Encrypting User Data with Attribute-Based Encryption Using Privacy Policies

Milo Watanabe
Boston College

April 2014

Abstract

As the internet and cloud services have pervaded our lives in nearly every aspect, one of the biggest issues facing the populace today is finding the best ways to protect our privacy. An important part of protection for the average user of a service is their privacy policy: essentially the only way today to let a client define how their data is used by the server. But often a client is not given a choice, and sometimes their policy is not even followed. We present here an way of encrypting user data such that they first create a privacy policy, and their data is protected from even the service unless their privacy requirements are met by the service.

1 Introduction

Protecting user privacy is a huge problem facing any internet company, and a huge issue for every client using those technologies. Tech giants, like Facebook and Google, and smaller app developers, like QuizUp publisher Plain Vanilla, have faced a large amount of backlash based on their sometimes-shady usage of client data.[1] More and more solutions are emerging for giving users finer control over their data, though.
Recently in the EU, a push has gone through the European Parliament to

[1]Lawler, Ryan. "QuizUp Sends Personal User Info To Strangers, Company Says Bug Contributed To Weakened Security." TechCrunch, 25 Nov. 2013.

create tighter regulations for privacy and enforce larger penalties for non-compliance.[2] Another solution involves a new programming language that inherently puts constraints, defined by privacy policies, on certain variables that correspond to user data.[3] These are good solutions, and they are important steps towards putting the client first. One of the biggest issues with the current state of privacy, though, is that even if a user edits their privacy settings, often they are still left unclear as to how their data is managed, and frankly can't tell if it was mismanaged anyway.

The solution presented here will not go so far as to ensure the data is used properly, but it will create a 'promise' that the service will implicitly make with every instance of the data being used. This scheme uses a type of Key-Policy encryption called Attribute-Based Encryption. It requires a client to share its data with a server, which will perform some action with the data, and a third-party authority to create and store the keys necessary for the data to be encrypted and decrypted. The prototype for such a system has been built as a web app called privateBook, in which clients create an account and set a privacy policy, then can write posts that they can view if signed in to their own account. This simple demonstration serves to show a practical way of ensuring data is encrypted for all parties, and can only be seen as a result of the service adhering to a user's privacy policy.

2 Data Encryption

At a high level, Attribute-Based Encryption (ABE) is a form of Public-Key cryptography which encrypts data with an access tree, or policy. A party attempting to decrypt the data must present an attribute list, which will successfully allow decryption provided the attributes satisfy the tree. We will discuss exactly how ABE uses policies and attributes to encrypt and decrypt data, and then how the privateBook implementation uses this type of encryption.

[2]Bajaj, Vikas. "Imagine if Companies Had to Ask Before Using Your Data." Taking Note. The New York Times, 13 Mar 2014.

[3]J. Yang, K. Yessenov, A. Solar-Lezama. A Language for Automatically Enforcing Privacy Policies. *POPL 2012.*

2.1 Public-Key Cryptography

In many systems of encryption there is one key that is used to encrypt and decrypt messages, called a cipher. This is analogous to one key that may lock or unlock a door. Public-Key cryptography is also called asymmetric cryptography, due to its use of two separate keys, one used to decrypt and one used to encrypt a message. In this form, there is both a public key and private key. The public key is used to encrypt data, while the private key is used to decrypt. The keys are generated with some function that creates a public key which does not allow a foreign party to discover the private key. This is based on certain problems which are essentially impossible to solve computationally, such as large integer factorization problems.

2.2 Attribute-Based Encryption

The implementation of ABE is provided by Charm, a Python framework providing many different crypto systems. There are four portions of the ABE scheme: the parameter setup, encryption, key generation, and decryption.

Setup The setup simply generates random groups, which are stored as PK, the public parameters, and MK, the private key. PK is the ABE version of a public key. The two keys here are bilinear group generators raised to randomly chosen exponents. These are stored as Python dictionaries.

