

# MIT Timely Secure Confessions

Rayden Chia      Anthony Liu      Faraaz Nadeem      Arman Talkar  
{rayden,igliu,faraaz,atalkar}@mit.edu

## Abstract

The "MIT Timely Confessions" page on Facebook is a popular online community for MIT students. The page promises a "completely anonymous" confessions system. However, the privacy of this system is completely dependent on Google Forms, the service used to collect confessions. So, while the administrators of the confessions page may not receive any user information, a user must trust the Google system to not abuse metadata for de-anonymization. Additionally, despite being called MIT Confessions, the page provides no mechanism to verify that a user submitting a confession is indeed part of the MIT community, lowering the quality and authenticity of the page content. In this paper we use MIT OpenId Connect, and Ring Signatures to propose two systems that allow for anonymous and authentic confessions.

## 1 Introduction

In addition to a strong physical sense of community, MIT students interact online on publicly accessible websites like Facebook.com and Reddit.com. The most popular of these online communities, with 3,140 followers and daily updates [1], is the "MIT Timely Confessions" (MTC) Facebook page. MTC provides two services: 1. a form for submitting confessions anonymously and 2. a public page where accepted and moderated confessions are posted daily. This page has drawn so much attention not only because the confessions are entertaining and relatable to MIT students, but also because the Facebook platform allows students to comment on and discuss the confessions.

We have identified 4 goals of a confessions system that make it useful to MIT students.

1. Anonymous: Many of the confessions are of a personal nature and students would not feel comfort-

able posting them if they knew they could be revealed as the confessor.

2. Authenticity: In order for the confessions to be relatable, they should actually come from the MIT community. Otherwise "bait" confessions and confessions of low quality appear in greater proportion to honest ones.
3. Convenient: Submitting a confession should not be hard. Ideally, a student should just have to type the confession in and click submit.
4. Moderated: Not all confessions are appropriate for posting. Some confessions may include names (in a bad light), may not actually be a confession, or be inappropriate in a general "you know it when you see it" sense.

While all the goals are important, we believe anonymity is the foundation of the system. Without it, users would be reticent to submit the confessions that make the page popular, and the lack of content would render the page useless. MTC uses Google Forms to collect submissions. This system is anonymous in the sense that the page administrators do not learn the confessors' identity. However, Google, with Google Account cookies, certainly has the ability to de-anonymize confessors. If this information from Google were to be leaked, the integrity of the system collapses.

We also place strong emphasis on authenticity. Much of the excited conversation around a confession relies upon it being true. While we cannot guarantee this in an anonymous model, we can at least hope that the confessor is an MIT student and will say something relevant to the MIT community. MTC has no provision for this. The submission form is open to the public, so anyone can confess.

In this paper, we use MIT OpenId Connect and Ring Signatures to develop two confessions systems that pri-

oritize anonymity and authenticity, while attempting to maintain convenience and moderation.

The remainder of the paper is organized as follows: Section 2 describes the technologies and cryptographic primitives used in our system. Section 3 describes an initial, simple "single blind privacy" system. Section 4 describes a more complex "double blind privacy" system that trades efficiency for greater plausible deniability. We start with a strawman system and build up to our final design. Section 5 describes our Proof of Concept implementation of the system and Section 6 evaluates the security of the two systems.

## 1.1 Threat Model

We operate under a compromised-server-information model. Our primary fear is that the information exchanged by the user and the system is made public. This fear stems from the recent and highly publicized data breach attacks [2]. So, we assume that this information is always public, and enforce that the system should never be able to associate a user with a confession. We focus on the cryptography necessary to build this system, and assume that connections between the user and the system are secure. In our proof-of-concept implementation we enforce this by explicitly using TLS. We also do not trust our users. We assume they may leak secrets given to them, intentionally or unintentionally.

## 2 Background Information

### 2.1 OpenId Connect

OpenID Connect 1.0 (OIDC) is a secure inter-operable authentication protocol [3] based on the OAuth 2.0 framework. OAuth 2.0 is specified by the IETF in RFCs 6749 and 6750 (published in 2012) and was designed to support the development of authentication and authorization protocols.

OIDC provides a variety of standardized message flows based on JSON and HTTP and enables developers to authenticate their users across websites and apps without having to own and manage password files. It provides a secure verifiable, answer to the question: What is the identity of the person currently using the browser or native app that is connected to me? The platform allows for clients of all types, including browser-based JavaScript and native mobile apps, to launch sign-in flows and receive verifiable assertions about the identity of signed-in users.

