Secure smart contract attestation using Intel SGX

Agniv Chatterjee
Supervisors: Prof. Dr. Kaitai Liang, Huanhuan Chen
TU Delft, The Netherlands

Abstract

Blockchain networks are increasingly recognized as a disruptive technology across sectors such as online services, finance, supply chain etc. They are underpinned by smart contracts which provide programmatic instruction for the blockchain to operate. A major obstacle in the widespread adoption of blockchain technology is the security of the underlying smart contracts and potentially exploitative flaws in their technical makeup that pose a risk to data privacy. Modern trusted execution environments, such as Intel SGX, leverage hardware and have been proposed to preserve privacy in smart contracts; however, practical research & development in this field has seen slower progress. This paper explores the process of attestation by which Intel SGX enhances smart contract security, examines development & execution of a prototype smart contract that utilizes SGX for secure e-voting and evaluates benefits & limitations of the process. Finally, we also propose improvements to our approach and present further scope of research on the topic.

1 Introduction

Blockchain technology is a fast-growing market and has received a lot of attention in recent times due to popular use-cases such as cryptocurrencies, non-fungible tokens, cross-border payments and other decentralized applications. Blockchain technology is underpinned by smart contracts, which are programs containing a set of actions to be executed when certain conditions are met and verified [10]. These actions update the blockchain and are visible only to the parties involved in the smart contract. This makes smart contracts useful for a variety of applications such as registration of assets in one's name, transfer of funds between parties, notarizing of claims for insurance purposes, automation in supply chain processes, liability management by firms etc. [10]

Given the monetary nature of many of these popular cases, it is essential that smart contracts are made secure, preserve privacy and that data contained in these programs are not easily accessible to external parties. To some extent, smart contracts benefit from the inherent encryption of blockchain networks, where data is structured into a 'chain' of blocks; these blocks are all inter-linked and attackers cannot access any single block without changing all the blocks connected to it as well.

However, smart contracts are not immune to having vulnerabilities and security issues which can result in network compromise and massive financial losses [17]. A popular example of this was the DAO hack in 2016, where attackers exploited a vulnerability in the DAO's codebase to siphon funds of ether worth $60 million following a massive token sale [17].

The vulnerabilities smart contracts are subject to mainly revolve around the transparency of data in the distributed ledger across all nodes of the network [7]. While this is a core property of the decentralized manner in which blockchains operate, it also introduces a lack of privacy which is increasingly necessary as enterprises look to leverage these systems to handle user data and accommodate customer requirements across various sectors [9].

As a potential solution to the issue of data security, hardware-based trusted execution environments (TEEs) can be used to run smart contracts in an isolated, secure container such that the data inside is kept hidden from potential attackers [5]. This paper focuses on Intel's Software Guard Extensions (SGX), which is a prominent TEE solution and offers many advantages with regards to the privacy, integrity and scalability of smart contracts.

Related studies Relevant research in this field has been conducted previously by Brandenburger et. al [7] [14], Bao et. al [5], Das et. al [9], Liang et. al [12] and others analyzing the integration of Intel SGX with smart contracts. Brandenburger et. al provide a well-detailed overview of the attestation process with SGX and identify pitfalls which can arise. Das et. al conduct similar research but highlight an extra security layer atop their design to achieve objectives of security, privacy and scalability of data. Abdul-Kader & Kumar [3] propose a three-layered protection scheme to preserve privacy in large-scale blockchain platforms by introducing randomized address generation, content erasure and Intel SGX as trusted hardware.

However, for much of the literature, with the notable exception of Brandenburger et. al [7], there are few attempts to back research with firsthand development of a blockchain prototype that leverages Intel SGX. Thus, it remains to be seen to what extent this technology can be applied to smart contracts in practice and how well they actually solve the issue of privacy.

Contribution The aim of the project is to address this and provide an accurate account of the development process involved in implementing an SGX-based smart contract. The paper explores the means by which Intel SGX enhances smart contract security, accompanied by the implementation of an e-voting scheme, and examines the benefits of this process as well as limitations with regards to security risks and potential vulnerabilities that can be exploited.

The paper focuses on smart contracts developed on the Hyperledger Fabric platform, which is a prominent enterprise-grade framework for deploying blockchains requiring permissioned access [7]. Unlike Bitcoin and Ethereum, these have owners that control who can see and commit to the network; this makes them more secure and compatible to enterprise use [9]. Smart contracts on Hyperledger Fabric can be developed using general-purpose languages [17], such as Go, Node.js, Python and Java, and this also reduces the learning curve for developers.

The main research question the paper is looking to answer is the following:

How can Intel SGX be used to enhance the security of smart contracts on Hyperledger Fabric?

The research this paper underlines can be broken up into the following sub-questions:

Q1: *How to apply Intel SGX technology to the execution of a Fabric smart contract?*

Q2: *What is the present literature on the security of Fabric smart contracts?*

Q3: *How effective is SGX as a trusted execution environment solution for smart contracts?*

Structure. The paper is structured into eight sections. Section 2 provides relevant background information on Hyperledger Fabric and Intel SGX technologies used in the prototype. Section 3 covers the methodology and approach followed for the project. Section 4 introduces the design, implementation and trial of the e-voting prototype. Section 5 provides an evaluation of the prototype, examines the benefits & limitations of the process and outlines potential improvements for the future. Section 6 examines the ethical implications of our study and reproducibility of the results. Section 7 incites discussion by means of a literature review of related works in the field. Section 8 summarizes the research and concludes the paper.