Encryption Encryption uses PK, γ, the access structure, along with the string message meant to be encrypted, M. A dictionary, E is created, which is the encrypted cipher text. γ, a string of a boolean expression, is the policy that must be satisfied for decryption.

Key Generation Key Generation uses a list of attributes, $\mathbb{A}$, and MK to generate a decryption key, D. This is the ABE version of a private key, used to decrypt. $\mathbb{A}$ is the list of attributes that must satisfy γ in order for D to successfully decrypt E.

Decryption Decryption takes E, the ciphertext containing γ, PK and D. It applies D to E in order to decrypt the message. If $\mathbb{A}$ satisfies γ, in the sense of the attributes fulfilling the boolean expression included in D, the message will decrypt.

2.3 Example Usage

An example following the paper describing ABE's fine-grained access capabilities can demonstrate how this system would play out if used in exactly the way described above.[4]
If there were a system for storing activity logs on a network, this data would need to be protected by an encryption scheme that would vary for different types of information and different types of activity. Here, ABE encryption would be useful, as each log could be stored with a specific access policy: say "user is Bob or Alice AND the date is between September 2010 and May 2014 AND the activity is related to updating or changing the financial information of projects". This information would then be encrypted with this policy. Anyone investigating the logs could then be given a secret key with a specific access list with their properties: user name, title, security clearance, activity type, date, etc. ABE would then only allow the analyst to access information if their information fit with the policy: all other data, which is not pertinent to their work, will be unaccessible.

3 ABE Privacy Prototype

The ABE prototype used here can apply to any service which uses client data to perform some function, whether it be like Facebook, just storing and displaying messages, text or multimedia, or something like Google Maps, which uses client location info to display a map or directions. In a sense, the prototype applies ABE in a backwards fashion: rather than encrypting with a policy, and users attempt to decrypt by presenting their attributes, here the attribute list is created and clients present their policies. Figure 1 displays the process.

[4]J. Bethencourt, A. Sahai, B. Waters. Ciphertext-Policy Attribute-Based Encryption.

