Secure Applications: Confidentiality and Integrity for App Deployment

Vy-An Phan and Crystal Yan
{pimasta314,crystalzyan}@berkeley.edu

1 Introduction

Modern companies are extremely interested in collecting client data for many purposes such as usage analysis, machine learning model training, and targeted advertising. However, there are many problems regarding privacy, anonymization, and safe handling of data. New legal frameworks like GDPR or CCPA, and general anti-discrimination and processing restriction rules, require companies to prove their compliance.

Our system seeks to provide a secure way for companies to store and execute their data, while also providing an easy way to prove personal responsibility.

2 Motivation

With the advent of big data, the data practices of web services has changed from curated data collection to the exhaustive logging of every minutiae of user behavior. Data on users is being collected on a massive scale, from a user's purchases on external websites through Facebook's Beacon program [7], to the keyword search trends and user click-through rates essential to Google's core AdSense business model. Even daily user household activity data can now be collected by smart appliances and relayed to remote web servers through the Internet-of-Things. The advertising profit to be made from pooling together user data into in-depth character profiles has spawned a whole new data broker industry, the so-called "trusted third parties" to which web services are notorious for selling off user data to [11].

But while personally identifiable data (such as social security numbers or medical data) is already a subject regulated for data propriety, it can be too easy to overlook the power and potential for misuse that behavioral data can have. The formation of not just demographic, but psychographic profiles, have proven how a user's social media activities, opinions, and interests can be compiled under a psychological lens to predict and target users on a most intimate level. By training a LASSO linear regression model on the Facebook Likes of users, web services are able to predict percentile scores of a user's personality on the "Big Five/OCEAN" traits of Openness, Conscientiousness, Extraversion, Agreeableness, and Neuroticism, known as the most scientifically-verifiable personality model in psychology today. With just 10 Facebook Likes, this psychographic model can predict undisclosed user behavior, such as their drinking habits, better than the user's own colleague can. With just 70 Facebook Likes, the model can outperform the user's roommate or friend, with 150 Likes it outperforms the user's family member, and with 300 Likes it outperforms the user's own spouse [13]. Advertisers in particular have long since caught on to the power of psychographic data, as a separate study shows that psychographic behavioral targeting is 670% more effective in click-through rates than traditional marketing [2].

Unfortunately, as web services ever broaden their capacity on collecting and analyzing user data, the resulting breakouts of data misuse ever justify the growing concerns for user privacy. The compilation of character profiles have enabled algorithms, integral in consequential processes such as loan verification or recidivism-based criminal sentencing, to inadvertently engage in racial or socio-economic discrimination known as "digital redlining" [10]. More intentionally, health insurance companies have been guilty of flagging and taking action on insurance members who engage in unhealthy purchases, including ice cream and plus-sized clothing [12]. Web services have even been known to engage in the predatory practice of compiling lists of vulnerable consumers experiencing what has been termed as "life triggers," from divorces to pregnancies to the vehicular death of their own child [4].

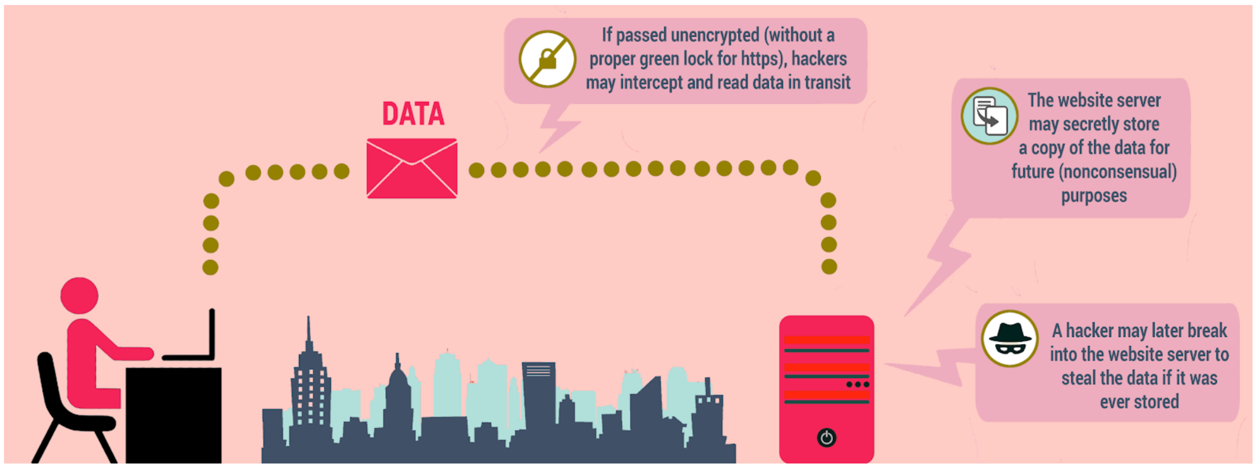

If passed unencrypted (without a proper green lock for https), hackers may intercept and read data in transit
DATA
The website server may secretly store a copy of the data for future (nonconsensual) purposes
A hacker may later break into the website server to steal the data if it was ever stored

Figure 1: There are multiple points along an online transaction where user data may be leaked or misused.

The most publicized case of psychographic data misuse is the case of Cambridge Analytica illicitly retaining the Like data on Facebook users that participated in an online 2015 OCEAN personality quiz/study. Since Facebook had no oversight mechanism in place, Cambridge Analytica was able to then sell the gathered user data to the Donald Trump 2016 presidential campaign, to be used for political targeting and voter manipulation [6]. This data leak was the result of both untrustworthiness on the part of the web service, and a lack of data collection transparency. Although it was Cambridge Analytica and not Facebook who acted in bad faith, the data itself was collected through Facebook's Graph API, a feature that included data on quiz-taking users' Facebook Friends without their consent. The scope of data collection was thus widened from 270,000 to an unsuspecting 87 million compromised users [3].

And while machine learning has inspired the creation of a multitude of convenient web services such as recommendation engines or advanced image search, it has also increased the ways that data can be weaponized and misused. In 2012, computer vision made a significant breakthrough with the creation of AlexNet, the first convolutional neural network model to showcase the power of machine learning in image recognition tasks [9]. In 2018, this very same image recognition technology was misused to predict the LGBT status of non-consenting users indirectly from the phrenology of their face, demonstrating how machine learning can facilitate discrimination from data as accessible as users' profile photos [8].

The prevalent misuse of user data in recent years has resulted in widespread user mistrust in data practices of online platforms. Following the Cambridge Analytica scandal, 59% of polled Internet users consider Facebook to be a net negative for society [1]. For web services in general, 54% of Internet users believe online tracking is harmful and invades their privacy, and 27% of Internet users would never give out personal information to a web service if given a choice. The call for punitive action on websites that violate data privacy is strong as well: an overwhelming 94% of Internet users advocate for some form of punishment, with 11% in favor of site owner imprisonment, and 26% in favor of site closure [5].