2 Background Research

This section presents background information required to grasp the technical aspects discussed later in the paper. Section 2.1 provides an overview of the architecture used by Hyperledger Fabric while Section 2.2 introduces Intel SGX and the process of attestation by which it enhances smart contract security.

2.1 Hyperledger Fabric

In Hyperledger Fabric, multiple parties collectively form a distributed ledger network, or DLT [7]. The ledger maintains a record of all interactions between the parties as *transactions* stored in a chain of ordered blocks. Each block has a unique hash identifier derived from the hash of its predecessor [1]; this contributes towards the encryption of blockchains, as mentioned in the previous section.

Transactions invoke functions of the smart contracts, or *chaincode*, underpinning applications running on the blockchain. Additionally, the *state* of the blockchain denotes the latest information stored on the ledger for quick retrieval; this is often in the form of a key-value store [7] which can be accessed or written to with *getState* and *putState* calls.

A Fabric network consists of three main types of entities: clients, peers and an ordering service [17]. The client invokes chaincode operations on behalf of the end-user through transaction requests to one or more peers. The peers execute the chaincode and produce a response back to the client, known as an *endorsement* [2].

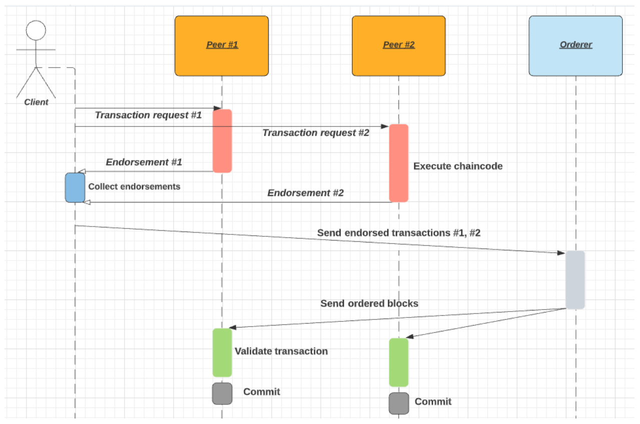


Figure 1: Sequence diagram representing the transaction flow in our Fabric test network.

The ordering service then establishes the order of the endorsed transactions it receives from the client, assigns them into blocks and communicates the blocks to all peers in the network. Upon receiving a block, the peer then validates the transactions inside and commits them to their own local copy of the distributed ledger [7]. Thus, in Fabric, it can be noted that the transactions are ordered *after* they have been executed & endorsed by peers hence the execute-order-validate transaction flow of Fabric networks.

Security Hyperledger Fabric provides data confidentiality to a certain extent, making use of in-built tools such as *channels* and *private data* storage [16].

Channels split the network into smaller networks, each containing a local ledger and 2 or more peers where transactions are hidden to entities outside the channel [16]. Private data storage allow peers to store data such that it is only accessible to specific peers. With these tools, data is kept hashed [16]) to all except the peers authorized.

Threats Given the execute-order-validate flow outlined above, a peer is also technically able to execute a transaction multiple times while the ordering service decides the order of transactions [7]. A malicious peer could take advantage of this flaw in order to learn sensitive information about the state. This is known as a *rollback* attack where the attacker can break confidentiality despite being not being authorized to see the data.

Kazuhiro et. al [17] also attribute non-determinism as posing a large risk to smart contracts on Hyperledger Fabric. This is namely, when there is inconsistency among the responses received from the endorsing peers in the network. Non-determinism can stem from various factors such as exploitation of global variables, transaction timestamps, third-party libraries, external services etc. [17]

2.2 Intel SGX

Intel's Software Guard Extensions (SGX) is a prominent TEE solution that leverages hardware [5] to provide security. Since blockchain networks contain a distributed ledger that maintains a record of all transactions validated, this data is visible to all nodes in the network [14]. This highlights a lack of confidentiality in the blockchain state and does not uphold privacy for users who may have sensitive personal data on the blockchain, such as health-related information, voting preferences, balance details etc.

Intel SGX provides a secure and isolated container, known as an enclave [5], which is guarded by hardware mechanisms and ensures there is confidentiality of data in the application, even in the event the entire platform is compromised [7]. Intel SGX provides functionality for remote attestation, intra-attestation as well as enclave state and data sealing [5] processes.

Remote attestation. When starting up an enclave, the data and code loaded into the SGX container is used by the CPU to generate a cryptographic hash, known as the mrenclave. Before the client can send transactions to the peer, this hash value is verified to ensure that the correct chaincode is being run by the enclave [5].