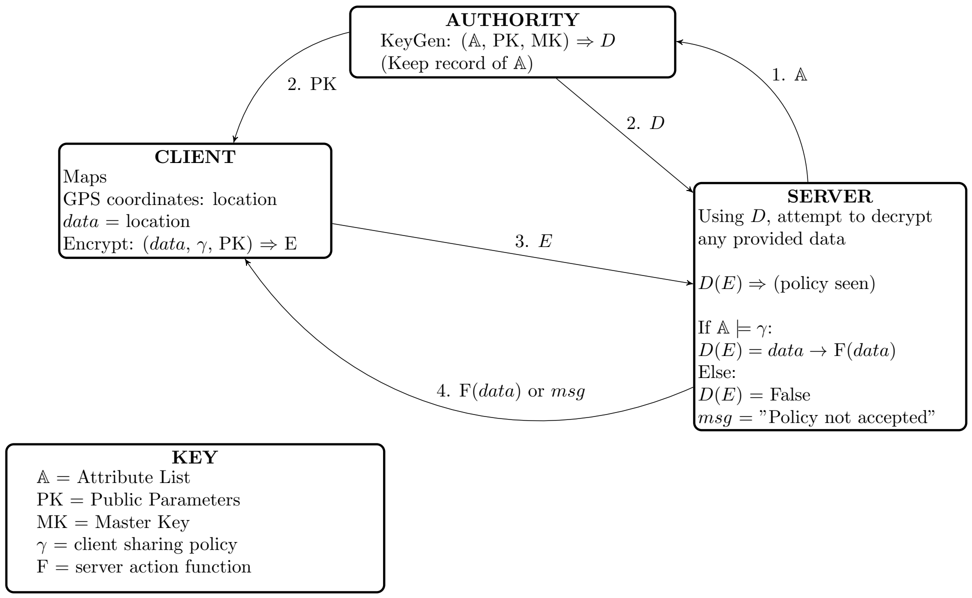

AUTHORITY
KeyGen: ($\mathbb{A}$, PK, MK) $\Rightarrow D$
(Keep record of $\mathbb{A}$)
1. $\mathbb{A}$
2. PK
2. D
CLIENT
Maps
GPS coordinates: location
$data$ = location
Encrypt: ($data$, γ, PK) $\Rightarrow$ E
3. E
SERVER
Using D, attempt to decrypt any provided data
$D(E) \Rightarrow$ (policy seen)
If $\mathbb{A} \models \gamma$:
$D(E) = data \rightarrow \mathrm{F}(data)$
Else:
$D(E)$ = False
msg = "Policy not accepted"
4. F($data$) or msg
KEY
$\mathbb{A}$ = Attribute List
PK = Public Parameters
MK = Master Key
γ = client sharing policy
F = server action function

Figure 1: Prototype encryption/decryption process

3.1 Parties Involved

In the prototype, there are three relevant parties involved: the server, which provides the service, a client, who uses the service, and the authority, a third party that creates, distributes and stores the keys for both parties. The server creates default settings for privacy, saving them as an attribute list. The client creates their own privacy settings, which is saved as a policy. The keys generated from these values are managed by an authority party. This is necessary as another layer of security for the client: if an authority generates the public and private keys, the client can be sure that the keys are generated properly. If a server generated the keys, they could possibly leave themselves a backdoor. The authority also would then have the ability to check the server's data use with their default privacy policy, and confirm that the server was truly using data in the way they had claimed.

3.2 The Full Process

There are four steps of this scheme: server registration, key distribution, client encryption, and server response.

1. Server Registration This is the first step, and only happens once. The server registers with the Authority, naming itself and providing a list of attributes, $\mathbb{A}$, which is a list of the default values created for a privacy policy. The Authority stores $\mathbb{A}$.

2. Key Distribution This happens once, in three parts. The Authority generates PK and MK, the public parameters and master key, which are used to generate D. $\mathbb{A}$, PK and D are stored by the Authority. Then, D is sent to the server, which will use it to decrypt data, and PK is sent to the client, who uses it to encrypt data.

3. Client Encryption This will happen with every client request. The client encrypts by providing their data, γ, their personal privacy policy, and PK. This yields E, a cipher text, which is sent to the server. The server then stores the data as a cipher text.

4. Server Response This happens after every client request. The server stores the encrypted data, and attempts to decrypt it by applying the decryption key D to E. This will return either the original data or False. If $\mathbb{A}$ satisfies γ, the data will be properly decrypted, and the server can perform whatever functions it needs with the data and responds to the client indicating success. If the policy is not successfully satisfied by the attribute list, the server responds to the client indicating failure.