The end result is that a technical solution is needed to restore user trust in how their data is being utilized by web services. Given how even innocuous data may be weaponized through inference and machine learning, transparency is needed not only in what data is being collected, but how that data is being used. This transparency can only be achieved through public scrutiny, where a verifiable record of all data usage by web services must remain available for audit.

User data must be protected both from misuse by untrusted web servers, and from leaks to unauthorized third parties. Given that there are many points along a web connection where data may be exposed to either, all stages of an online transaction must be secured from data leakage or misuse [Figure 1]. First, during web transmission, unencrypted user data is vulnerable to eavesdropping and sniffing attacks by local or on-path attackers. Second, when user data enters the website-controlled servers, illicit copies of the data may be retained by the server itself, a la Cambridge Analytica. Third, any user data stored on the website's server may later be stolen if the server gets compromised by a hacker.

To cover all above concerns, a solution must not only produce an immutable record of data usage activities, but also mathematically guarantee confidentiality and integrity throughout the entire data collection process. We thus propose the following solution, utilizing blockchain to fulfill the former requirement, and utilizing both encryption and trusted hardware to fulfill the latter.

3 Solution

Our solution utilizes a combination of cryptographic techniques and blockchain to ensure both confidentiality and integrity in user data transmissions/processing [Figure 2], as well as a means for validating exactly what a web service company has done with the user data, whether to the public or to an audit. Confidentiality is the property that only authorized parties are capable of reading the user data, while integrity is the property that data cannot be tampered or modified by untrusted parties.

The cryptographic techniques in question that will be explained in this section are encryption, hashes, and enclaves. Although normally, these cryptographic techniques alone are enough to secure the confidentiality and integrity of the user data during transmission, there is no means to verify in a future audit that the web service in question has indeed conducted itself honestly and have not made illicit copies when processing past user data. We thus utilize the immutable and trustless nature of the blockchain to provide a record of all code run on the web service's servers that ever had access to the user data. The blockchain element is thus the key and inseparable piece in our solution that enables the detection of any user data violations on the part of the web service.

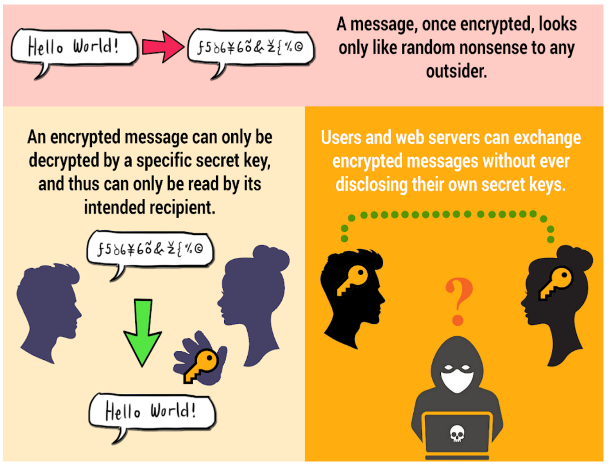

Hello World!
ʃ5δ6¥6ŏ&ž{%◎
A message, once encrypted, looks only like random nonsense to any outsider.
An encrypted message can only be decrypted by a specific secret key, and thus can only be read by its intended recipient.
ʃ5δ6¥6ŏ&ž{%◎
Hello World!
Users and web servers can exchange encrypted messages without ever disclosing their own secret keys.
?

Figure 3: Encryption allows for the confidential transmission of user data between a user and a receiver that possesses a secret key.

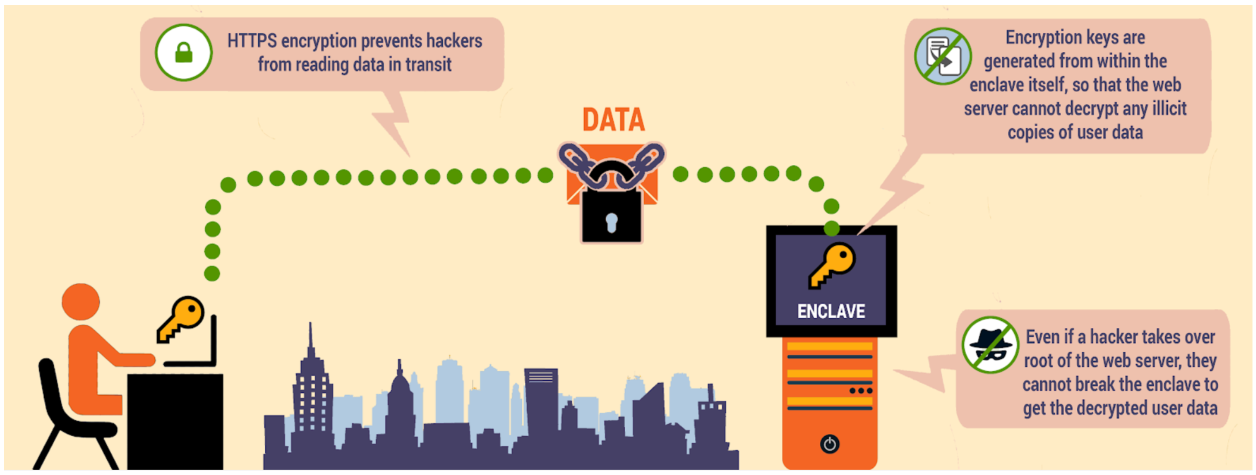
In order to secure online transmission of user data between the user and the web service, we utilize encryption [Figure 3]. Encryption is the process of converting an intended message, called the plaintext, into a form that appears indistinguishable from randomized bits and is unreadable to outside parties, known as the ciphertext. Decryption is the process that reverses ciphertext back into its readable plaintext state, and can only be done by possessing a secret known as a key. Even if an unauthorized party knows the algorithm used to generate the encrypted message, the unauthorized party cannot decrypt the ciphertext without obtaining the secret key, nor can an unauthorized party make educated guesses to partially decrypt the ciphertext. The contents of the message are only readable to parties that are authorized/in possession of the secret key, and thus encryption assures the confidentiality of the message.

When it comes to key generation, there are two forms of encryption: symmetric encryption, and asymmetric encryption. Symmetric encryption, such as AES, is a form of encryption in which the same secret key is used to be able to encrypt or decrypt a message. In our use case, this would mean that both the user and the web service would be able to decrypt a message, but that a shared secret key must also have to be established between the user and the web service prior to the first encrypted/secured communication. A new secret key would also have to be generated for each individual user that communicates with the web service to prevent users from being able to decrypt each other.