At the time of writing, MIT has implemented an OpenID Connect pilot service which allows enabled sites and applications to log the user in using MIT account credentials. The pilot service provides both an OAuth

2.0 authorization server and an OpenID Connect identity provider (OIDC IDP) for use in a wide variety of sites and applications. MIT account holders can log in to the service using their username/password, Kerberos tickets, or MIT certificate. The pilot service is available to everyone at MIT.

### 2.2 Ring Signatures

Ring signatures, initially proposed by Rivest, Shamir and Tauman [5], are a method of signing a message such that it is verifiable that the signer belongs to some predetermined set of users, but it is impossible to identify the signer from the set of users. Ring signatures are not to be confused with group signatures, which uses a central authority or fixed group. Instead, a signer can extemporaneously choose a set of users and include them in a ring signature (even without their permission). Ring signatures have a number of applications, such as leaking secrets in a semi-anonymous way. For instance, consider a member of the White House who wishes to leak some secret information to the press. That staff member of the White House wants to be sure that no one can identify him/her as the signer. However, he/she also wants to convince people that the information is provided by some staff member of the White House. This is achievable with the staff member signing on the message using a ring signature, where the ring includes all staff members of the White House. This proves that only a member of the White House could have signed, but the signer is not personally identifiable.

The security of ring signatures is defined as such: a signature  $\sigma$  on a message  $m$  is associated with a set of public keys  $P = \{pk_1, \dots, pk_n\}$ . Unforgeability is codified by claiming that no adversary without knowledge of any secret key in  $S = \{sk_1, \dots, sk_n\}$  can output  $(m, \sigma)$  where  $\sigma$  is a valid ring signature with respect to  $pk_i \in P$  ( $\forall i$ ). The adversary is given access to a signing oracle and succeeds if it outputs a valid ring signature on any message not queried to its oracle. Regarding anonymity, the requirement is that for any signature  $\sigma$  generated with a secret key  $sk_i$ , the probability that the signing algorithm with  $sk_i$  outputs  $\sigma$  equals the probability that the signing algorithm with  $sk_j$  outputs  $\sigma$ , for all  $i \neq j$ . In other words:

$$\Pr[D_{sk_i}(m) = \sigma] = \Pr[D_{sk_j}(m) = \sigma] \quad \forall i \neq j$$

which gives perfect anonymity by definition.

In this system, we will be implementing a linkable ring signature scheme that satisfies signer-indistinguishability (anonymity), linkability (that two signatures by the same signer can be linked) and spontaneity (there is no group secret, and hence no key setup for secret-sharing) as described in [4].