At runtime, the client issues an attestation challenge to the peer running the enclave, to generate a *quote* proving that the correct chaincode is loaded in the enclave [5]. To do this, the enclave first conducts intra-attestation with an entity known as the Quoting Enclave (QE). The QE verifies the quote was produced by a valid enclave and signs it using a shared identity key [12]. The quote is then sent to the client who forwards it to the Intel Attestation Service (IAS). Once this is also verified, the client can confirm the chaincode is running the appropriate chaincode and can communicate [7].

Data sealing. SGX provides support for data sealing of an enclave, by encrypting the data with a unique key [5] before it is stored externally in persistent storage. This helps the enclave reduce memory consumption [5] and in the event of a crash, the stored data can be retrieved and decrypted easily [7]. However, the main limitation of this is that, while storing and retrieving data may be secure, the data itself can be inferred [7] by malicious nodes looking to break confidentiality of the system.

3 Methodology

The approach undertaken in this project consists of three stages: background research & study, prototype development and evaluation. A technical roadmap outlining the steps taken over the course of the project has been included and can be found in Appendix A in section 9.

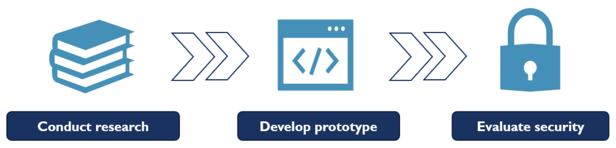


Figure 2: Diagram depicting the 3-step approach taken towards the project

3.1 Research Process

To begin, it is necessary to develop a fundamental understanding of how a Fabric network operates, how transactions are processed by the entities in the network, potential risks of Fabric smart contracts, SGX process of remote attestation and the security it can provide to Fabric smart contracts. This was predominantly done by process of literature study and analyzing present approaches to evaluate Intel SGX as a trusted execution environment.

The process of finding literature on this topic involved use of Google Scholar as it features a wide range of scientific research that can prove to be useful in acquiring background information. Keywords used include: *Smart contracts*, *blockchain*, *Hyperledger*, *Intel SGX*, *e-voting* etc. In order to ensure sufficient literature was found on this topic, a time window from 2017 to 2021 was chosen for articles; this was at the risk of some information being possibly outdated in the older articles. Thus, those findings were cross-checked and used for more trivial purposes, such as definitions, background information etc.

3.2 Development Process

The research is supported by the development of an e-voting smart contract on Hyperledger Fabric. This was done in a process similar to software engineering with Scrum, where the prototype was iteratively implemented in week-on-week sprints and in close coordination with the responsible professor.

To start, there was a process of ideation regarding the smart contract and the chaincode functions to implement. The contract allows an organizer (the client) to open elections with a total of 3 candidates, for whom votes can be submitted by peers in the network. Then, a backlog was produced containing issues of all the features the smart contract would have. The implementation of the election was carried out in a timeboxed manner similar to weekly sprints, accompanied by meetings with the project group and responsible professor. The comments and feedback provided were used to add new issues to the backlog that would be worked on in the coming week.

The smart contract is implemented using C++ and Golang, while the client application used to invoke the contract is written in Javascript. This choice of programming languages can be attributed to a combination of prior coding experience, support provided by Hyperledger and SGX SDKs and existing resources that can be found on smart contract development.

4 Prototype

This section details the e-voting prototype built using Hyperledger Fabric and Intel SGX technologies. Section 4.1 details the design of the system, while Section 4.2 covers the implementation of the smart contract & integration with SGX. Section 4.3 trials the prototype and provides the results of execution.

4.1 System Design

Our approach makes use of a local Fabric test network consisting of a client node, two peers and an ordering service. Each node belongs to a Membership Service Provider (MSP) organization, which exists in Hyperledger Fabric to manage identities of the members in the network [17]. In our application, the peers represent voters that submit votes for the candidates in our election while the organizer of the election denotes the invoking client.

In order to incorporate trusted hardware, each peer in the Fabric network is equipped with a CPU that enables Intel SGX and can execute chaincode within a secure enclave. As opposed to running the entire blockchain node in SGX, only part of the peer resides in the enclave [7]; this minimizes the size of the trusted computing base (TCB), the attack surface and makes the system easier to evaluate as well.

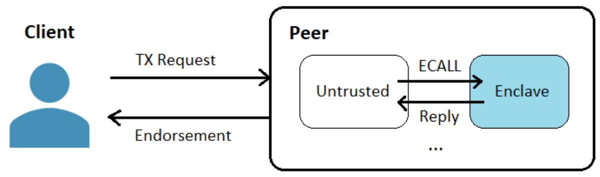


Figure 3: Diagram of the peer representing the components discussed above.

A peer in our network contains the following components:

- *Secure enclave*: the enclave hosted by the peer executes the e-voting chaincode, including key operations such as submitting encrypted votes, querying the votes, displaying them and evaluating the winner candidate. This isolates the core business logic from the rest of the peer and preserves privacy.
- *Untrusted component*: this is the remaining (untrusted) section of the peer containing code that runs outside SGX. This contributes largely in the remote attestation process and forwards requests from the client to the chaincode enclave via ECALL messages [9].
- *Other parts*: the peer also has several other internal components such as its own copy of the distributed ledger, the most-recent blockchain state, a registry with all the enclave identifiers and a transaction validator to validate transactions received from the ordering service [7].