In asymmetric encryption, such as RSA, the recipient of an encrypted message possesses two keys: a public key and a private key. Only the private decryption key is the secret, while the public encryption key may be broadcasted to the public and reused among users so that anyone may send an encrypted message readable only to the possessor of the private key, in our case, the web service.

However, asymmetric encryption is much slower than symmetric encryption. Therefore, asymmetric encryption is generally only used to securely exchange fixed-size symmetric keys, which are then used for general communication. This is the standard implementation in HTTPS, the well-accepted protocol for ensuring end-to-end confidentiality, integrity, and authentication (a third property that verifies the identity of the web service to the user) in user data transmissions. Our solution is not necessarily an alternative to HTTPS, but an extension of it. Standard HTTPS alone would mean that the web server running the web service is in possession of the secret key, as it is the intended and trusted recipient of user data transmissions. However, in our case, the web server itself is also untrusted, as it may make copies of the data to sell to outsiders, or perform operations on the user data outside of acceptable perimeters. The web server may even be susceptible to compromise by third-party attackers. We thus must move the secret key from the hands of the web server to an environment that can be trusted. We thus utilize trusted hardware, known as an enclave.

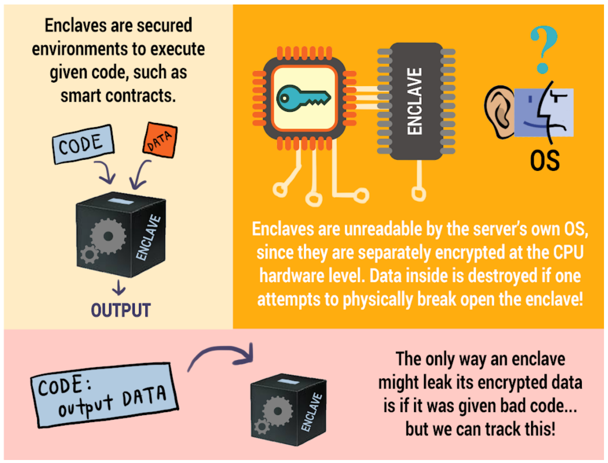
A preexisting alternative to our solution is to use what is known as homomorphic encryption. Homomorphic encryption is a special form of encryption that preserves


HTTPS encryption prevents hackers from reading data in transit
DATA
Encryption keys are generated from within the enclave itself, so that the web server cannot decrypt any illicit copies of user data
ENCLAVE
Even if a hacker takes over root of the web server, they cannot break the enclave to get the decrypted user data
Figure 2: With our solution, we secure data from both the untrusted web server, and from eavesdroppers or hackers.

mathematical properties of the plaintext in its ciphertext form. Homomorphic ciphertext, while still appearing as random bits, would then allow for mathematical operations and data processing to be applied when the user data remains in its encrypted ciphertext state. There are varying degrees of homomorphic encryption, depending on what mathematical operations must remain available to be performed by the web service. The most minimal form of homomorphic encryption only preserves ordering (where if $x >= y$ then $enc(x) >= enc(y)$), which would only permit operations such as sorting to be performed on the data. Partial forms of homomorphic encryption that exist include Paillier, which preserves addition, and ElGamal, which preserves either multiplication or addition. Even fully-homorphic encryption solutions exist, in which for any function $f(x,y)$, $f(enc(x), enc(y)) = enc(f(x,y))$.

The intention of homomorphic encryption is for web servers to never decrypt the user data while still processing the data to perform their web service. Homomorphic encryption would then theoretically be an alternative solution to both encryption and enclaves in our solution. However, homomorphic encryption is not a practical solution generalizable for all web services, which must retain full functionality on the data. The current forms of fully homomorphic encryption are 10^6 times slower to process than unencrypted data, which is an unacceptable performance cost for most web services.

The faster option to homomorphic encryption, and what is used in our solution, is the hardware enclave [Figure 4]. An enclave is a secure execution environment that minimizes the trusted computing base in our solution from the entire web server to only the CPU processor and data processing code. Enclaves exist only within temporary system memory, where code and user data is stored under another layer of encryption by a hardware secret key accessible only to the CPU package itself. User data only ever exists in a decrypted form inside the enclave, for which access to is so restricted that the operating system and all other running code on the web server cannot access it. The user data also is only retained while the enclave is processing on it, as data will be deleted inside the enclave once finished. The only information that comes out of the enclave in a readable state to the web service is the final output of the data processing code uploaded into the enclave.


Enclaves are secured environments to execute given code, such as smart contracts.
CODE
DATA
ENCLAVE
OUTPUT
ENCLAVE
?
OS
Enclaves are unreadable by the server's own OS, since they are separately encrypted at the CPU hardware level. Data inside is destroyed if one attempts to physically break open the enclave!
CODE: output DATA
ENCLAVE
The only way an enclave might leak its encrypted data is if it was given bad code... but we can track this!
Figure 4: Enclaves act as a black box for user data processing to occur without any spying or tampering by the web server itself.

We generate and store secret keys for decrypting user data transmission only from within the enclave, thus user data still remains unreadable and encrypted to the web server as it passes the user data into the enclave for processing by the CPU. If the user data must then leave the enclave, it is reencrypted by the enclave to be again unreadable by the web server.

If the web server attempts to make an illicit copy of the user data when transferring it to/from the enclave, that illicit copy would remain in an useless, encrypted state, as the secret key remains inaccessible inside the enclave. If the web server attempts to write malicious code to access and leak the decrypted data from inside the enclave, that malicious code must be passed into the enclave by our solution SDK and will be recorded down permanently in our blockchain ledger.

If a hacker later manages to take over root of the web server and attempt to steal the user data, the hacker similarly cannot access the enclave's secret key without passing through our blockchain ledger, and can only see user data as unreadable ciphertext from outside of the enclave. In fact, even if the site owner or other attacker attempts to physically access the enclave to obtain its data, the encrypted data itself is stored on temporary memory that will be irrevocably flushed and deleted upon any physical tampering event.

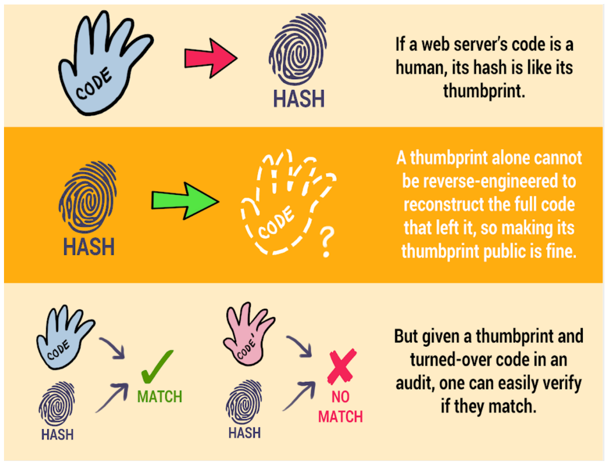

CODE
HASH
If a web server's code is a human, its hash is like its thumbprint.
HASH
CODE
?
A thumbprint alone cannot be reverse-engineered to reconstruct the full code that left it, so making its thumbprint public is fine.
CODE
HASH
MATCH
CODE'
HASH
NO MATCH
But given a thumbprint and turned-over code in an audit, one can easily verify if they match.