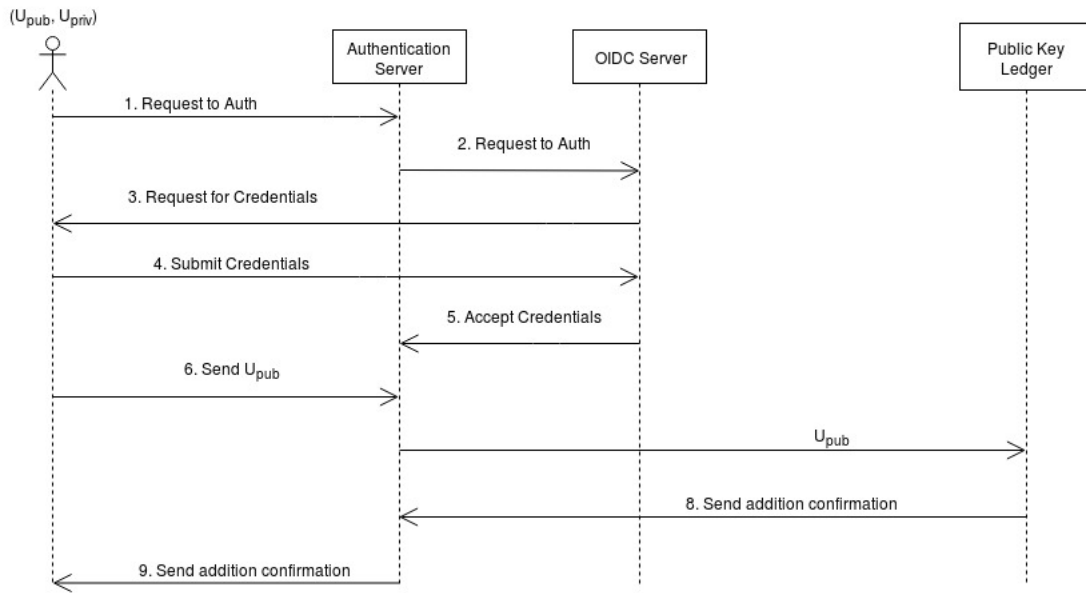


Figure 1: Single Blind Setup System

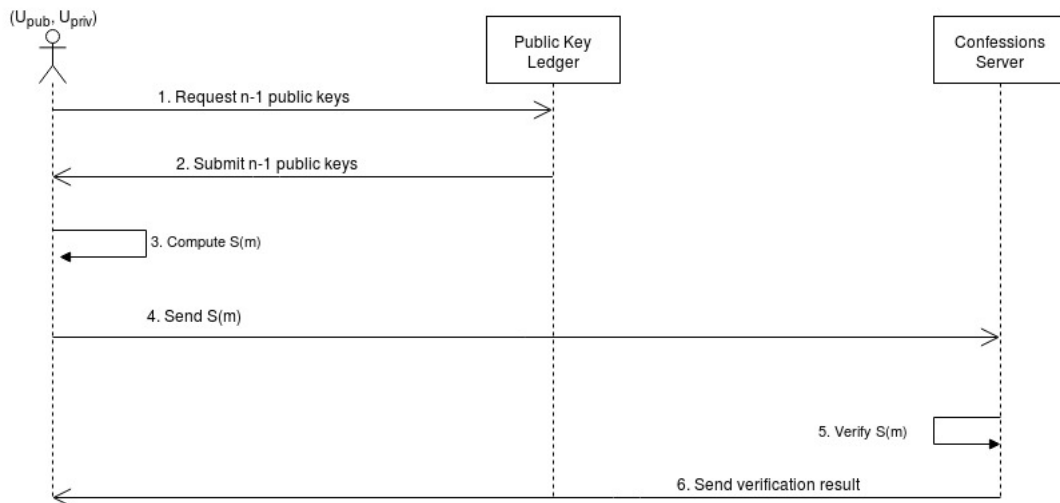


Figure 2: Single Blind Submission System

### 3 Single Blind System

This system uses four principals: the user ( $U$ ), the authentication server ( $A$ ), the confessions server ( $C$ ), and the OpenId Connect server ( $OIDC$ ). To avoid tedious implementation details, we will abstract the  $OIDC$  functionality described in Section 2.1 into an oracle entity. When queried about  $U$ ,  $OIDC$  will respond with whether or not  $U$  is a member of the MIT community.

Figure 1 and Figure 2 detail the system timeline.  $U$ , wishing to submit a confession for the first time, queries  $A$  for access.  $A$  uses the  $OIDC$  oracle to authenticate  $U$  as a valid MIT community member.  $U$  then generates a public/private key pair  $(u_{priv}, u_{pub})$ .  $U$  sends  $u_{pub}$  to  $A$ , which adds  $(U : u_{pub})$  to a public table of public keys.

When  $U$  wishes to send a confession,  $U$  chooses a security parameter  $p$ . This is the number of public keys they will use in the ring signature. The computation of the ring signature is done by the user, so a higher  $p$  results in greater security but more computation.  $U$  accesses the public table of keys on  $A$  to choose  $p - 1$  public keys at random.  $U$  uses these keys to sign confession  $m$  with a ring signature, generating  $S(m)$ .  $U$  then sends  $(S(m), m, p)$  to  $C$ .  $C$  verifies the ring signature and accepts  $m$ .

This system achieves our goals. A user can only be added to the list of possible confessors after authenticating, and thus can only submit a valid confession if they have access to an MIT individual’s private key. The system uses ring signatures, so the identity of the confessor is not obvious, and privacy is maintained. However, we consider this system to be only “single-blinded.” The amount of privacy depends on the security parameter  $p$ . A user without much computation power may choose a relatively low  $p$ . An observer, who sees the signed confession and has background information about the confession could inspect the list of public keys used in the signature (the possible confessors) and use the background information to de-anonymize the confessor. So, while this system is extremely simple, because it is restricted to the relatively small MIT community and has users compute the signature, it offers limited privacy.