Now, we discuss at length the transaction flow of the Fabric network and the role of cryptographic algorithms such as key generation, encryption and digital signatures to establish secure data sharing between parties on the smart contract.

To begin, the peer starts up the SGX enclave and generates a private-public key pair within the container [9]. Then, it registers the enclave with the client & Intel Attestation Service (IAS) through means of remote attestation (as explained in Section 2.2). To briefly summarize, the chaincode enclave produces a quote containing the mrenclave and the hashed public key of the enclave. This is sent by the client to the IAS for verification; if successfully verified, the client communicates this to the peer, which then stores the successful result along with the public key of its enclave to the distributed ledger. This is available to all nodes on the network [7].

Following enclave registration, the client can securely communicate with the peer and invoke chaincode operations through transaction requests. Within a request, the client encrypts the desired operation payload using a randomly-generated composite key and the public key of the enclave that it received earlier [9].

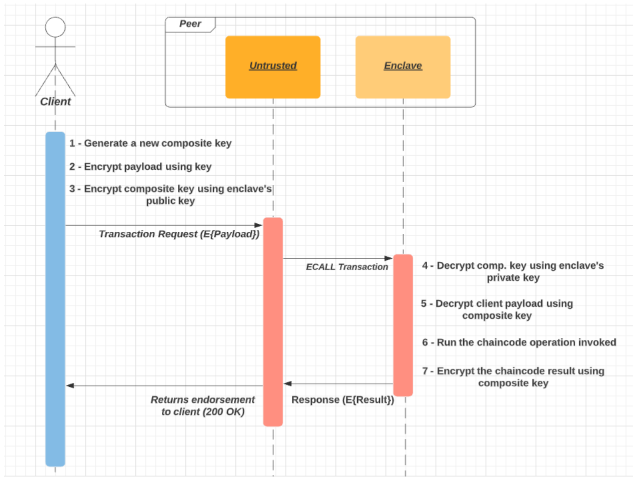

Client
Peer
Untrusted
Enclave
1 - Generate a new composite key
2 - Encrypt payload using key
3 - Encrypt composite key using enclave's public key
Transaction Request (E{Payload})
ECALL Transaction
4 - Decrypt comp. key using enclave's private key
5 - Decrypt client payload using composite key
6 - Run the chaincode operation invoked
7 - Encrypt the chaincode result using composite key
Response (E{Result})
Returns endorsement to client (200 OK)

Figure 4: Secure communication between the client and peer running SGX

The peer takes the request into its enclave and decrypts the composite key using the private key of the enclave [9]. The chaincode operation is fulfilled inside the enclave and the result is encrypted using the composite key [9]. The peer forwards this as an endorsement to the client. Finally, the client decrypts the execution result present in the endorsement and submits the transaction to the ordering service.

Upon receiving a block from the orderer later, the peer validates the transactions within [7] before committing them to ledger.

4.2 Implementation

This section outlines the implementation of the e-voting smart contract and details the method used to incorporate Intel SGX for the prototype. While the full code and setup procedure can be found on Github [8], this section provides some background and can prove useful for reproducibility of the results outlined below.

Smart contract The e-voting scheme has 7 main functions, which allow the user to create an election, query it, submit encrypted votes, close the election, make votes visible and ultimately determine the candidate with most votes.

The functions are all called by instances of the smart contract in classes of the client application. The user is able to invoke these classes through the CLI approach. The class diagram detailing the implementation design can be found in Appendix B below.

Intel SGX Fabric Private Chaincode, or FPC, is a framework by Brandenburger et. al [6] that allows for the execution of chaincode using Intel SGX. This framework was used in order to run the e-voting smart contract within an enclave and provides support for various components such as the chaincode enclave, enclave endorsement validation, shim and enclave registry [6].

The advantage of using FPC is that it packages all the required dependencies within a Docker development image, thereby improving the learning curve and also allowing the developer to focus principally on building the smart contract.

4.3 Execution

This section trials the smart contract on mock inputs and provides the results of the execution in listings. The application does not use personal data and the inputs entered bear no resemblance to any real-world entities.

The organizer can start a new election by invoking the *CreateElection* function in our smart contract, with the election name and the three candidates to vote for as function parameters. The chaincode then initializes a new election with the election name, candidates, ID of the organizer and placeholders to store the votes. Once the election is created, it is put into state using *putState* and the status of the election is set to *open*.

```
// 1. Create a new election titled 'Prime'
$ node buildElection.js organizer org1 electionPrime ben
simon jim
--> Invoke e-voting chaincode: Create a new election
--> Result: OK
```

Listing 1: Creating a new election titled Prime

Once created, the election can be queried by invoking the *QueryElection* function in the smart contract, with the election name as parameter. The chaincode verifies the election exists and retrieves it from the blockchain state.