Figure 5: Hashes allow web service propietary code to remain secret, but generate a publishable fingerprint that can verify the original code in an audit.

Hashes [Figure 5] are the output of passing in a message, or in this case the body of the web service's code, through what is known as a cryptographically strong hash function. A cryptographically strong hash function is a one-way function that maps variable input to seemingly random bits; thus a hash appears as unreadable as ciphertext is to an outsider. However, cryptographically strong hash functions contain the property that they are pre-image resistant, in that while the hash of x, $hash(x)$, is easy to compute, it is intractable to invert and solve for x from $hash(x)$. Unlike encryption, there exists no secret key to reverse or decrypt the hash in some manner. Additionally, cryptographically strong hash functions express the property of collision resistance, where it is also intractable to find for some alternative code body x' that leaves the same hash as original code body x, such that $hash(x) = hash(x')$.

Because hashes are deterministic functions, the hash left by the original code body will be the same hash each time. Thus, a hash of the code computed at enclave upload time will match a hash of the original code computed when handed over in an audit. However, if a dishonest web company attempts to modify their code to cover any incriminating lines of code in an audit, the modified code would inherently output a distinctly different hash. And due to collision resistance, the web company will be unable to find a modified version of their code that somehow replicates the same hash as the original. A court will then be able to verify the integrity of the audited code by comparing the hash stored in the private mode blockchain with a hash computation right at audit time and seeing if they match.

Evidently, the final piece of our solution is the blockchain itself, which secures web service code allowed inside of the enclave by keeping a permanent record to detect malicious behavior. However, while our blockchain is intended to be a public chain appended by many different web companies, company code is often meant to be proprietary and kept secret.

Our solution thus offers a private mode to web companies, where instead of storing each update of their code body in readable form on the public blockchain, they may store hashes of their code on the blockchain instead [Figure 6]. More information about how the hash signature is generated will be discussed in Section 4.

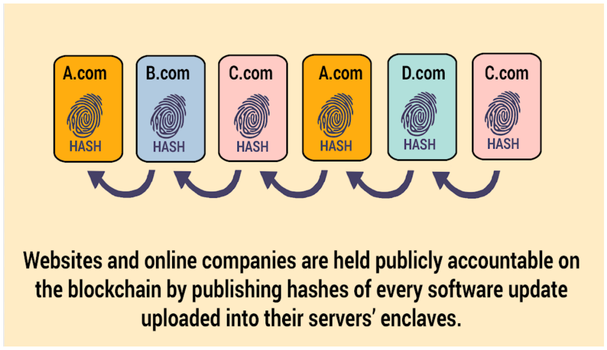

A.com
HASH
B.com
HASH
C.com
HASH
A.com
HASH
D.com
HASH
C.com
HASH
Websites and online companies are held publicly accountable on the blockchain by publishing hashes of every software update uploaded into their servers' enclaves.

Figure 6: Blockchain ensures proper usage of private user data by keeping record of all code that ever ran in the enclave as evidence.

For the most part, we expect the blockchain to simply update normally and remain private. In the case a company is ever accused of abuse of data, however, we can verify to a court that the code run in the enclave (and displayed on the blockchain) corresponds to the code a company has provided. In order to maintain confidentiality of code while providing a means for verifying the integrity of code turned over by the web company in an audit, we use hashes, which act as a fingerprint or sort of 'signature' of the original code.

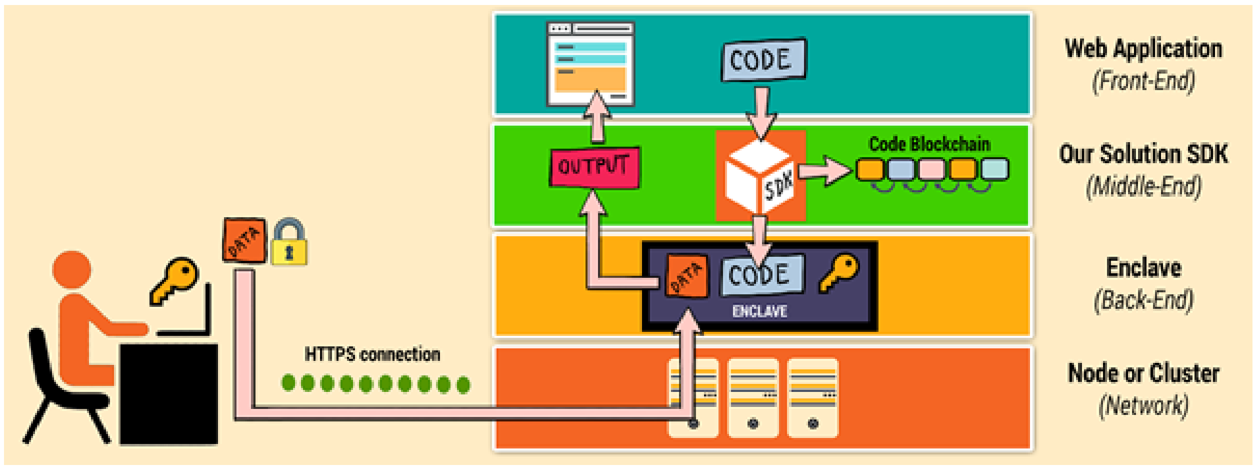


Figure 7: The four main layers are, from top to bottom: the web application, the blockchain, the hardware enclave, and the network.

For example, suppose Company C is accused of malicious activity. All client data is accessible only within the enclave, so the only way Company C would have been able to read the client data in plaintext is if the code that they were running in the enclave was malicious. However, all the code they have provided to the court when audited is seemingly benign.

Normally this would only prove that Company C possessed benign code, not that they ran it. However, if they used our service, all we need to do is take the signature of the benign code and verify that it matches the signature displayed on the public blockchain. If it does, this would mean that Company C did indeed run only their benign code in the enclave.

4 Implementation

Our system is built around four main layers: the web application, the blockchain, the secure hardware enclave, and the network [Figure 7]. For the sake of simplicity we will focus mostly on the second layer, where the blockchain is actually implemented. We assume that we have access to a preexisting network and a secure hardware enclave such as Intel SGX.