### 4 Double Blind System

We will now build up to our Double Blind system that trades efficiency for greater confessor plausible deniability. We use the same principals as in the Single Blind system.

#### 4.1 Strawman solution: one key system

We call our initial approach a “one key system.”  $C$  has a public/private key pair  $(c_{priv}, c_{pub})$ .  $A$  also has this same

key pair.  $U$ , wishing to submit a confession for the first time queries  $A$  for access.  $A$  uses the  $OIDC$  oracle to authenticate  $U$  as a valid MIT community member.  $A$  then gives  $U$   $c_{priv}$ . When  $U$  wishes to submit a confession  $m$ ,  $U$  signs  $m$  with  $c_{priv}$ , creating  $S_{c_{priv}}(m)$  and sends  $(m, S_{c_{priv}}(m))$  to  $C$ .  $C$  then verifies  $S_{c_{priv}}(m)$  and, if valid, accepts  $m$  as a confession to post.  $U$  can submit further confessions with the same key - there is not need to interact with  $A$  again.

Without any malicious actors, this achieves our two main objectives. Firstly, assuming enough users authenticate with  $A$  before any confessions are submitted (this can be arranged by having a “registration period” before the service is active) the system cannot associate any single confession with a particular  $U$ . All registered users have the same signing key, so they are all plausible confessors. Additionally, because we use the  $OIDC$  oracle,  $A$  is only giving the signing key to MIT community members.

However, this scheme breaks with untrusted users. Users may intentionally or unintentionally leak this key. While a singular leak does not compromise anonymity, it does break the authenticity of the system. Now anyone with access to the leak can submit valid confessions to  $C$  and there is no detection mechanism. The main takeaway from this system is the concept of the authentication server giving the user a secret for signing, as opposed to only the user generating a secret.

#### 4.2 Two key system

The “two key system” provides a primitive form of detection. Now, there are two public/private key pairs,  $(c1_{priv}, c1_{pub})$  and  $(c0_{priv}, c0_{pub})$ .  $U$  authenticates as before, but is not immediately given a secret key by  $A$ . First,  $U$  generates a random nonce  $n$ . In their request for the secret,  $U$  submits this nonce. If the nonce ends with a 1, the server gives  $U$   $(c1_{priv}, c1_{pub})$ , and otherwise reveals  $(c0_{priv}, c0_{pub})$ .  $U$  submits a confession  $m$  to  $C$  as before.

Now the server has means for detection. Since users generate random nonces, the ratio of users with each key should be around 1 : 1. Similarly, the ratio of confessions signed by a particular key should be 1 : 1. If one of the keys is leaked, an arbitrary number of people can use that key to sign messages. This would skew the ratio and allow detection. For example, if  $c0_{priv}$  is leaked, the ratio of confessions signed with  $c0_{priv}$  to those signed with  $c1_{priv}$  will be greater than 1. The system can set a threshold of deviation from the expected signing ratio it is willing to tolerate and, upon detecting an intolerable anomaly, invalidate the offending key. On invalidation,  $A$  and  $C$  generate a new key pair to replace the invalidated pair. Users attempting to submit confessions

with this key will be redirected to  $A$ , forcing them to re-authenticate and receive a new secret key.

There are two problems with this scheme. Firstly, a single MIT user could authenticate twice and intentionally use nonces with different parity each time. A user able to leak both keys at the same time poses a threat to the detection system, which relies on only one key being accessible to anyone. This problem is easily solvable. Now,  $A$  maintains a table that matches a registered user to the last valid secret they received. With this, a user cannot register twice. Additionally, the server now knows how many users are issued each secret. This gives the system a more accurate idea of the ratio of confessions they can expect to be signed by each key, and allows the server to dynamically set the tolerance threshold.

The second problem is collusion. Two users can register with nonces of different parity, thus receiving both secrets. They can then collude to release the keys at the same time, compromising detection.

We also note that in the case of invalidation, around half of the users need to re-authenticate and get a new secret. While this can be implemented practically as a background process, this is a significant amount of "collateral damage" - one user may have leaked the key, but 50% of the users now need to do additional work to use the system.

