Myles Hobbs
Ethan Luong

469 Project Report

Project Requirements

For this project our group chose track 2. We did this because we had never created something that interacts with blockchain or how it could run on Ethereum. These were both concepts we were familiar with but didn't know that they could have been used in programming. We thought this would be a balance of something challenging but also interesting for us. The project for Track 2 involves creating a digital solution using open-source blockchain frameworks to maintain a secure and immutable chain of custody. This is particularly significant in digital forensics, where ensuring the integrity and authenticity of evidence is crucial. The project challenges us to design and implement a system that records all interactions with evidence, such as adding, checking in and out, and removing items from a case. The records must be stored transparently and securely on a blockchain, leveraging its cryptographic properties to prevent tampering.

Unlike Track 1, which focused on developing solutions in a controlled environment, Track 2 challenged us to work with open-source frameworks and real-world blockchain tools. This required not only implementing the core requirements outlined in the project but also adapting these to the strengths and limitations of blockchain platforms like Ethereum. Through this process, we learned to balance the challenges of secure data storage, immutability, and cryptographic functionality, reinforcing the relevance of blockchain in modern digital forensics. The added complexity of building a solution offered us invaluable insights into the practical considerations of using blockchain in evidence management.

To accomplish this, we are tasked with using frameworks like Hyperledger or Ethereum. These platforms will allow us to efficiently manage the creation, verification, and storage of blockchain blocks, ensuring each action is traceable and secure. The project also emphasizes understanding the practicality of blockchain in real-world applications, encouraging us to evaluate its appropriateness for a chain of custody solution and explore its strengths and limitations. This hands-on experience provides valuable insight into blockchain technology's role in improving the

reliability of digital evidence handling systems while honing our technical and critical thinking skills.

Additionally we are tasked fulfilling the common requirements of the document like defining the block structure, where each block must include specific fields like a 32-byte previous hash, an 8-byte timestamp, 32-byte case and evidence IDs, a 12-byte state, and additional details like creator and owner information, all adhering to strict byte alignment rules. To meet these requirements, the project challenges us to implement these blocks in hashed format, ensuring every field is properly packed, stored, and retrievable.
One of the critical aspects involves integrating AES encryption for the case and evidence IDs. This means that while storing the IDs, they must be securely encrypted using the provided AES key in ECB mode, ensuring confidentiality. Furthermore, the system must allow decryption of these IDs only when a valid password, from the predefined set in the document, is provided during operations like viewing cases or items. This adds an extra layer of security and ensures that unauthorized users cannot access sensitive information.

Additionally, we are required to implement the common commands, such as adding evidence, checking in and out items, and verifying the blockchain's integrity. These functions need to enforce access control, meaning that actions like "add," "remove," or "check history" can only be executed by users with valid credentials. To make this work, the program must handle authentication seamlessly and ensure that decrypted IDs are only displayed when the correct password is supplied. These detailed requirements push us to think critically about secure data handling and access control while honing our technical skills in cryptography and blockchain development.

Design Decisions Made

Implementing a blockchain application from scratch initially felt overwhelming, but the provided video guide was an invaluable resource, walking us through key steps and configurations. Fortunately, the frontend development was not a requirement for this project, allowing us to focus entirely on the back-end and blockchain functionality. The project required leveraging an open-source blockchain framework, and starting with Ganache proved to be a practical approach.

Ganache allowed us to simulate a local Ethereum blockchain environment, making it easier to develop and test without interacting with a live network or incurring real transaction costs.

Ganache and Truffle were chosen for their robust feature sets and ease of use in blockchain development. Truffle's extensive documentation and built-in testing tools provided a strong foundation for deploying and managing smart contracts, while Ganache's intuitive UI and ability to simulate a local Ethereum blockchain enabled rapid prototyping and debugging. We considered alternative tools like Hardhat but found that Truffle's ecosystem was more suited to our needs, particularly for managing migrations and testing. While omitting the frontend limited the usability of our solution, it allowed us to focus resources on optimizing back-end processes and meeting the project's core requirements. These design choices ensured a streamlined development process and laid the groundwork for potential future scalability.