The idea is for companies to be able to run their code on their clients' private data within the secure hardware enclave. Assuming that the clients can securely connect to the server (that is, the modern public key and network security infrastructure works as intended), and the hardware enclave is implemented correctly and runs properly, the client data will be protected from misuse by both malicious actors as well as the company itself.

When not in use, all the user data will be stored encrypted. The data is only decrypted within the enclave; while within the enclave, nothing else can access it without the proper permissions. We therefore only need to worry about the code running in the enclave itself, which is determined by the company. The blockchain layer ensures that the code running in the enclave and the signatures displayed on the web application can be securely verified.

The code for the blockchain itself is split into two main parts. `blockchain_utils.py` [Appendix II] corresponds to Layer 2 of Figure 7 and contains the backend code for building the blockchain itself, and takes care of formatting and organizing the stored files [Figure 8].

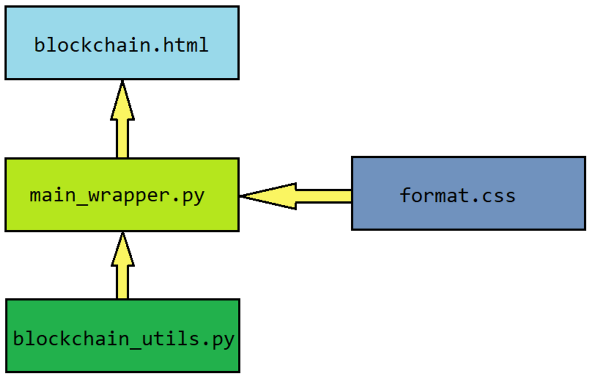
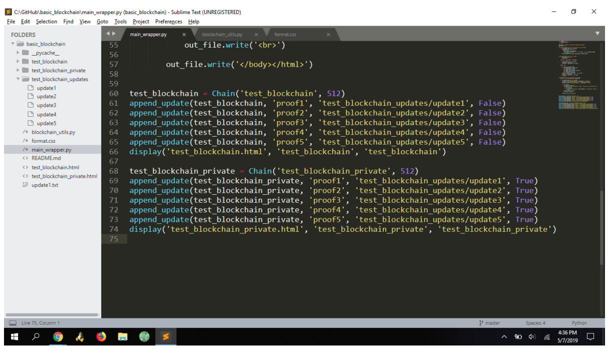


Figure 8: The main wrapper uses functions from a utility file and a formatting file to generate the final web page, which can be displayed publicly.

One important thing to note is that a company would not want to reveal their intellectual property to the public. Therefore, it is also important that we do not leak the plaintext input when generating the update files in our backend. Our service offers a "private mode" which makes it impossible to leak a company's code while still making it possible to verify the validity of said code should the need arise [Figure 9].

We do so by using a salted hash — that is, we append a random string (the "salt") to the body of a piece of code, and then take a hash over the whole thing. (This is the same standard method used to store passwords securely.) The random salt and the resulting hash are then both publicly displayed on the web application.



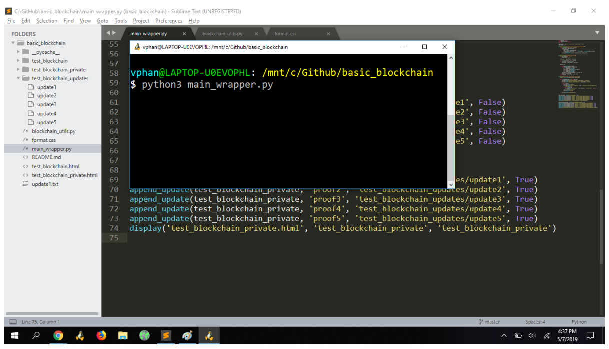
```
        out_file.write('<br>')

    out_file.write('</body></html>')


test_blockchain = Chain('test_blockchain', 512)
append_update(test_blockchain, 'proof1', 'test_blockchain_updates/update1', False)
append_update(test_blockchain, 'proof2', 'test_blockchain_updates/update2', False)
append_update(test_blockchain, 'proof3', 'test_blockchain_updates/update3', False)
append_update(test_blockchain, 'proof4', 'test_blockchain_updates/update4', False)
append_update(test_blockchain, 'proof5', 'test_blockchain_updates/update5', False)
display('test_blockchain.html', 'test_blockchain', 'test_blockchain')

test_blockchain_private = Chain('test_blockchain_private', 512)
append_update(test_blockchain_private, 'proof1', 'test_blockchain_updates/update1', True)
append_update(test_blockchain_private, 'proof2', 'test_blockchain_updates/update2', True)
append_update(test_blockchain_private, 'proof3', 'test_blockchain_updates/update3', True)
append_update(test_blockchain_private, 'proof4', 'test_blockchain_updates/update4', True)
append_update(test_blockchain_private, 'proof5', 'test_blockchain_updates/update5', True)
display('test_blockchain_private.html', 'test_blockchain_private', 'test_blockchain_private')
```

Figure 9: When appending updates to the blockchain, one can specify a public (False) or private (True) mode.

Due to the nature of a one-way function, it is mathematically impossible to reverse a company's proprietary code based on the hash alone. The random string provides an additional layer of security by ensuring that the same piece of code will appear different in different update blocks. That way, simply examining the public blockchain does not reveal if a company used the same piece of code twice in two different updates (e.g., if they reverted back to a previous update).

To verify a signature of some questioned code, we simply append the "salt" listed on the public blockchain to the code in question and take the hash of the whole thing. If the salted hash of the questioned code matches the salted hash on the public blockchain, the code is verified.

Ideally, our customers would never have to deal with `blockchain_utils.py` [Appendix II] directly. Instead, they interact with `main_wrapper.py` [Appendix I], which is the interface between Layer 2 and Layer 1 [Figure 10].



```
vphan@LAPTOP-U0EVOPHL: /mnt/c/Github/basic_blockchain
$ python3 main_wrapper.py
```

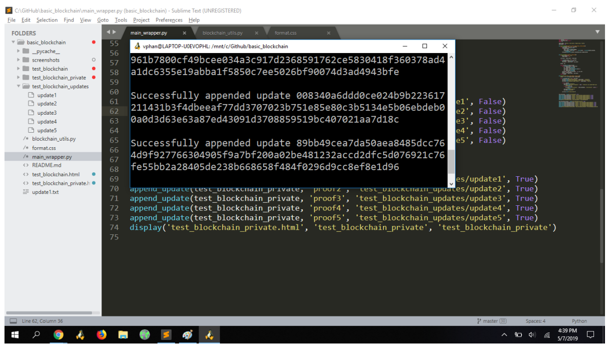
Figure 10: Running the script.

`main_wrapper.py` [Appendix I] contains simplified, user-friendly, intermediary functions that call the more complicated, machine-friendly, base functions in `blockchain_utils.py` [Appendix II] to append update blocks or read from an existing blockchain. It can also export an HTML file for a blockchain for public display on a web application by using `format.css` [Appendix III].