### 4.3 The $N$ key system

We solve the issue of collusion by making collusion harder. Instead of just two key pairs we have  $N$  key pairs. We will now refer to each of these key pairs as "bucket pairs," because users given the same secret are considered to be in the same "bucket." We also introduce a public function  $F$  that maps the user-generated nonces to some bucket key. There are many ways  $F$  could be implemented. The only requirement is that it uniformly maps all possible nonce values to the bucket keys.  $A$  uses  $F$  to determine which key it should give  $U$ . Now, the threshold to completely break the detection system is  $N$  users colluding. Additionally, with  $P$  users, the number of users who need to re-authenticate is dropped from 50% to around  $P/N$ .

While this system solves issues with detection, it creates new privacy issues. In a compromised-server-information model, if the table mapping users to secrets is leaked, an observer can see which users fall into each bucket. The observer, with a confession and accompanying signature, would know which bucket of users signed the confession. Using metadata and background information about the confession and the people in the bucket, the observer could de-anonymize the confessor. Small buckets decrease the privacy of the system.

## 4.4 Final Design

Our final design uses ring signatures and a public ledger. As described in section 2.2, ring signatures allow a signature verifier to assert that the signer was part of some group without revealing which member of the group it was. Our definition of a public ledger is loose, as many different technologies could implement the requisite functionality. We require a publicly readable and appendable ledger with a notion of time. We also require that the ledger central authority be able to remove additions to the server. A centralized blockchain is a heavy-weight but feasible example of this functionality.

The timelines of the three distinct interaction chains are presented in Figure 3, Figure 4 and Figure 5. As in the  $N$  key solution,  $U$  authenticates and receives a secret  $Cn_{priv}$  as before. However, instead of signing a confession with this message,  $U$  generates a personal key pair  $(u_{priv}, u_{pub})$ .  $U$  signs  $u_{pub}$  with  $Cn_{priv}$ , generating  $S_{Cn_{priv}}(u_{pub})$ .  $U$  adds  $(u_{pub}, S_{Cn_{priv}}(u_{pub}))$  to the public ledger.

Instead of detecting secret leaks with confessions as before, secret leaks are based on the number of keys added to the public ledger signed with a particular  $Cn_{priv}$ . Once again, the authorization server uses the table of users to secrets to determine the ideal ratio of signed additions. When a leak is detected the authentication server uses its authority to remove the invalid keys from the ledger.

When  $U$  wishes to submit a confession, they pick a parameter  $p$ . This is the number of public keys they will use in the ring signature. The computation of the ring signature is done by the user, so a higher  $p$  results in greater security but more computation.  $U$  uses the public key associated with each secret as well as  $A$ 's public key to verify additions to the ledger and pick  $p - 1$  valid public keys for inclusion in the ring signature.  $U$  signs  $m$ , generating  $R(m)$  and sends  $(R(m), m, p)$  to the  $C$ .  $C$  verifies that the public keys used in  $R(m)$  are valid and that the signature is valid, and then accepts  $m$ .

In this system, there are two levels of anonymity. Given a confession with security parameter  $p$ , the observer knows the  $p$  public keys that could have submitted the confession. For each public key, assuming the confessor is diligent about choosing public keys signed by different bucket keys, there are  $P/N$  registered users who could have added that entry to the ledger. This "double blinding" makes it harder for an observer to correlate a confession and its associated background information with a small set of individuals who could have written it.