Solidity was surprisingly intuitive to work with, thanks to its heavy influence from C++, which made it feel familiar and straightforward to learn. Within our Solidity contract, I was able to define all the necessary functions to interact seamlessly with the Ganache blockchain. One key implementation was a blocks mapping, which allowed me to associate indexes with blocks in the blockchain. When the add function was called with the appropriate parameters, it added the items to the blocks mapping, and their respective hashes were stored on the blockchain. Each transaction created a new block in Ganache, providing a clear and practical representation of how blockchain states evolve. For instance, adding three new item IDs with a single call to add resulted in one hashed block on the Ganache blockchain while locally storing each item as a unique hash. This uniqueness was ensured by using keccak256(abi.encodePacked()) to hash each item's parameters into a bytes32 format. This same hashing method was applied to password handling, maintaining security and consistency.

To manage errors and enforce rules, I used Solidity’s require() function, which proved invaluable for validating inputs and triggering error responses. For example, when calling remove, the function set isRemoved to true for the specified ID, ensuring subsequent operations like add or checkout on the same ID were disallowed. This design made error handling systematic and

straightforward. However, a limitation was that while interacting with the blockchain updated Ganache's state, it didn't always synchronize perfectly with the local blocks mapping, creating occasional discrepancies.

On the JavaScript side, I developed a robust input parsing system to handle user commands similar to the project documentation (e.g., -c, -i, -g, -p). By indexing the input arguments, I validated them and threw errors for incorrect or missing inputs. JavaScript was also used to create an instance of the deployed Solidity contract, using truffle migrate to deploy the contract and then passing the parsed arguments as parameters to the Solidity functions. Errors thrown by Solidity's require() were captured and logged via the JavaScript console, providing clear feedback to the user. All of the show functions were implemented in JavaScript too due to challenges between the hashed output of Solidity.

Overall, this setup allowed for a smooth integration of Solidity and JavaScript, with Ganache serving as a reliable testing environment. While this provided a functional scope for the project, further implementation details, especially related to challenges, are discussed in the challenges section.

Challenges

Implementing this solution was far from straightforward, requiring significant effort to set up and initialize the local environment before I could even start coding. One of the biggest challenges was establishing seamless communication between the Solidity contract, the Ganache blockchain, and the Truffle backend with the JavaScript frontend. Beyond that, encrypting and securely displaying output took the most time and effort. The project requirements were primarily tailored to Track 1, so adapting them intuitively for Track 2 required creativity and problem-solving.

Although debugging code itself wasn't an issue—thanks to prior experience—there were unique challenges specific to this project. Surprisingly, writing the Solidity contract turned out to be one

of the easier tasks. However, implementing AES-ECB encryption and decryption for user inputs, such as caseID and itemID, proved to be much more complex. Since Solidity does not support AES-ECB natively, I decided to handle encryption and decryption in JavaScript. To facilitate this, I created an encryptionUtil.js file, which utilized a crypto library installed via npm. While the encryption and decryption themselves were straightforward, feeding the encrypted data back into Solidity raised numerous issues.

The main problem stemmed from the type mismatch between JavaScript and Solidity. CaseID and itemID were strings of varying lengths, but Solidity required them to be exactly bytes32. This necessitated extensive padding in JavaScript to match the required length, which was both time-consuming and error-prone. Furthermore, the encryption process produced base64-encoded outputs, which had to be converted into bytes32 for Solidity. This added another layer of complexity and debugging. These type-related issues were compounded when returning data from Solidity or Ganache, where handling outputs consistently became another hurdle.