For every new update submitted by the customer, a block is appended to the blockchain to mark the occasion. A single Block consists of the block body (in our case, either the update code in the public version, or the salted hash signature in the private version), the hash of the previous block, the proof of work, and the timestamp. The identity of the Block is the hash over all of the previous four fields.

When there is a new update, a Block is created for that update and a corresponding file created for that Block. In order for a Block to be successfully created, the provided proof of work must also verify properly. A company can choose what sort of proof-of-work algorithm they wish to use, e.g. Bitcoin, Ethereum, Algorand, or some other provably secure algorithm.

A Blockchain is a collection of block files. Apart from the individual update blocks, we also have a single metadata file storing the most recent update. To read the blockchain, simply start from the most recent update and follow the previous block hash until the head is reached, similar to reading a linked list.

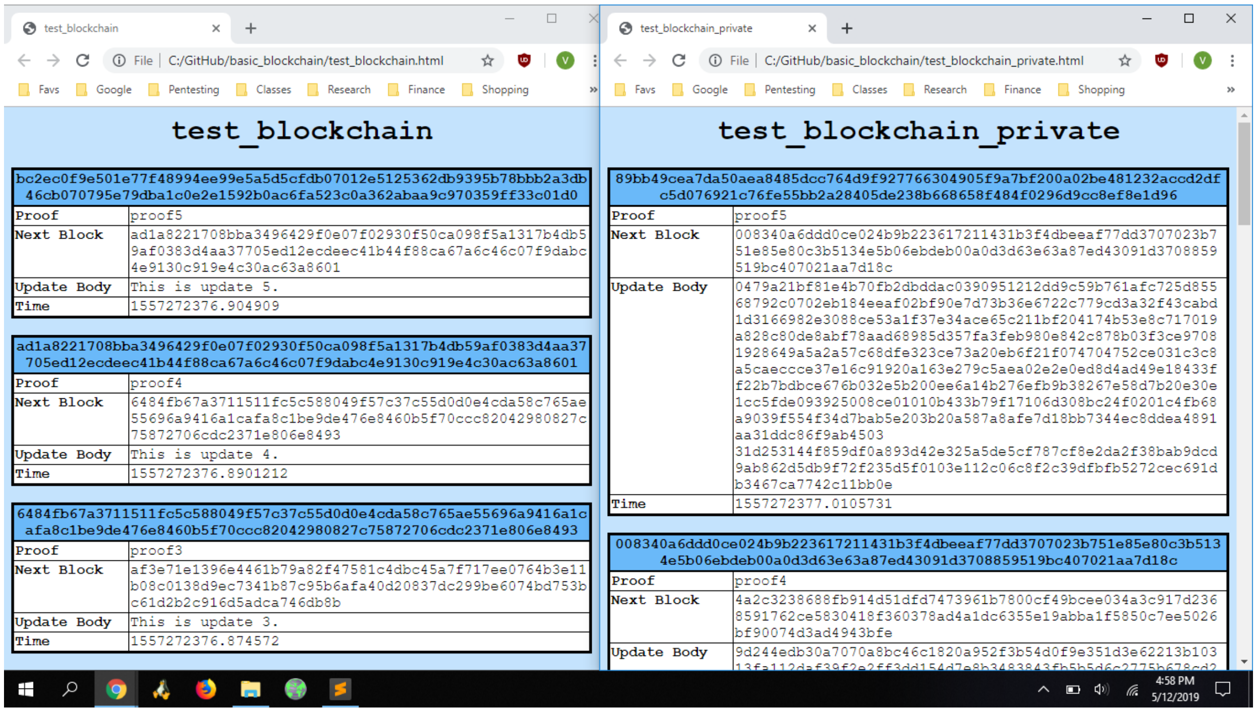


```
961b7800cf49bcee034a3c917d2368591762ce5830418f360378ad4
a1dc6355e19abba1f5850c7ee5026bf90074d3ad4943bfe

Successfully appended update 008340a6ddd0ce024b9b223617
211431b3f4dbeeaf77dd3707023b751e85e80c3b5134e5b06ebdeb0
0a0d3d63e63a87ed43091d3708859519bc407021aa7d18c

Successfully appended update 89bb49cea7da50aea8485dcc76
4d9f927766304905f9a7bf200a02be481232accd2dfc5d076921c76
fe55bb2a28405de238b668658f484f0296d9cc8ef8e1d96
```

Figure 12: The updates have been successfully appended to the blockchain.

After `blockchain_utils.py` [Appendix II] successfully appends all update blocks to the blockchain in the backend, `main_wrapper.py` [Appendix I] updates the web application displaying the blockchain [Figure 12].

The public version of the blockchain displays updates in plain text. This should not be used in reality and is meant to allow comparison only. The private version of the blockchain only displays the salted hash of the update. This ensures that proprietary code cannot be leaked or reverse-engineered, but still allows the code itself to be verified if necessary.


test_blockchain

bc2ec0f9e501e77f48994ee99e5a5d5cfdb07012e5125362db9395b78bbb2a3db46cb070795e79dba1c0e2e1592b0ac6fa523c0a362abaa9c970359ff33c01d0	
Proof	proof5
Next Block	ad1a8221708bba3496429f0e07f02930f50ca098f5a1317b4db59af0383d4aa37705ed12ecdeec41b44f88ca67a6c46c07f9dabc4e9130c919e4c30ac63a8601
Update Body	This is update 5.
Time	1557272376.904909

ad1a8221708bba3496429f0e07f02930f50ca098f5a1317b4db59af0383d4aa37705ed12ecdeec41b44f88ca67a6c46c07f9dabc4e9130c919e4c30ac63a8601	
Proof	proof4
Next Block	6484fb67a3711511fc5c588049f57c37c55d0d0e4cda58c765ae55696a9416a1cafa8c1be9de476e8460b5f70ccc82042980827c75872706cdc2371e806e8493
Update Body	This is update 4.
Time	1557272376.8901212

