial internet of things: current standards and future challenges. *IEEE Access*, 8, 152351–152366. DOI: 10.1109/ACCESS.2020.3016937.

[6] Y. Andaloussi, M. D. El Ouadghiri, Y. Maurel, J. M. Bonnin, and H. Chaoui. 2018. Access control in iot environments: feasible scenarios. *Procedia Computer Science*, 130, 1031–1036.

[7] Hemanta Maji, Manoj Prabhakaran, and Mike Rosulek. 2010. Attribute-based signatures. In volume 2010. (January 2010), 1–35. DOI: 10.1007/978-3-642-19074-2_24.

[8] Jin Li, Man Ho Au, Willy Susilo, Dongqing Xie, and Kaili Ren. 2010. Attribute-based signature and its applications. In (January 2010), 60–69. DOI: 10.1145/1755688.1755697.

[9] Jin Li and Kwangjo Kim. 2010. Hidden attribute-based signatures without anonymity revocation. *Information Sciences*, 180, (May 2010), 1681–1689. DOI: 10.1016/j.ins.2010.01.008.

[10] Javier Herranz, Fabien Laguillaumie, Benoît Libert, and Carla Ràfols. 2012. Short attribute-based signatures for threshold predicates. In volume 7178. (February 2012), 51–67. DOI: 10.1007/978-3-642-27954-6_4.

[11] Willy Susilo, Guomin Yang, Fuchun Guo, and Qiong Huang. 2017. Constant-size ciphertexts in threshold attribute-based encryption without dummy attributes. *Information Sciences*, 429, (November 2017). DOI: 10.1016/j.ins.2017.11.037.

[12] Marina Blanton, Mikhail Atallah, Keith Frikken, and Qutaibah Malluhi. 2012. Secure and efficient outsourcing of sequence comparisons. In volume 7459. (September 2012), 277–287. DOI: 10.1007/978-3-642-33167-1_29.

[13] Mikhail Atallah and Keith Frikken. 2010. Securely outsourcing linear algebra computations. In (January 2010), 48–59. DOI: 10.1145/1755688.1755695.

[14] Cong Wang, Kaili Ren, and Jia Wang. 2011. Secure and practical outsourcing of linear programming in cloud computing. In (April 2011), 820–828. DOI: 10.1109/INFCOM.2011.5935305.

[15] Markus Jakobsson and Susanne Wetzel. 2001. Secure server-aided signature generation. In volume 1992. (February 2001), 383–401. DOI: 10.1007/3-540-44586-2_28.

[16] Bicakci K and N Baykal. 2004. Server assisted signatures revisited. In volume 2964, 143–156. DOI: 10.1007/978-3-540-24660-2_12.

[17] J. A. Gallian. 1994. *Contemporary abstract algebra*. D. C. Heath and Company.

[18] J. B. Fraleigh. 2014. *A first course in abstract algebra*. Pearson.

[19] Whitfield Diffie and Martin E Hellman. 1976. New directions in cryptography. *IEEE Transactions on Information Theory*, 22, (November 1976), 644–654. DOI: 10.1109/TIT.1976.1055638.

[20] R.L. Rivest, A. Shamir, and L. Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21, (February 1978), 120–126. DOI: 10.1.1.607.2677.

[21] Tom St Denis and Simon Johnson. 2007. *Cryptography for Developers*. Syngress Publishing, Inc.

[22] P.W. Shor. 1994. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 124–134. DOI: 10.1109/sfcs.1994.365700..

[23] S.A. Vanstone. 2003. Next generation security for wireless: elliptic curve cryptography. *Computers and Security*, 22, 412–415. DOI: 10.1016/S0167-4048(03)00507-8.

[24] Neal Koblitz. 1994. *A Course in Number Theory and Cryptography*. Springer-Verlag.

[25] Christof Paar and Jan Pelzl. 2010. *Understanding Cryptography*. Springer.

[26] J.H. Silverman. 1995. *The arithmetic of elliptic curves*. Springer-Verlah.

[27] B Lynn. 2007. *On the Efficient Implementation of Pairing-Based Cryptosystems*. PhD Thesis, Stanford University.

[28] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. 2013. Blake2: simpler, smaller, fast as md5. In (June 2013), 119–135. DOI: 10.1007/978-3-642-38980-1_8.

[29] Roel Wieringa. 2009. Design science as nested problem solving. In *Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology* (DESRIST '09). Association for Computing Machinery, Philadelphia, Pennsylvania. ISBN: 9781605584089. DOI: 10.1145/1555619.1555630. https://doi-org.ezproxy.canterbury.ac.nz/10.1145/1555619.1555630.