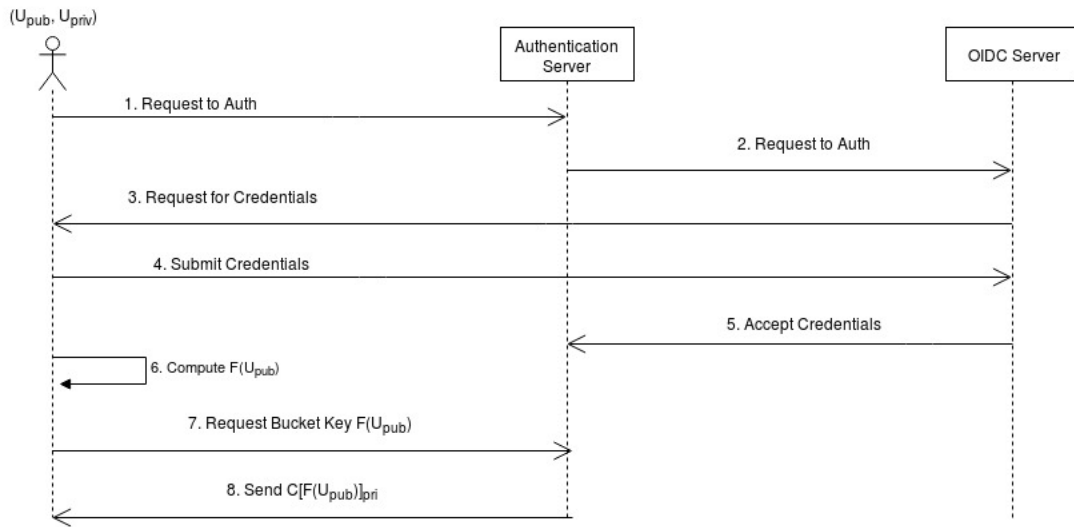


Figure 3: Double Blind Key Distribution System



Figure 4: Double Blind Ledger Add System

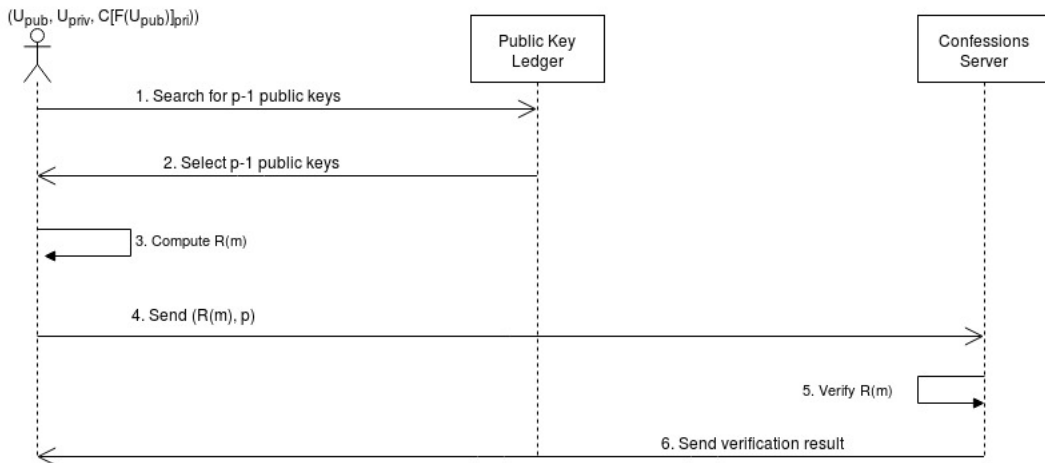


Figure 5: Double Blind Submission System

## 4.5 Anomaly Detection Algorithm

Here we present a simple algorithm for anomaly detection and subsequent key invalidation.  $A$  starts with a dictionary of priors  $P$  mapping bucket secret  $Cn$  to  $u_n$ , the number of users given that secret. This can be constructed using  $A$ 's table mapping users to the secret they were issued. We also define a "threshold multiplier"  $m$  that signifies the tolerance for anomalies. This algorithm is run after an "observation window." During the observation window additions to the public ledger are recorded into  $D$ . This data is fed to the algorithm, which reports whether any bucket keys should be invalidated.

1. Normalize  $P$  by transforming it into a mapping of secret  $Cn$  to  $u_n / \sum_{\forall n} u_n$ .
2. Parse the data in  $D$  to create  $P_o$ , mapping secret  $Cn$  to  $a_n$ , the number of public keys appended to the ledger and signed by  $Cn$  during the observation window.
3. Normalize  $P_o$  by transforming it into a mapping of secret  $Cn$  to  $a_n / \sum_{\forall n} a_n$ .
4. Iterate over all  $Cn$ . If  $(a_n / \sum_{\forall n} a_n) > m * (u_n / \sum_{\forall n} u_n)$  mark  $Cn$  as invalid.
5. Iterate over the public ledger and invalidated all public keys signed with  $Cn$ .

## 5 Implementation

The GitHub repository for our code is located at: <https://github.com/turbomaze/secure-whistle>. This contains client-side code necessary to submit confessions, as well as instructions for doing so. You can also visit [confess.anthony.ai](https://confess.anthony.ai) to view the system endpoints, such as the sample confessions and the public ledger. Appendix A contains screenshots of our proof-of-concept.

A list of endpoints on the server, along with their descriptions, are as follows. These descriptions are available on the home page, along with links to recent confessions.