6484fb67a3711511fc5c588049f57c37c55d0d0e4cda58c765ae55696a9416a1cafa8c1be9de476e8460b5f70ccc82042980827c75872706cdc2371e806e8493	
Proof	proof3
Next Block	af3e71e1396e4461b79a82f47581c4dbc45a7f717ee0764b3e11b08c0138d9ec7341b87c95b6afa40d20837dc299be6074bd753bc61d2b2c916d5adca746db8b
Update Body	This is update 3.
Time	1557272376.874572

test_blockchain_private

89bb49cea7da50aea8485dcc764d9f927766304905f9a7bf200a02be481232accd2dfc5d076921c76fe55bb2a28405de238b668658f484f0296d9cc8ef8e1d96	
Proof	proof5
Next Block	008340a6ddd0ce024b9b223617211431b3f4dbeeaf77dd3707023b751e85e80c3b5134e5b06ebdeb00a0d3d63e63a87ed43091d3708859519bc407021aa7d18c
Update Body	0479a21bf81e4b70fb2dbddac0390951212dd9c59b761afc725d85568792c0702eb184eeaf02bf90e7d73b36e6722c779cd3a32f43cabd1d3166982e3088ce53a1f37e34ace65c211bf204174b53e8c717019a828c80de8abf78aad68985d357fa3feb980e842c878b03f3ce97081928649a5a2a57c68dfe323ce73a20eb6f21f074704752ce031c3c8a5caeccce37e16c91920a163e279c5aea02e2e0ed8d4ad49e18433ff22b7bdbce676b032e5b200ee6a14b276efb9b38267e58d7b20e30e1cc5fde093925008ce01010b433b79f17106d308bc24f0201c4fb68a9039f554f34d7bab5e203b20a587a8afe7d18bb7344ec8ddea4891aa31ddc86f9ab4503 31d253144f859df0a893d42e325a5de5cf787cf8e2da2f38bab9dcd9ab862d5db9f72f235d5f0103e112c06c8f2c39dfbfb5272cec691db3467ca7742c11bb0e
Time	1557272377.0105731

008340a6ddd0ce024b9b223617211431b3f4dbeeaf77dd3707023b751e85e80c3b5134e5b06ebdeb00a0d3d63e63a87ed43091d3708859519bc407021aa7d18c	
Proof	proof4
Next Block	4a2c3238688fb914d51dfd7473961b7800cf49bcee034a3c917d2368591762ce5830418f360378ad4a1dc6355e19abba1f5850c7ee5026bf90074d3ad4943bfe
Update Body	9d244edb30a7070a8bc46c1820a952f3b54d0f9e351d3e62213b103

Figure 11: The public and private version of the blockchain.

As seen in Figure 11 of the demo, the private version displays very differently from the public version. However, both blockchains were generated from the same base update files.

5 Conclusion

Our service provides a secure, user-friendly method for safely running and verifying code. It ensures that user data is being responsibly handled, while also enabling companies to provide technical proof of their actions should a need for legal cover arise.

This technical report was created by Crystal Yan (Sections 2 and 3) and Vy-An Phan (Sections 1, 4, 5, and 6). Crystal Yan was responsible for the overall architecture and graphics (originally included in the slide deck) of the project, while Vy-An Phan provided the code for the appendix.

References

[1] Facebook fares very poorly in this survey. *CB Insights*, 2018.

[2] What is psychographics? understanding the 'dark arts' of marketing that brought down cambridge analytica. *CB Insights*, 2018.

[3] J. Albright. The graph api: Key points in the facebook and cambridge analytica debacle. *Tow Center*, 2018.

[4] L. Beckett. Everything we know about what data brokers know about you. *ProPublica*, 2014.

[5] S. Fox, L. Rainie, J. Horrigan, A. Lenhart, T. Spooner, and C. Carter. Trust and privacy online: Why americans want to rewrite the rules. In *The Internet Life Report*. Pew Research Center, 2000.

[6] K. Granville. Facebook and cambridge analytica: What you need to know as fallout widens. *The New York Times*, 2018.

[7] C. J. Hoofnagle and J. King. Consumer information sharing: Where the sun still don't shine. *SSRN*, 2007.

[8] M. Kosinski and Y. Wang. Deep neural networks are more accurate than humans at detecting sexual orientation from facial images. *Journal of Personality and Social Psychology*, 114:246–257, 2018.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. pages 1097–1105, 2012.

[10] J. Podesta, P. Pritzker, E. J. Moniz, J. Holdren, and J. Zients. Big data: Seizing opportunities, preserving values. Executive Office of the President, 2014.

[11] E. Ramirez, J. Brill, M. K. Ohlhausen, J. D. Wright, and T. McSweeny. Data brokers: A call for transparency and accountability. Federal Trade Commission, 2014.

[12] J. Weiczner. How the insurer knows you just stocked up on ice cream and beer. *The Wall Street Journal*, 2013.

[13] Y. Wu, M. Kosinski, and D. Stillwell. Computer-based personality judgements are more accurate than those made by humans. *Proceedings of the National Academy of Sciences of the United States of America*, 112:1036–1040, 2015.

6 Appendix

All code, as well as the example results, can be found at: `https://github.com/vyanphan/basic_blockchain`

Appendix I: `main_wrapper.py`
Appendix II: `blockchain_utils.py`
Appendix III: `format.css`

Appendix I: main_wrapper.py

05/12/19 05:08:21 C:\GitHub\basic_blockchain\main_wrapper.py

```python
from html5lib import *
from blockchain_utils import *
import os

TABLE_LABELS = ['Proof', 'Next Block', 'Update Body', 'Time']
SEED_LENGTH = 256

# adds a new block to the webpage
def append_update(blockchain, proof, update_file, private_mode):
    with open(update_file, "r") as rd_file:
        update_block = rd_file.read()
        if private_mode:
            salt = os.urandom(SEED_LENGTH)
            update_block = salt.hex() + ' ' + HASH_FN(salt +
str.encode(update_block)).hexdigest()
        blockchain.append_block(proof, blockchain.tail, update_block)


# reads in information from an existing chain
def parse_chain(chain_folder, chain_head):
    if chain_folder[-1] != '/':
        chain_folder += '/'

    curr_block = chain_folder + chain_head

    with open(curr_block, "r") as ch_file:
        curr_block = ch_file.read()
        next_block = chain_folder + curr_block

    blocks = []

    while os.path.isfile(next_block):
        with open(next_block) as block_file:
            b1 = block_file.readline()
            b2 = block_file.readline()
            next_block = block_file.readline()[:-1]
            b3 = block_file.readline()
            b4 = block_file.readline()
        curr_block = (b1, b2, next_block + "\n", b3, b4)
        blocks += [curr_block]
        next_block = chain_folder + next_block
    return blocks

# uses existing blockchain files to generate the web page for that blockchain
def display(output_file, chain_folder, chain_head):
    blocks = parse_chain(chain_folder, chain_head)

    with open(output_file, 'w+') as out_file:
        out_file.write('<html><head><title>' + chain_folder + '</title>')
        out_file.write('<link rel="stylesheet" href="format.css"></head>')
        out_file.write('<body><h1 align="center">' + chain_folder + '</h1>')

        for b in blocks:
            out_file.write('<table>')
            out_file.write('<tr><th colspan=2>' + b[0] + '</th></tr> <col width="20%"/><col width="80%"/>')
            for i in range(0,4):
                out_file.write('<tr><td><b>' + TABLE_LABELS[i] + '</b></td><td>' + b[i+1] + '</td>')
            out_file.write('</table>')
            out_file.write('<br>')

        out_file.write('</body></html>')


# Example test code

# Generates the raw public version of the blockchain, with updates in plain text.
test_blockchain = Chain('test_blockchain', 512)
append_update(test_blockchain, 'proof1', 'test_blockchain_updates/update1', False)
append_update(test_blockchain, 'proof2', 'test_blockchain_updates/update2', False)
append_update(test_blockchain, 'proof3', 'test_blockchain_updates/update3', False)
append_update(test_blockchain, 'proof4', 'test_blockchain_updates/update4', False)
append_update(test_blockchain, 'proof5', 'test_blockchain_updates/update5', False)
display('test_blockchain.html', 'test_blockchain', 'test_blockchain')

# Generates the private version of the blockchain, with a salted and hashed version of the update, to protect proprietary code.
test_blockchain_private = Chain('test_blockchain_private', 512)
append_update(test_blockchain_private, 'proof1', 'test_blockchain_updates/update1', True)
append_update(test_blockchain_private, 'proof2', 'test_blockchain_updates/update2', True)
append_update(test_blockchain_private, 'proof3', 'test_blockchain_updates/update3', True)
append_update(test_blockchain_private, 'proof4', 'test_blockchain_updates/update4', True)
append_update(test_blockchain_private, 'proof5', 'test_blockchain_updates/update5', True)
display('test_blockchain_private.html', 'test_blockchain_private', 'test_blockchain_private')
```