One of the biggest challenges I faced was dealing with the complexities of encryption, hashing, and byte conversions when trying to retrieve data from Solidity. Since much of the data was stored as bytes, attempting to access it via the Truffle console often returned undefined, adding layers of frustration. This issue, compounded by the encrypted format of caseIDs and itemIDs, required a significant amount of conversion just to retrieve and display the data properly. To address this, I decided to store encrypted IDs, statuses, and timestamps locally, which simplified the retrieval process. I implemented a caseTracker.js utility to manage these locally stored values, primarily for display purposes in the show functions. For functions that didn’t require a password, I displayed the encrypted values directly, whereas for password-protected functions, I decrypted the data to reveal the actual values. This approach allowed me to navigate the constraints of Solidity's limited handling of complex data types more effectively.

Another significant hurdle was demonstrating test cases for the project. Unlike Track 1, there were no predefined test cases for Track 2, so error handling became critical to proving the robustness of my implementation. However, Truffle tests posed a unique challenge: they operate by spawning a temporary Ethereum network instance (Ganache) specifically for testing. While

this isolated environment is great for ensuring consistency, it proved ineffective for a demonstration where context and continuity between tests were essential. For example, a test for add followed by tests for checkIn or checkOut would not carry over state, as each test ran in a vacuum. This lack of context made it difficult to showcase how errors or state changes would naturally occur.

To overcome this, I prioritized creating a demonstration framework that allowed users to execute tests locally, showcasing the real-time behavior of the system. This approach not only demonstrated the system's error-handling capabilities but also avoided repetitive and redundant code that would have been required to recreate context across isolated Truffle test cases. By focusing on an interactive demonstration, I was able to effectively convey the functionality and reliability of the implementation in a more intuitive and practical manner.

Despite these challenges, I was able to implement a working solution by refining the interaction between the encryption utility, JavaScript, and Solidity. Each step of the process required careful handling of data types and precise communication between components, highlighting the nuanced difficulties of bridging multiple technologies in a blockchain environment.

Why NOT Blockchain

In the context of this project, where a chain of custody is meant to ensure the immutability of evidence, using blockchain, particularly in the way Ganache and Solidity function, might not have been the most suitable implementation. One significant limitation is that Ganache creates a new block for every single interaction with the Solidity contract, rather than for every logical block representing evidence. To achieve a reflection of the intended number of blocks in the blockchain, I had to iteratively call functions, which was inefficient and cumbersome. Moreover, despite the supposed immutability of blockchain, I discovered that it was possible to go back and edit existing blocks in my personal blockchain, as reflected in the Truffle console. For instance, the checkIn and checkOut statuses of an itemID required updates to indicate whether an item was checked in or not, and these updates were hashed and reflected in the blockchain. This

fundamentally undermines the concept of immutability, as both Ganache and Solidity allowed mutable changes to the blockchain's internal state.

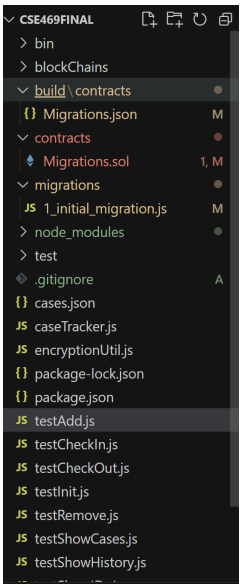
Another drawback of using blockchain in this context lies in the reliance on Truffle's migration process. Deploying the Solidity contract with truffle migrate is mandatory, but it creates two unnecessary blocks in the blockchain, which adds unnecessary clutter. This further highlights how the number of blocks created in Ganache does not align with the logical structure of blocks in the actual blockchain. Ganache hashes every function call that modifies the contract, creating new blocks even if a function reverts or doesn't introduce meaningful changes. This behavior not only misrepresents the actual chain but also adds complexity and inefficiency, making it less than ideal for a chain of custody solution.

Lastly, another issue with this approach is that Ganache's blockchain is local and temporary. For a chain of custody solution, which requires permanent and immutable records, Ganache's quick-start functionality falls short. While it is convenient for testing and user experience, it allows the blockchain to be easily reset or discarded, negating the permanence required for tracking evidence. Though Ganache offers workspace-saving features, they are not comprehensive, as it does not automatically save every quick-started workspace. This transitory nature of Ganache makes it unsuitable for a robust chain of custody solution, as it contradicts the very principles of immutability and reliability necessary for evidence tracking.