- GET `/bucket/id`: start the authentication process for getting a private key id
- GET `/private`: after a successful authentication, send the private bucket key
- GET `/public/id`: send the user public key id
- GET `/ledger`: return all the valid public keys in the ledger

- POST `/ledger/add`: add a public key / signature pair to the ledger
- POST `/confess`: confess a message, signed with a ring signature

The server implementation is written in Python using Flask. Some major dependencies include the `ecdsa` Python library, since we implemented ring signatures using elliptic curves. We used the `ECC Linkable Ring Signatures` library.

We expected ring signature and public key computation to be the limiting factor in terms of performance of our implementation. Running 2.4 ghz on one core, for a ring of size 45, signing a message took 12.47 seconds. This is a long amount of time, but we expect this to be an upper bound on our system, which is designed to handle thousands of users. Additionally, the computation can be done in the background, so once the user types and submits their confession, they can multitask while the signature is calculated.

## 6 Security Evaluation

**Confidentiality.** The confidentiality of data-in-transit when sent from the end-user to our servers is protected end-to-end with Transport Layer Security (TLS/SSL) at Layer 4-5 of the OSI model. Our servers are configured to only use algorithms approved under the Federal Information Processing Standard 140-2 (FIPS 140-2), which ensures perfect forward secrecy and strong end-to-end encryption.

**Integrity.** Since the signer computes the ring signature, it follows from section 2.2 that as long as the ring signature was computed correctly (without any malicious interference on the signer's device), the message integrity can be verified using the ring signature and rejected if the verification fails. Modification of the signature is not possible since the signature is computed on the hash of the message, so there is no malleability. In addition, the TLS layer adds another layer of integrity during data transit.

**Authentication and Authorization.** We transfer the risks associated with authentication by making use of MIT OIDC. Our servers will not store any passwords or credentials that may be used to authenticate with other MIT services. In addition, HTTPS certificate public key pinning (HPKP) is implemented on our server to ensure the identity of the MIT OIDC Identity Provider endpoint.

**Anonymity.** In the final design, we make use of the 'double-blinding' method to make it harder for an observer to correlate a confession (and its associated metadata) to a small set of individuals who could have submitted it. Assume that adversaries know the  $p$  public

keys used in signing the confession. For each public key, assuming that signer/confessor is careful about choosing public keys signed by different bucket keys (i.e. using our client), there are  $P/N$  registered users (where  $P$  is the number of users, and  $N$  is the number of keys in the bucket) who could have added that entry to the public ledger. In addition, ring signatures further mix the keys used to prevent revocation of anonymity.

## 6.1 Goals

These are the general confessions system goals stated in Section 1 and our evaluation of how our systems address them:

1. Anonymity: Both systems provide for anonymity. In the event of a compromise, an observer could specify a group of potential confessors for a given confession. However, with a large enough security parameter  $p$  there is enough plausible deniability for one single user in this group to not feel threatened by this compromise.
2. Authenticity: Use of OpenId Connect ensures that only MIT community members can receive secrets and anomaly detection allows us to find and stop breaches of authenticity.
3. Convenient: MTC only requires users to click on a form, enter the confession, and click submit. Our system requires considerably more interaction. However, aside from the initial setup of downloading a client-side package and generating a keypair, all other processes - getting the bucket key, finding keys on the public ledger, computing the ring signature - can be run in the background. Packaged correctly, the user's interface during the submission process can be just as straightforward as MTC's.
4. Moderated: We have not addressed moderation as it is a human issue - there should be someone present to view the valid confessions and filter or edit inappropriate ones. This would be trivial to implement on a production system. In the interim, we can submit our validated confessions to MTC and use them for moderation. In this setup our system would function as a mix net - a single source of confessions with no way to trace a confession to its original confessor.

## References

- [1] ADMINISTRATORS. MIT Timely Confessions Facebook Page. <https://www.facebook.com/timelybeaverconfessions/>, 2018.
- [2] ARMERDING, T. Biggest Data Breaches of the 21st Century. *IOG Communications* (2018).

- [3] DANIEL FETT, RALF KUSTERS, G. S. The web sso standard openid connect: In-depth formal security analysis and security guidelines. <https://arxiv.org/pdf/1704.08539.pdf>, 2017.
- [4] JOSEPH LIU, VICTOR WEI, D. W. Linkable spontaneous anonymous group signature for ad hoc groups. <https://eprint.iacr.org/2004/027.pdf>, 2004.
- [5] RONALD RIVEST, ADI SHAMIR, Y. T. How to leak a secret. <https://people.csail.mit.edu/rivest/pubs/RST01.pdf>, 2001.