Appendix II: blockchain_utils.py

05/12/19 05:04:07 C:\GitHub\basic_blockchain\blockchain_utils.py

```python
'''
A very, very naive blockchain.

Only used for demo purposes.

Not provably secure. Meant to show that it works in the context of this project.
'''



import hashlib
import os
from stat import S_IREAD, S_IRGRP, S_IROTH
import binascii
import time

HASH_FN = hashlib.sha512
HASH_LENGTH = 512


# a placeholder function that users can customize, if they want a different layout
def format_update(update_file):
    ''' Converts raw update file to a format for the blockchain. '''
    with open(update_file, "rb") as update_file:
        update_body = update_file.read()
    # do whatever formatting here
    return update_body # should be in byte format


# raised when a proof of work check fails
class ProofException(Exception):
    ''' Raised when proof of work fails to verify. '''
    pass


class Block():
    '''
    A single update unit to a blockchain. Block information is formatted in this order.

    self.hash   =   hash of current block (includes everything else)
    proof       =   proof of work
    prev        =   hash of previous block
    body        =   body of update for current block
    append_time =   timestamp

    Everything must be string format.
    '''
    def verify_proof(self, proof, prev, body):
        # return proof[:6] == '000000'
        return True

    def __init__(self, chain_name, proof, prev, body):
        # verify proof
        if self.verify_proof(proof, prev, body):
            # generate hash of block
            append_time = str(time.time())
            try:
                self.hash = HASH_FN(str.encode(proof) + str.encode(prev) + str.encode(body) + str.encode(append_time)).hexdigest()
            except:
                print("All arguments should be passed as string format.")

            # write to record file
            with open(chain_name + "/" + self.hash, "w+") as block_file:
                block_file.write(self.hash + '\n')
                block_file.write(proof + '\n')
                block_file.write(prev + '\n')
                block_file.write(body + '\n')
                block_file.write(append_time + '\n')

            # make file read-only
            os.chmod(chain_name + "/" + self.hash, S_IREAD)
        else:
            # do not create block if proof does not verify
            raise ProofException("Proof of work failed.")


class Chain():
    '''
    Made of many Blocks strung together.
    The first Block is always a bunch of random gibberish, i.e. a standard header.
    This is to provide randomness and security to the rest of the blockchain,
    and to prevent null pointers.

    chain_header stores the tail, aka the hash of the most recently appended block.

    self.name = name of the chain
    self.tail = hash of most recently appended block
        This is just meant to make appending easier for the sake of testing and demonstration.
        This tail file is not meant to be a secure display of the latest appended block. Unlike the record files it is not write protected.
    '''

    # reads from existing chain if available, or creates a new one
    def __init__(self, chain_name, seed_length):
        self.name = chain_name

        if os.path.isfile(chain_name): # existing chain already exists
            print("Loading blockchain '" + chain_name + "'.\n")
            with open(chain_name + '/' + chain_name, "r") as chain_header:
                self.tail = chain_header.readline()

        else: # initialize new blockchain
            print("Blockchain '" + chain_name + "'' does not exist. Initializing new chain.\n")
            os.mkdir(chain_name)

            # create new root block
            with open(chain_name + '/' + chain_name, 'w+') as chain_header:
                proof = os.urandom(seed_length).hex()
                prev = os.urandom(HASH_LENGTH).hex()
                body = os.urandom(seed_length).hex()

                root_block = Block(self.name, proof, prev, body)
                chain_header.write(root_block.hash)

            # initialize blockchain t ail
            with open(chain_name + '/' + chain_name, "r") as chain_header:
                self.tail = chain_header.readline()


    # adds a new block onto the chain
    def append_block(self, proof, prev, body):
        try: # attempt to create new block
            new_block = Block(self.name, proof, prev, body)

            # update header file and tail
            with open(self.name + '/' + self.name, 'w') as chain_header:
                chain_header.write(new_block.hash)
                self.tail = new_block.hash
            print("Successfully appended update " + new_block.hash + "\n")

        except ProofException:
            # do not create new block record or update tail if proof fails
            print("Proof of work failed. Block not appended.")




# test_chain = Chain('newchain_1', 720)
# test_chain.append_block('proof1', test_chain.tail, 'update1')
# test_chain.append_block('proof2', test_chain.tail, 'update2')
# test_chain.append_block('proof3', test_chain.tail, 'update3')
# test_chain.append_block('proof4', test_chain.tail, 'update4')
# test_chain.append_block('proof5', test_chain.tail, 'update5')
```

Appendix III: `format.css`

```
05/12/19 05:09:10 C:\GitHub\basic_blockchain\format.css
```

```
body {
  background-color: #c4e3fd;
  font-family: "Monospace", Courier;
}
table {
  table-layout: fixed;
  width: 100%;
  background-color: #FFFFFF;
  border: 3px solid black;
  border-collapse: collapse;
}
th {
  word-wrap: break-word;
  background-color: #6ab9f9;
  border: 2px solid black;
  border-collapse: collapse;
}
tr {
  word-wrap: break-word;
}
td {
  border: 1px solid black;
  border-collapse: collapse;
  vertical-align: top;
}
td+td {
  width: auto;
}
```