While the election is open, peers can submit votes to the election by invoking *SubmitVote* transaction. The code snippet for this function, as implemented for SGX, can be found in Appendix C below. The chaincode generates a composite key to encrypt the vote (as explained in Section 4.1) and stores the data in the peer's private storage until the election is closed.

```
// 2. Voter submits a vote for candidate Ben
$ node addVote.js voter1 org1 electionPrime ben
--> Invoke e-voting chaincode: Add a new vote
--> Voter ID: CN=voter1
--> Result: Vote: 738dcfb07a4f2308677dca8c1d4ce6cb29197...
--> Result: OK
```

Listing 2: Submitting a vote for candidate Ben

Upon submitting the vote to the election, a transaction ID is returned to the user by the smart contract and can be used by the voter to query their vote. This is because the transaction ID is used to re-construct the composite key that encrypted the vote. Additionally, querying the election confirms the hash of the vote in private storage is visible in the election. The votes remain hashed until the election is ended.

```
// 3. Query the vote submitted using the transaction ID
$ node queryVote.js voter1 org1 electionPrime 738dcfb07...
--> Invoke e-voting chaincode: Query vote
--> Result: Vote: {"voteFrom": "CN=voter1", "voteTo": "ben"}

// 4. Query the election to confirm vote was added
$ node queryElection.js organizer org1 electionPrime
--> Invoke e-voting chaincode: Get Election
--> Result: Election: {
    "name": "electionPrime",
    ...,
    "privateVotes": {
        "\u0vote\u0electionPrime\u0738dcfb07a4f...": {
            "hash": "9073790a8a849ffb6a5f0475a53532e53f6..."
        }
    },
    "publicVotes": {},
    ...
}
```

Listing 3: Querying the vote and election made

The organizer can close the election by invoking the *CloseElection* function in the smart contract, taking election name as parameter. This prevents votes submitted thereafter from being counted and acts as an indicator for when displaying the votes to the public.

In order to display the votes, the smart contract will first check if the election is closed before confirming the hash generated by the composite key matches the hash of the private vote that is now present in the election. It will also confirm that the peer invoking this function was the one to submit the vote.

Thereafter, the vote is made public and visible for anyone querying the election.

```
// 5. Close the election 'Prime'
$ node closeElection.js organizer org1 electionPrime
--> Invoke e-voting chaincode: Close election
--> Result: OK

// 6. Let 'voter1' display his vote from earlier
$ node displayVote.js voter1 org1 electionPrime 738dcf07a...
--> Invoke e-voting chaincode: Make vote public
--> Result: Vote: {"voteFrom": "CN=voter1", "voteTo": "ben"}
--> Result: OK

// 7. Query the election 'Prime' to see the vote public
$ node queryElection.js organizer org1 electionPrime
--> Invoke e-voting chaincode: Get Election
--> Result: Election: {
    "name": "electionPrime",
    ...,
    "privateVotes": {
        "\u0vote\u0electionPrime\u0738dcfb07a4f...": {
            "hash": "9073790a8a849ffb6a5f0475a53532e53f6..."
        }
    },
    "publicVotes": {
        "\u0vote\u0electionPrime\u0738dcfb07a4f...": {
            "voteFrom": "x509::CN=voter1...",
            "voteTo": "ben"
        }
    },
    ...
}
```

Listing 4: Process of closing the election and making votes public

Once the election is closed and the votes have been displayed, the organizer can determine the candidate with the most number of votes and declare them the winner. This can be done by invoking *EvaluateElection*. The code snippets for this can be found in the Appendices D and E of the report; the core logic involves counting up the tallies for each candidate and finding the largest total with greater-lesser checks.

```
// 8. Determine the candidate with highest number of votes
$ node evaluateElection.js organizer org1 electionPrime
--> Invoke e-voting chaincode: Evaluate election
--> Result: The candidate with most votes is ben
--> Result: OK

$ node queryElection.js organizer org1 electionPrime
--> Invoke e-voting chaincode: Get Election
--> Result: Election: {
    "name": "electionPrime",
    ...,
    "publicVotes": {
        "\u0vote\u0electionPrime\u0738dcfb07a4f...": {
            "voteFrom": "x509::CN=voter1...",
            "voteTo": "ben"
        }
    },
    "winner": "ben",
    "numVotes": 1,
    "status": "completed"
}
```

Listing 5: Evaluating the winner of the election

5 Evaluation

This section evaluates the effectiveness of our prototype and use of SGX with regards to smart contract security. Then, we also discuss the limitations of our model and potential improvements which can be made to the model & methodology to navigate pitfalls.

Strengths With the chaincode running in the SGX enclave, the operations of the smart contract cannot be tampered with by untrusted entities [5]. An unauthorized peer also cannot access data residing in the enclave; this can be seen above when only the voter that submitted his/her vote could query it using the transaction ID. The votes are encrypted for all other entities as they do not have access.

As touched upon in Section 2.1, rollback attacks pose a threat to privacy in smart contracts. This can occur when a malicious node manipulates the order in which transactions are run and breaking confidentiality by learning sensitive data [7]. In our example, this could be learning about the election winner, resetting the enclave and submitting further votes to see if there is a change in the winning candidate.

As a solution to this, the prototype makes use of a 'barrier' [7] in the form of closing the election and displaying the votes before the winner can be calculated. When the election is closed, a malicious peer would be unable to submit a new vote to the election and the tallies of the candidates only take into account the votes received when status is 'open'.