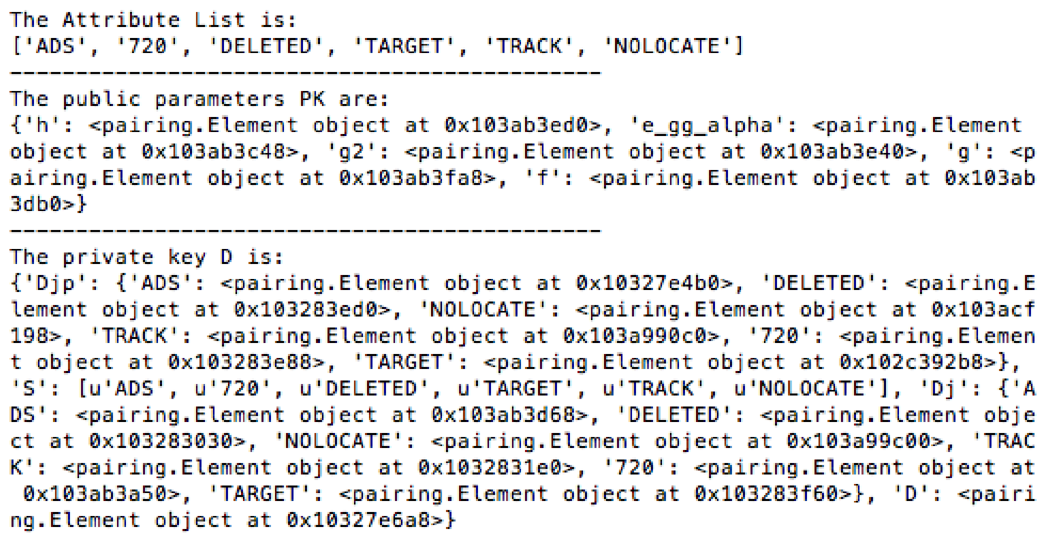
4 privateBook

The prototype of this scheme mimics an online diary: a user can make their own account, then write notes to themselves that will display on their page. The idea of using ABE is that, in decrypting the data, the server must know the privacy policy of the user presenting it. Thus successful decryption becomes analogous to an active promise that the service has read and will follow the user's wishes for privacy, rather than potentially ignoring a policy saved in some database.
There are three parts of the site: a server registration page, on which the server creates its attribute list and the Authority stores the keys and $\mathbb{A}$ in an Authority table, a client registration page, where a client creates his or her privacy policy, storing γ and their identification information in a Policy

table, and a homepage, where the client can write their updates and have them displayed; each update is stored by the server as a cipher text with a user ID in a Posted Data table. We will demonstrate each of the four steps in the privateBook implementation.

4.1 privateBook Server Registration

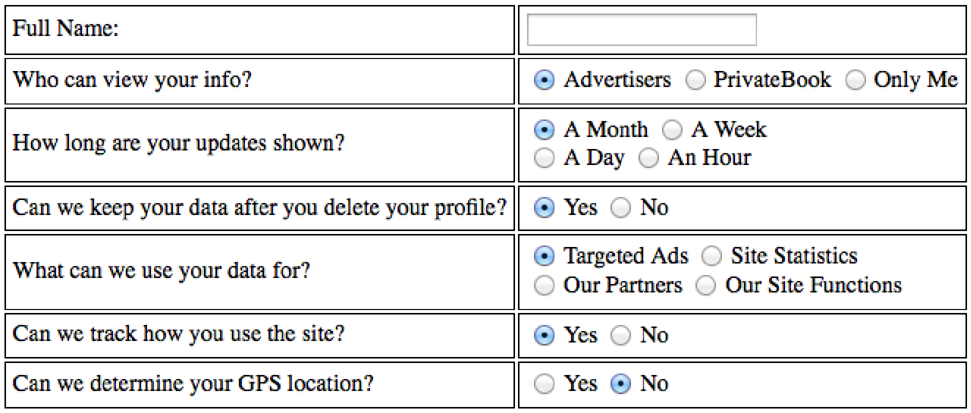
The privateBook server registers its name and default privacy settings with the Authority, which saves them as an attribute list, and then creates PK and D, the public parameters and private key. Figure 2 shows the printed values created from the default privateBook privacy values.



```
The Attribute List is:
['ADS', '720', 'DELETED', 'TARGET', 'TRACK', 'NOLOCATE']
---------------------------------------------
The public parameters PK are:
{'h': <pairing.Element object at 0x103ab3ed0>, 'e_gg_alpha': <pairing.Element
object at 0x103ab3c48>, 'g2': <pairing.Element object at 0x103ab3e40>, 'g': <p
airing.Element object at 0x103ab3fa8>, 'f': <pairing.Element object at 0x103ab
3db0>}
---------------------------------------------
The private key D is:
{'Djp': {'ADS': <pairing.Element object at 0x10327e4b0>, 'DELETED': <pairing.E
lement object at 0x103283ed0>, 'NOLOCATE': <pairing.Element object at 0x103acf
198>, 'TRACK': <pairing.Element object at 0x103a990c0>, '720': <pairing.Elemen
t object at 0x103283e88>, 'TARGET': <pairing.Element object at 0x102c392b8>},
'S': [u'ADS', u'720', u'DELETED', u'TARGET', u'TRACK', u'NOLOCATE'], 'Dj': {'A
DS': <pairing.Element object at 0x103ab3d68>, 'DELETED': <pairing.Element obje
ct at 0x103283030>, 'NOLOCATE': <pairing.Element object at 0x103a99c00>, 'TRAC
K': <pairing.Element object at 0x1032831e0>, '720': <pairing.Element object at
 0x103ab3a50>, 'TARGET': <pairing.Element object at 0x103283f60>}, 'D': <pairi
ng.Element object at 0x10327e6a8>}
```

Figure 2: Authority values (Attribute list, PK, D)

These default values are based off the client privacy policy creation page, which is shown in Figure 3.