## 7 Appendix A

(See next page)

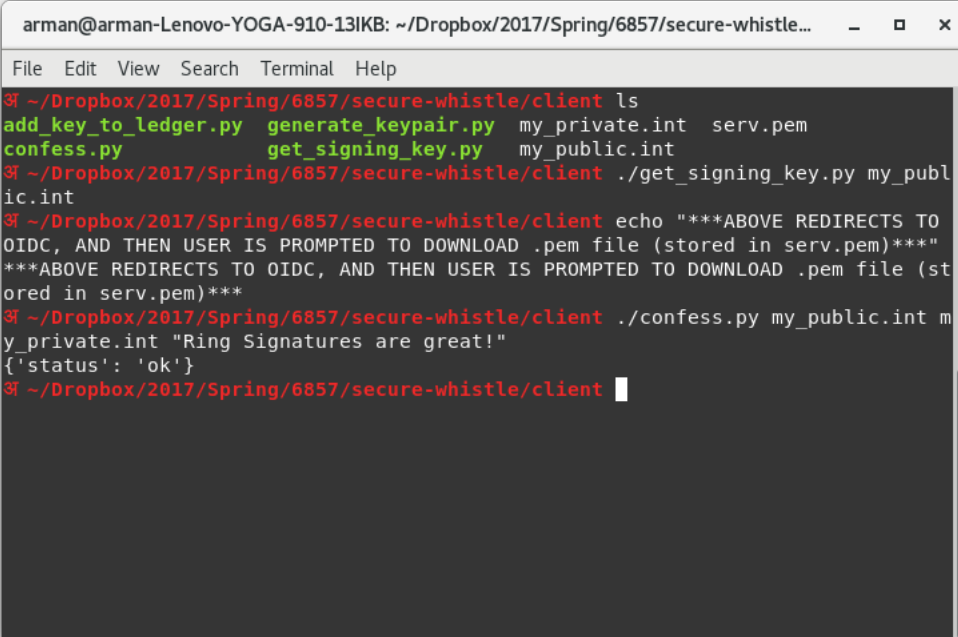


## Secure Confessions

```
2018/05/16 @ 20:58 ive always been afraid that mit confessions knows who i am
2018/05/16 @ 20:16 Josh Noel is so cool.
2018/05/16 @ 20:03 deep down i actually like javascript
2018/05/16 @ 19:39 hello world
```

```
GET /bucket/id start the auth process for getting private key id
GET /private after successful auth, send the private bucket key
GET /public/id send the user public key id
GET /ledger return all the valid public keys in the ledger
POST /ledger/add add a public key / signature pair to the ledger
POST /confess confess a message, signed with a ring signature
```

Figure 6: confess.anthony.ai landing page and existing confessions



```
arman@arman-Lenovo-YOGA-910-13IKB: ~/Dropbox/2017/Spring/6857/secure-whistle... - □ ×
File Edit View Search Terminal Help
arman@arman-Lenovo-YOGA-910-13IKB: ~/Dropbox/2017/Spring/6857/secure-whistle/client$ ls
add_key_to_ledger.py generate_keypair.py my_private.int serv.pem
confess.py get_signing_key.py my_public.int
arman@arman-Lenovo-YOGA-910-13IKB: ~/Dropbox/2017/Spring/6857/secure-whistle/client$ ./get_signing_key.py my_public.int
arman@arman-Lenovo-YOGA-910-13IKB: ~/Dropbox/2017/Spring/6857/secure-whistle/client$ echo "****ABOVE REDIRECTS TO
****ABOVE REDIRECTS TO OIDC, AND THEN USER IS PROMPTED TO DOWNLOAD .pem file (stored in serv.pem)****"
****ABOVE REDIRECTS TO OIDC, AND THEN USER IS PROMPTED TO DOWNLOAD .pem file (stored in serv.pem)****
arman@arman-Lenovo-YOGA-910-13IKB: ~/Dropbox/2017/Spring/6857/secure-whistle/client$ ./confess.py my_public.int my_private.int "Ring Signatures are great!"
{"status": "ok"}
arman@arman-Lenovo-YOGA-910-13IKB: ~/Dropbox/2017/Spring/6857/secure-whistle/client$
```

Figure 7: Proof-of-concept client secret retrieval and confession submission