Limitations The use of Intel SGX can also have several disadvantages.

One of the shortcomings of our system is that peers are required to keep constant communication with the channel during the e-voting process and manually enter the transactionID to query their votes, display them etc. This can potentially be improved by storing the transactionID in the transient map of the chaincode; this way it can be retrieved and used for invocation without having to be saved in peer's private data collection.

Our implementation also lacks in-depth testing of different input cases which can cause non-determinism [17]. This can include global variables, timestamps, read-your-write cases etc. Testing for such vulnerabilities can make the prototype implementation more secure and would produce accurate results that support our conclusion.

SGX also has some inherent weaknesses which affect any applications making use of this technology. To start, SGX supports secure memory of up to 128 MB for its enclave [5] and there can be loss of performance if this is exceeded, which can have implications for scalability.

Moreover, since Fabric is used to deploy permissioned blockchains, the network using SGX is owner-controlled; thus, the platform owner is able to partially access the data in the enclave despite it being encrypted [5]. This can affect the security of the system, such as replay attacks [5] where the owner uses previously-run inputs in the enclave to achieve a favorable result.

Another limitation of SGX-based applications is the risk of receiving side-channel attacks. This arises when SGX shares resources with other programs and a malicious node uses those shared resources to infer private data in the enclave [5]. In our example, this can be seen by a rollback attack where a malicious node may aim to acquire control flow of the candidate tally by executing chaincode [5].

Future improvements To start, static analysis tools such as Oyente and Zeus can be applied to verify smart contract code and pinpoint potential vulnerabilities [17]. This is done without actually running the program and instead by examining symbolic properties such as the control flow, dependency graph, compliance & violation patterns, code behaviors etc. The results of testing the smart contract using these tools and comparing them to that in an unprotected environment can provide further evidence on the viability of Intel SGX for smart contract security.

The methodology undertaken during the implementation process can incorporate this with test-driven development. This approach would involve writing smart contract tests & conducting analyses throughout development and can also serve to provide relevant code coverage numbers along with, primarily, the potential vulnerability risks in the code.

6 Responsible Research

This section outlines the ethical implications of the research and the reproducibility of our findings.

The aim of the research is to investigate how Intel SGX can be used to enhance the security of smart contracts on Hyperledger Fabric. Going back to the methodology undertaken in the study, the literature used consists of scientific papers and articles which are in the public domain and available to anyone looking to conduct a similar research. The security threats and issues outlined in the paper are also publicly available and hence, poses little to no risk for those invested in the technology.

The prototype used in this study is also developed using a public framework and deployed on a local test network. The information depicted in the election scheme bears no significance to any real entities and no personal or user data was used during the development & trial of the application. The smart contract works anonymously and does not, at any point in its runtime, make attempts to collect data other than what the user choose to input. Hence, the study takes into consideration data privacy and is also in accordance with GDPR.

The paper also makes use of code samples and diagrams in order to show how Intel SGX has been integrated for our Fabric smart contract and the execution of such a contract. This is primarily intended as evidence for our findings but also serves to make the research more reproducible for future study. The samples have been kept concise, thus making them easy to follow and implement for developers. The full code for the application is available on Github [8] such that the project can be easily forked by developers who may find it useful. As the smart contract does not collect any user data, the code poses no risk of data misuse by anybody who may take up the repository.

7 Discussion & Related Works

This section looks at related works in the field by means of literature study and is meant to provide inspiration to conduct further research & development in the field.

With TEEs positioned to play an important role in context of smart contract security, several solutions have been derived from research in this field in the past few years.

To begin, Hawk [11] is a framework which enables developers to build privacy-preserving smart contracts. A Hawk program consists of a private and public portion, where the former is responsible for taking input data. A manager is established who can see the users' input and run the contract; Hawk guarantees transactions are confidential to any parties not in the contract [11]. In order to ensure the manager does not reveal any data, it can be be run inside an SGX container and instantiated such that the computation is secure. That said, there is also greater ovehead associated with Hawk protocols which can reduce efficiency [5].

Teechain [13] is a payment solution that performs transactions on top of Bitcoin and Ethereum. These transactions are off-chain [13] in the sense they only write to blockchain when settling the final payment. To ensure there are reliable payment channels among the various network parties, Teechain makes use of SGX to maintain nodes in the enclave that can exchange transactions and manage funds [5]. These channels allow transactions to be exchanged efficiently and results in impressive throughput for a payment system [13].

BITE [15] is a blockchain privacy-preserving approach that aims to protect smaller clients that outsource their computational load onto bigger blockchain nodes. Matetic et. al propose leveraging Intel SGX on these nodes and processing requests from small clients from within the enclave. Similar to the solution provided in this paper, BITE involves a client performing remote attestation prior to sending transactions over to the enclave running on the big node [5]. However, the way in which the node responds is quite different; BITE proposes two solutions for how the node can respond.

First, the node can respond with the Merkle paths of the block the transaction belongs to, in turn allowing the client to confirm that all its transactions have been committed to blockchain [15]. Alternatively, the enclave on the node keeps a record of unspent transaction outputs (UTXO) [5] and checks this upon receiving a client request to respond appropriately [15].