While blockchain offers unparalleled immutability and transparency, alternative solutions might be more practical for a chain of custody system. For example, traditional databases combined with cryptographic hashing could achieve similar security and traceability without the overhead of blockchain's consensus mechanisms. A distributed ledger technology (DLT) like Hyperledger Fabric could also provide a more controlled environment, allowing for efficient evidence tracking without the inefficiencies observed in Ganache. These alternatives would eliminate issues like unnecessary block creation and mutable states while still preserving the key features of a reliable chain of custody solution. Evaluating these options highlights how blockchain, while innovative, might not always be the most efficient or appropriate choice.

Running & Testing

1. npm install -g npm
2. Download and Install Node.js
    a. node -v
    b. npm -v
3. Download & Install Ganache
4. Install Truffle Framework
    a. npm install -g truffle
    b. To verify run
        i. truffle version
5. Make sure the versions align with:
    a. Truffle v5.0.2 (core: 5.0.2)
    b. Solidity v0.5.0 (solc-js)
    c. Node v8.9.0
6. Set Up Project Directory
    a. mkdir blockchain-chain-of-custody
    b. cd blockchain-chain-of-custody
    c. Could be any folder name
7. truffle init
    a. This will help create the project set up
8. Download Contract & JS Files from Repo
    a. Github
9. Configure your directory to reflect if you don't want to change file locations in the actual code
    a. →
10. Configure Truffle for Ganache
    a. Ensure your truffle-config.js & initial_migration align with your local set up
11. truffle compile –all
    a. Running npm install may be helpful to install necessary libraries



12. truffle migrate –reset
    a. truffle console will now allow you to interact with your contract
    b. This console is Javascript
13. Run the outline test using:
    a. truffle exec test___.js *valid arguments*

Storage

In terms of storage, I implemented two main strategies to manage and retrieve data effectively. When calling Solidity functions that modified the state of my contract, I utilized the blocks mapping within Solidity to store all block data, including hashed IDs and metadata. This allowed me to display the stored data in the Truffle console at any time. The hash address for each transaction was also saved in Ganache, making it easy to reference and trace these calls within the Ganache UI, including identifying the contract associated with each block. While this method was effective for storing block data, it introduced challenges in retrieving information due to the way Solidity handles data access.

To address this, I created additional utilities, including caseTracker and a cases.json file. These files were specifically designed to enhance the usability of the show functions by locally storing relevant block data corresponding to the data stored in Solidity. For instance, when calling a function like add or checkIn, a new block would not only be created in the Solidity blocks mapping but also logged in cases.json. This dual-layered approach ensured consistency between the blockchain data and what was accessible through the JavaScript interface.
The system also implemented strong access control to protect the stored data. To call a show function and retrieve sensitive information, users needed to input the correct password, which was hashed on the backend using Solidity. This ensured that passwords could not be reverse-engineered or retrieved from the JavaScript side, as the hashing process happened securely within the smart contract. If an invalid password was provided, the system would only display encrypted IDs, making it nearly impossible for unauthorized users to interpret or misuse the data.

This combination of Solidity storage for on-chain blocks, local JSON storage for display purposes, and password-protected access provided a robust solution for managing and retrieving

transaction data securely. The source code implementing this storage and access logic can be found in the linked GitHub repository.

GitHub + Video

- Github Repo
- Video
    - Zoom
        - Password: jV&L.s8%
    - Youtube

References

- *Text generated by ChatGPT, OpenAI, December 5, 2024, https://chat.openai.com/chat.*
- ChatGPT, response to “Explain why Hyperledger may be more appropriate for a chain of custody” OpenAI, *December 5, 2024*.
- ChatGPT, response to “Please expand on my report” OpenAI, *December 5, 2024*.