[30] Mohammad Mushfequr Rahman. 2019. Waterfall model: the scientific method of software engineering, (December 2019).

[31] Atlassian. 2019. Git feature branch workflow. https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow.

[32] Moreno Ambrosin, Arman Anzanpour, Mauro Conti, Tooska Dargahi, Sanaz Rahimi Moosavi, Amir M. Rahmani, and Pasi Liljeberg. 2016. On the feasibility of attribute-based encryption on internet of things devices. *IEEE Micro*, 36, (November 2016). DOI: 10.1109/MM.2016.101.

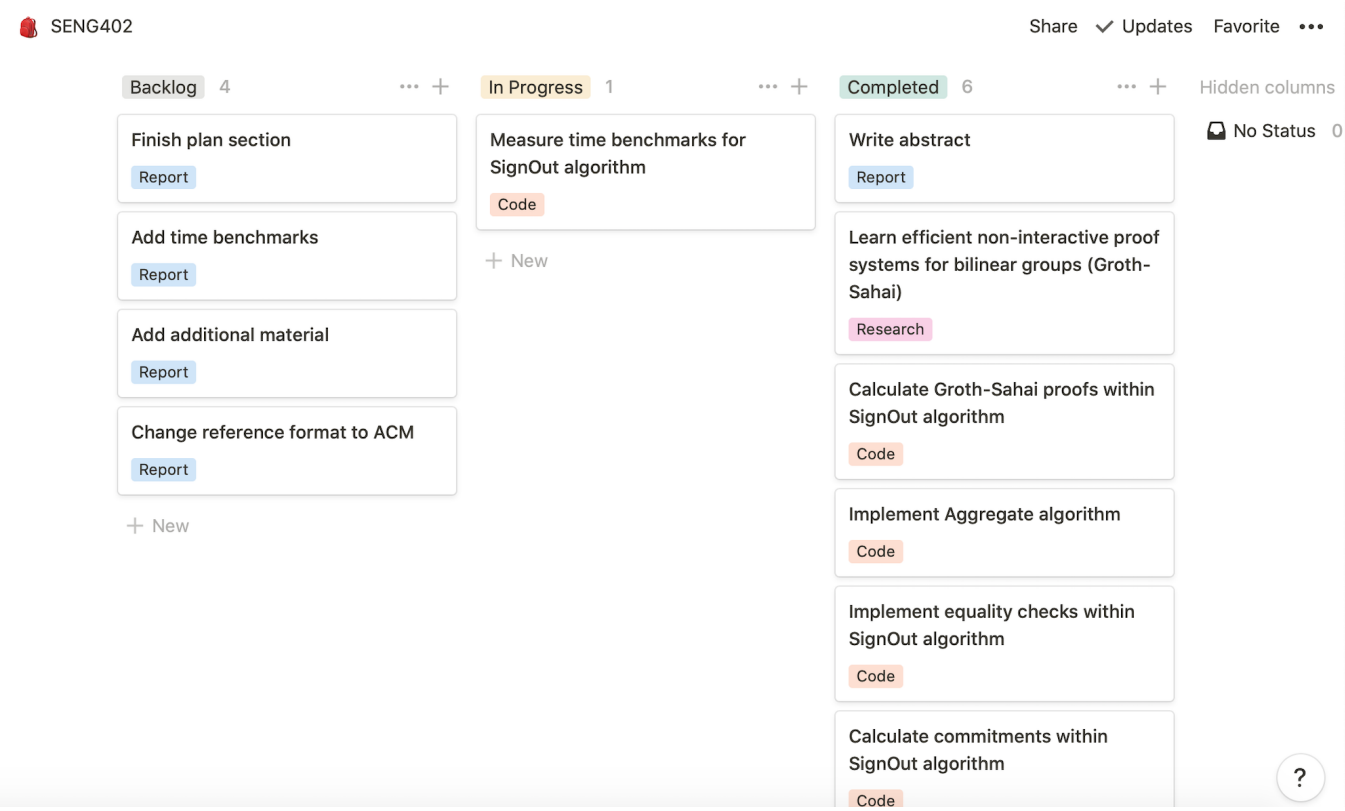

Figure 12: Kanban board for organising tasks

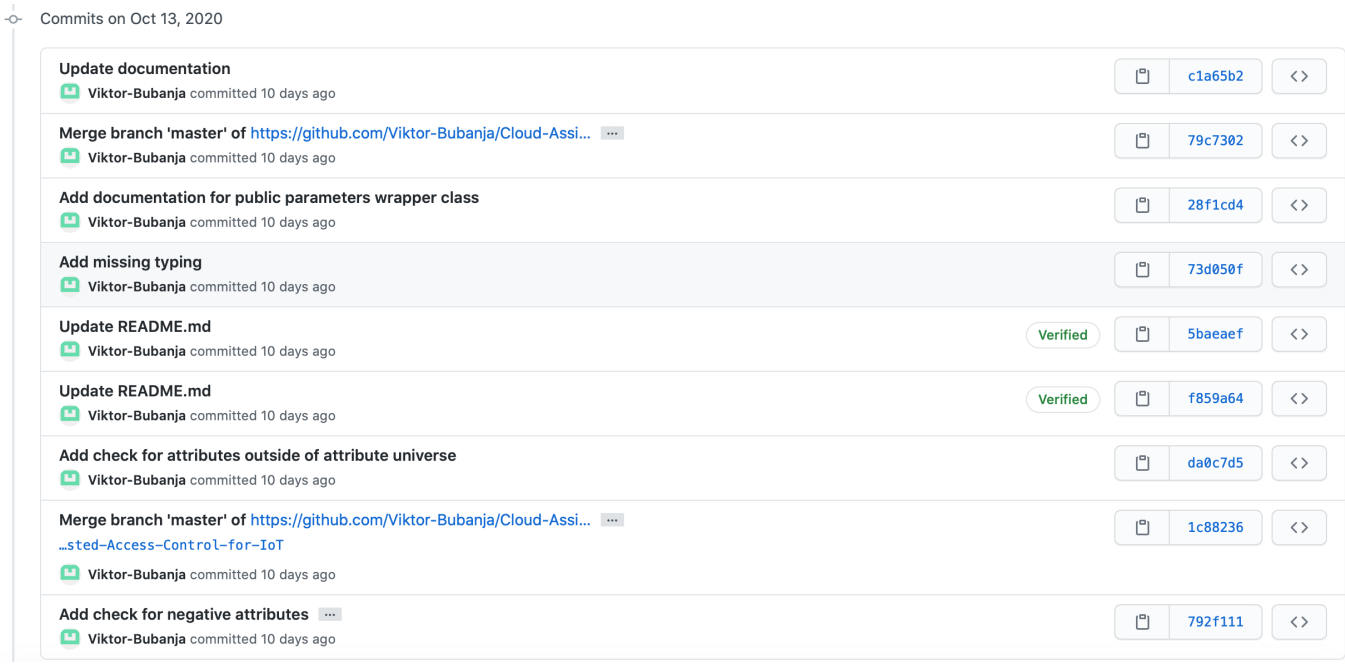

Figure 13: Example commits on Github

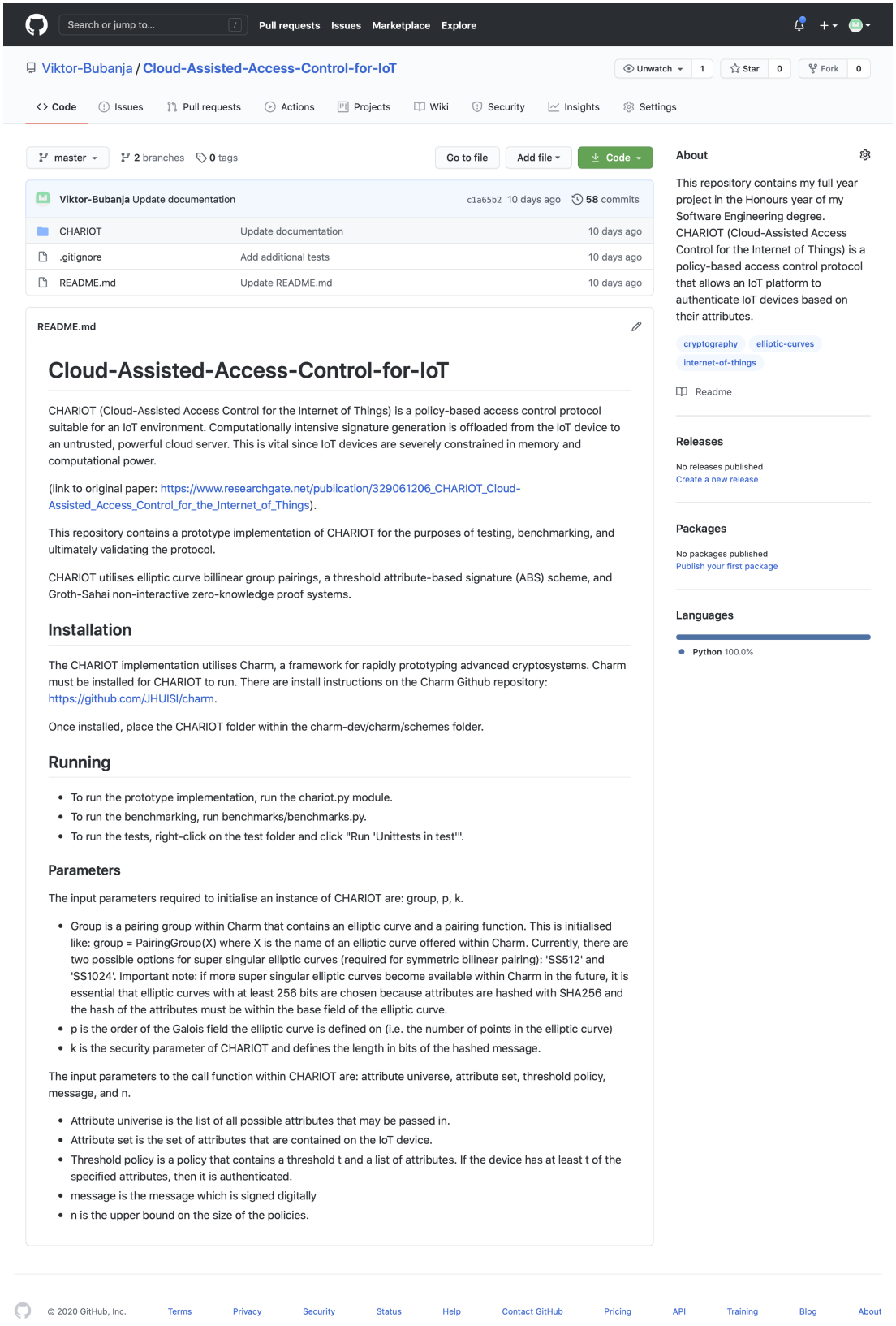

Figure 14: Project repository on GitHub

```python
class TestPolynomial(unittest.TestCase):

    def test_calculate_correct_polynomial_coefficients2(self):
        numbers = [-2, 4, 5, -1]
        polynomial_coefficients = get_polynomial_coefficients(numbers)
        x = 5
        factored_form = reduce(operator.mul, [x + num for num in numbers])
        expanded_form = x ** 4 + reduce(
            operator.add,
            [coefficient * x ** i for i, coefficient in enumerate(polynomial_coefficients)])