Trusted hardware for smart contracts is also beginning to find its role in cloud computing, where decentralized networks represent a potential shift from single cloud service providers. Airtnt [4] is a distributed cloud computing scheme that enables untrusted users to rent computations on secure enclaves. This is done by establishing a payment channel - not unlike Teechain - between the service provider and requester [5]. Intel SGX is applied to tackle the integrity issue and enable secure execution of the smart contract.

Similar to our approach, the requester can send transaction requests to the provider, which runs the chaincode in the enclave and returns the result encrypted by a composite key. After verifying with the IAS, the requester can send the payment details to the provider which writes to the state and updates the balance [5]. Subsequently, the requester can use the key to query the chaincode result obtained earlier and confirm the transaction. This method is conventional but Airtnt also experiences side-channel and single-point vulnerabilities, whereby a malicious node can get paid without fulfilling the client payload [4].

8 Conclusion

The paper explores the process of using Intel SGX to perform attestation and enhance the security of smart contracts on Hyperledger Fabric. This involved investigating the security issues these smart contracts are susceptible to (such as rollback attacks [14] and non-determinism [17]), conceptualizing the process by which the SGX enclave operates and implementing firsthand an e-voting smart contract that leverages the enclave. In doing so, we also provide an account of the process of development undertaken for the prototype and the method by which SGX was integrated into the smart contract.

The results obtained from running the application confirm isolation of chaincode logic and SGX-leveraged security where only authorized peers have access to confidential data in the network. Several improvements were suggested to the approach, including testing to find cases that could be exploited and the use of static analysis tools to pinpoint symbolic flaws in the implementation.

As a preferred framework for deploying enterprise-grade blockchain applications, Hyperledger Fabric stands to gain immensely from use of trusted hardware, or TEEs, to reinforce security in its smart contracts. This will encourage further adoption by various sectors and enable blockchain solutions that can securely address their customers' needs.

References

[1] Introduction - hyperledger fabric. https://hyperledger-fabric.readthedocs.io/en/release-2.2/blockchain.html. [Online; accessed 15-11-2021].

[2] Transaction flow - hyperledger fabric. https://hyperledger-fabric.readthedocs.io/en/release-2.2/txflow.html. [Online; accessed 15-11-2021].

[3] M.M. Abdul-Kader and S.G. Kumar. Enhanced privacy protection in blockchain using sgx and sidechains. *ICCIDS 2021: Computational Intelligence in Data Science*, 611:200–209, 2021.

[4] Mustafa Al-Bassam, Alberto Sonnino, Michał Król, and Ioannis Psaras. Airtnt: Fair exchange payment for outsourced secure enclave computations, 2018.

[5] Zijian Bao, Qinghao Wang, Wenbo Shi, Lei Wang, Hong Lei, and Bangdao Chen. When blockchain meets sgx: An overview, challenges, and open issues. *IEEE Access*, 8:170404–170420, 2020.

[6] Marcus Brandenburger. Fabric private chaincode. https://github.com/hyperledger/fabric-private-chaincode, 2019. [Online; accessed 15-11-2021].

[7] Marcus Brandenburger, Christian Cachin, Rüdiger Kapitza, and Alessandro Sorniotti. Blockchain and trusted computing: Problems, pitfalls, and a solution for hyperledger fabric. *ArXiv*, abs/1805.08541, 2018.

[8] Agniv Chatterjee. E-voting. https://github.com/agnivchtj/e-voting, 2022. [Online; accessed 20-01-2022].

[9] Batsayan Das, Srujana Kanchanapalli, and Vigneswaran R. Enhancing security and privacy of permissioned blockchain using intel sgx. 2019.

[10] IBM. What are smart contracts on blockchain? https://www.ibm.com/topics/smart-contracts. [Online; accessed 23-11-2021].

[11] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 839–858, 2016.

[12] Xueping Liang, Sachin Shetty, Deepak Tosh, Peter Foytik, and Lingchen Zhang. Towards a trusted and privacy preserving membership service in distributed ledger using intel software guard extensions. 12 2017.

[13] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. Teechain: A secure payment network with asynchronous blockchain access. page 63–79, 10 2019.

[14] Brandenburger Marcus and Cachin Christian. Challenges for combining smart contracts with trusted computing. In *SysTEX '18: Proceedings of the 3rd Workshop on System Software for Trusted Execution*, pages 20–21, 2018.

[15] Sinisa Matetic, Karl Wüst, Moritz Schneider, Kari Kostiainen, Ghassan Karame, and Srdjan Capkun. Bite: Bitcoin lightweight client privacy using trusted execution. pages 783–800, 08 2019.

[16] Mark Soelman, Vasilios Andrikopoulos, Jorge Perez, Vasileios Theodosiadis, Karel Goense, and Arne Rutjes. Hyperledger fabric: Evaluating endorsement policy strategies in supply chains. pages 145–152, 08 2020.