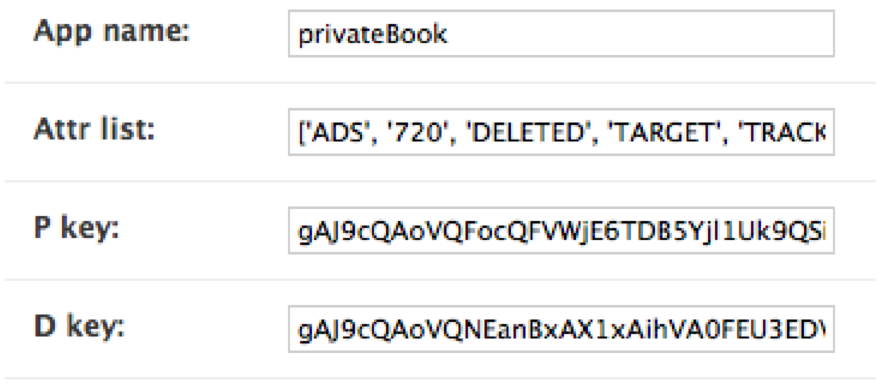


Full Name:	
Who can view your info?	[✓ Advertisers] PrivateBook Only Me
How long are your updates shown?	[✓ A Month] A Week A Day An Hour
Can we keep your data after you delete your profile?	[✓ Yes] No
What can we use your data for?	[✓ Targeted Ads] Site Statistics Our Partners Our Site Functions
Can we track how you use the site?	[✓ Yes] No
Can we determine your GPS location?	Yes [✓ No]

Figure 3: Client Policy creation page

4.2 Authority Key Storage and Distribution

The Authority then serializes the dictionary keys and stores them, shown in Figure 4.



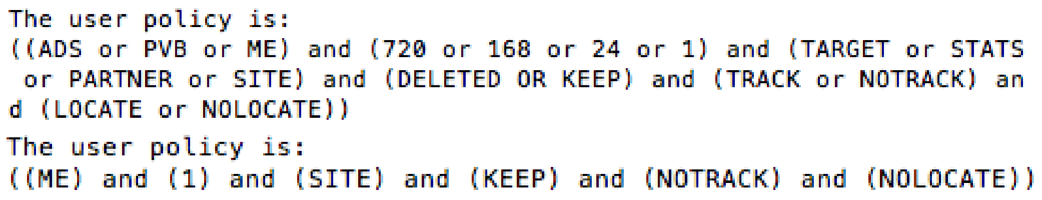
App name:	privateBook
Attr list:	['ADS', '720', 'DELETED', 'TARGET', 'TRAC
P key:	gAJ9cQAoVQFocQFVWjE6TDB5YjI1Uk9QS
D key:	gAJ9cQAoVQNEanBxAX1xAihVA0FEU3ED

Figure 4: Authority key storage

These keys are then provided to the server and the clients by way of database queries to the Authority table. In a real distributed system, the Authority would send back D to the server, which would store it on their own. The clients would then receive PK upon registering for the service, an added complexity that this prototype does not delve into.

4.3 Client Privacy Policy Creation

The client, to become a privateBook user, registers themselves with a name and their privacy policy, as shown in Figure 3. They are then given their own ID, simply the order in which they were created, as their password, and username of their full name. These design choices are meant to simplify the account creation process: the focus of the prototype is not concerned with creating users as much as maintaining the privacy of their data, so privateBook has foregone security measures that should be in place in practical applications. Two policies are shown in Figure 5. They are strings of boolean expressions. The first is a policy that has the least privacy restrictions as given by the service, and the second has the strictest settings. The first includes an 'or' for every type of privacy setting which includes all possible attributes; this allows for any default setting in the attributes list to satisfy the policy. The second, stricter policy includes the fewest attributes, making it the most difficult to satisfy.