        self.assertEqual(factored_form, expanded_form)

    def test_successfully_return_empty_list(self):
        self.assertEqual(get_polynomial_coefficients([]), [])
```

```python
class TestChariot(unittest.TestCase):

    def test_too_few_matching_attributes_raises_exception(self):
        with self.assertRaises(NotEnoughMatchingAttributes):
            attribute_set = [1 for _ in range(6)]
            policy = [i for i in range(6)]
            threshold_policy = ThresholdPolicy(t, policy)
            message = "abcd"
            chariot.call(attribute_universe, attribute_set, threshold_policy, message, n)

    def test_t_equal_to_size_of_policy_succeeds(self):
        attribute_set = [i for i in range(6)]
        policy = [i for i in range(6)]
        threshold_policy = ThresholdPolicy(t, policy)
        message = "abcd"
        output = chariot.call(attribute_universe, attribute_set, threshold_policy, message, n)
        self.assertTrue(output)

    def test_t_smaller_than_size_of_policy_succeeds(self):
        attribute_set = [i for i in range(6)]
        policy = [i for i in range(6)]
        t = 2
        threshold_policy = ThresholdPolicy(t, policy)
        message = "abcd"
        output = chariot.call(attribute_universe, attribute_set, threshold_policy, message, n)
        self.assertTrue(output)
```

```python
class TestAggregate(unittest.TestCase):

    def test_aggregate_calculates_correct_value(self):
        r = 5
        g = group.random(G1)
        gamma = group.random(ZR)
        x_array = [group.random(ZR), group.random(ZR), group.random(ZR)]
        p_array = [g ** (r / (gamma + i)) for i in x_array]
        output = aggregate(x_array, p_array)
        expected_output = g ** (r / reduce(operator.mul, [gamma + i for i in x_array], 1))
        self.assertEqual(output, expected_output)
```

```python
class TestHash(unittest.TestCase):

    def test_hash_outputs_k_bits(self):
        num_bytes = 2
        num_bits = 2 * 8
        output = hash_message(num_bytes, bytes("test", "utf-8"))
        self.assertEqual(len(output), num_bits)

    def test_hash_outputs_type_string(self):
        output = hash_message(2, bytes("random string", "utf-8"))
        self.assertIsInstance(output, str)

    def test_hash_outputs_k_1s_and_0s(self):
        output = hash_message(2, bytes("testing string", "utf-8"))
        self.assertEqual({'0', '1'}, set(list(output)))
```

Figure 15: Examples of unit tests