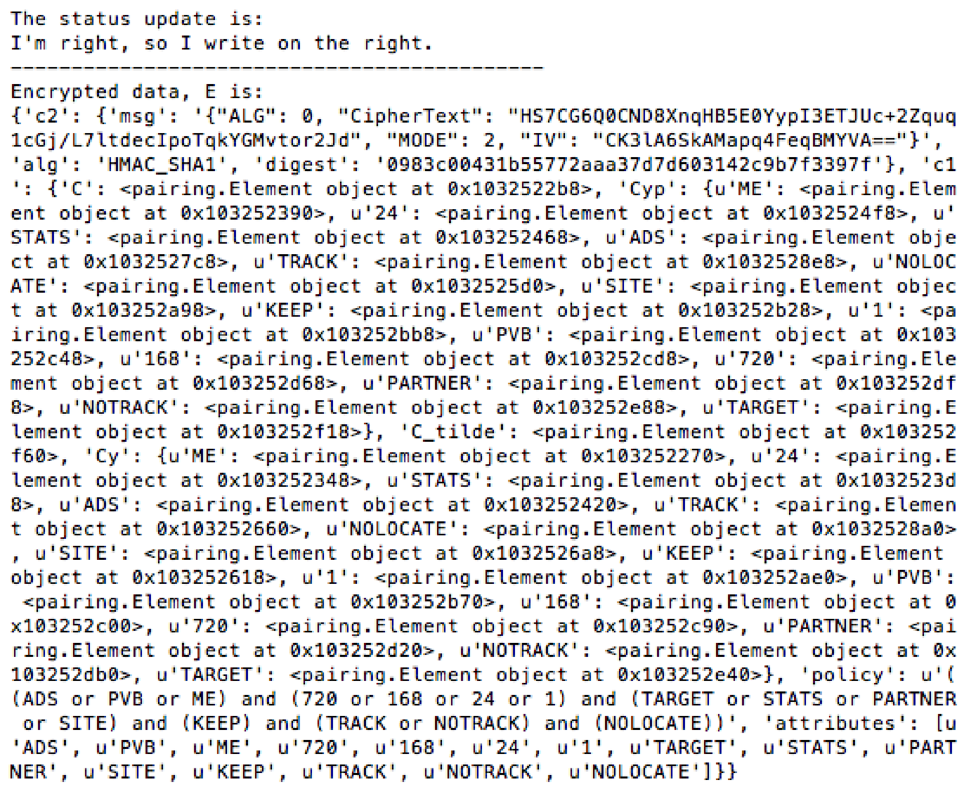
```
The user policy is:
((ADS or PVB or ME) and (720 or 168 or 24 or 1) and (TARGET or STATS
 or PARTNER or SITE) and (DELETED OR KEEP) and (TRACK or NOTRACK) an
d (LOCATE or NOLOCATE))
The user policy is:
((ME) and (1) and (SITE) and (KEEP) and (NOTRACK) and (NOLOCATE))
```

Figure 5: Least strict and most strict user privacy policies

This method of creating a default attribute list was to allow the server to fail. In practice, a service that intends to have wide use should never fail, and thus an early idea was to simply let each attribute be the type of privacy setting it was, and store the values elsewhere. Then the server would still need to decrypt the data and thus inherently read the privacy policy, but that would offer no real difference from what exists currently, as the actual privacy settings would still be stored somewhere and potentially ignored, as they are now. So a difficulty arose in finding an appropriate method to allow one attribute list to successfully satisfy many different policies. This method was chosen because it is based on an idea of having different levels of privacy, where a baseline is chosen by the service and clients can be more or less strict; more, rendering the service useless for that user, and less, allowing for full use.

4.4 Client Encryption and privateBook Response

The client then makes a status update, much like a Facebook one, and submits that. The data is encrypted with the update, a string, along with the user's privacy policy and the public key PK. This results in a long cipher text, which is stored in privateBook's database as a serialized dictionary. The results of this process are shown in Figure 6. E is stored in the database, then the page is reloaded, displaying all of the user's updates. The server, upon loading of the user's homepage, first attempts to decrypt their data. If it is successful, the date, time and actual content are shown. If it is unsuccessful, the content is replaced with: "Your status could not be displayed: this service does not support your privacy policy!" Thus this service will not work for the client if their privacy settings are too strict, and therefore their policy is not satisfied by privateBook's default attribute list.



```
The status update is:
I'm right, so I write on the right.
--------------------------------------------
Encrypted data, E is:
{'c2': {'msg': '{"ALG": 0, "CipherText": "HS7CG6Q0CND8XnqHB5E0YypI3ETJUc+2Zquq
1cGj/L7ltdecIpoTqkYGMvtor2Jd", "MODE": 2, "IV": "CK3lA6SkAMapq4FeqBMYVA=="}',
'alg': 'HMAC_SHA1', 'digest': '0983c00431b55772aaa37d7d603142c9b7f3397f'}, 'c1
': {'C': <pairing.Element object at 0x1032522b8>, 'Cyp': {u'ME': <pairing.Elem
ent object at 0x103252390>, u'24': <pairing.Element object at 0x1032524f8>, u'
STATS': <pairing.Element object at 0x103252468>, u'ADS': <pairing.Element obje
ct at 0x1032527c8>, u'TRACK': <pairing.Element object at 0x1032528e8>, u'NOLOC
ATE': <pairing.Element object at 0x1032525d0>, u'SITE': <pairing.Element objec
t at 0x103252a98>, u'KEEP': <pairing.Element object at 0x103252b28>, u'1': <pa
iring.Element object at 0x103252bb8>, u'PVB': <pairing.Element object at 0x103
252c48>, u'168': <pairing.Element object at 0x103252cd8>, u'720': <pairing.Ele
ment object at 0x103252d68>, u'PARTNER': <pairing.Element object at 0x103252df
8>, u'NOTRACK': <pairing.Element object at 0x103252e88>, u'TARGET': <pairing.E
lement object at 0x103252f18>}, 'C_tilde': <pairing.Element object at 0x103252
f60>, 'Cy': {u'ME': <pairing.Element object at 0x103252270>, u'24': <pairing.E
lement object at 0x103252348>, u'STATS': <pairing.Element object at 0x1032523d
8>, u'ADS': <pairing.Element object at 0x103252420>, u'TRACK': <pairing.Elemen
t object at 0x103252660>, u'NOLOCATE': <pairing.Element object at 0x1032528a0>
, u'SITE': <pairing.Element object at 0x1032526a8>, u'KEEP': <pairing.Element
object at 0x103252618>, u'1': <pairing.Element object at 0x103252ae0>, u'PVB':
 <pairing.Element object at 0x103252b70>, u'168': <pairing.Element object at 0
x103252c00>, u'720': <pairing.Element object at 0x103252c90>, u'PARTNER': <pai
ring.Element object at 0x103252d20>, u'NOTRACK': <pairing.Element object at 0x
103252db0>, u'TARGET': <pairing.Element object at 0x103252e40>}, 'policy': u'(
(ADS or PVB or ME) and (720 or 168 or 24 or 1) and (TARGET or STATS or PARTNER
 or SITE) and (KEEP) and (TRACK or NOTRACK) and (NOLOCATE))', 'attributes': [u
'ADS', u'PVB', u'ME', u'720', u'168', u'24', u'1', u'TARGET', u'STATS', u'PART
NER', u'SITE', u'KEEP', u'TRACK', u'NOTRACK', u'NOLOCATE']}}
```

Figure 6: The plaintext update and it's corresponding encrypted cipher text