[17] Kazuhiro Yamashita, Yoshihide Nomura, Ence Zhou, Bingfeng Pi, and Sun Jun. Potential risks of hyperledger fabric smart contracts. In *2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 1–10, 2019.

9 Appendix A

This diagram visualizes the technical roadmap followed during the course of the project.

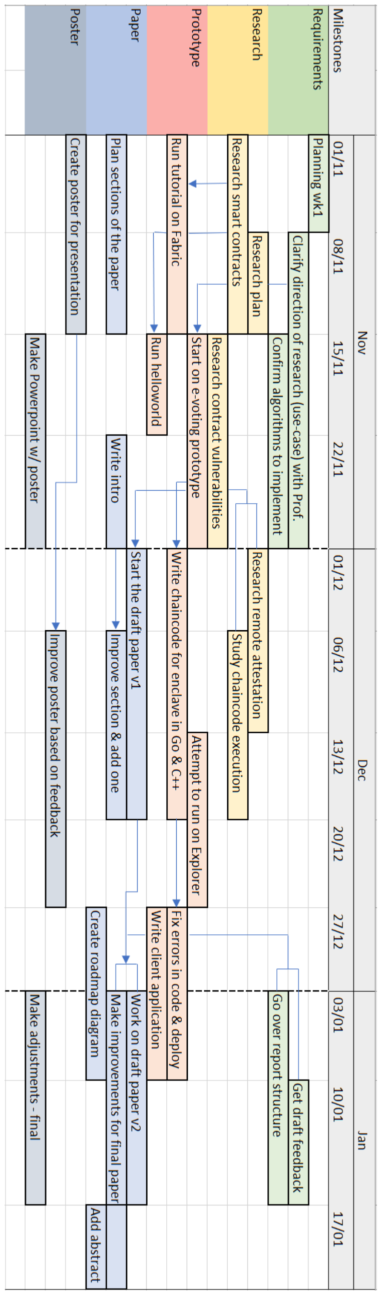
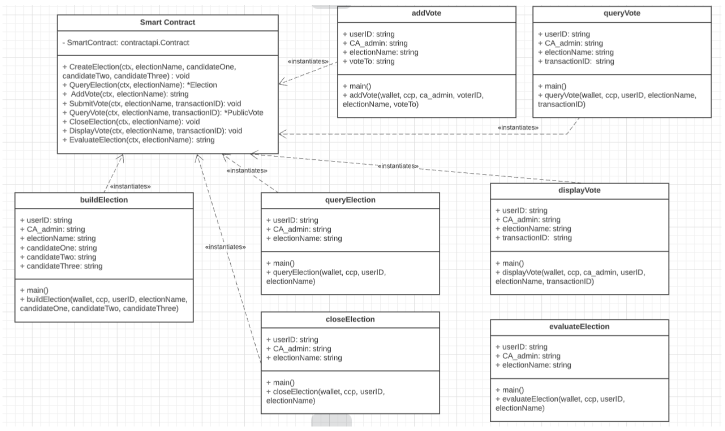


Figure 5: Technical roadmap followed for the project

10 Appendix B

The class diagram representing the dependencies between the client application in our prototype and the Fabric smart contact for e-voting.



Smart Contract
- SmartContract: contractapi.Contract
+ CreateElection(ctx, electionName, candidateOne, candidateTwo, candidateThree) : void
+ QueryElection(ctx, electionName): *Election
+ AddVote(ctx, electionName): string
+ SubmitVote(ctx, electionName, transactionID): void
+ QueryVote(ctx, electionName, transactionID): *PublicVote
+ CloseElection(ctx, electionName): void
+ DisplayVote(ctx, electionName, transactionID): void
+ EvaluateElection(ctx, electionName): string
addVote
+ userID: string
+ CA_admin: string
+ electionName: string
+ voteTo: string
+ main()
+ addVote(wallet, ccp, ca_admin, voterID, electionName, voteTo)
queryVote
+ userID: string
+ CA_admin: string
+ electionName: string
+ transactionID: string
+ main()
+ queryVote(wallet, ccp, userID, electionName, transactionID)
buildElection
+ userID: string
+ CA_admin: string
+ electionName: string
+ candidateOne: string
+ candidateTwo: string
+ candidateThree: string
+ main()
+ buildElection(wallet, ccp, userID, electionName, candidateOne, candidateTwo, candidateThree)
queryElection
+ userID: string
+ CA_admin: string
+ electionName: string
+ main()
+ queryElection(wallet, ccp, userID, electionName)
displayVote
+ userID: string
+ CA_admin: string
+ electionName: string
+ transactionID: string
+ main()
+ displayVote(wallet, ccp, ca_admin, userID, electionName, transactionID)
closeElection
+ userID: string
+ CA_admin: string
+ electionName: string
+ main()
+ closeElection(wallet, ccp, userID, electionName)
evaluateElection
+ userID: string
+ CA_admin: string
+ electionName: string
+ main()
+ evaluateElection(wallet, ccp, userID, electionName)
«instantiates»

Figure 6: Class diagram for e-voting prototype developed

11 Appendix C

```
std::string submitVote(
    std::string election_name, std::string voter_name,
    std::string vote_to, shim_ctx_ptr_t ctx
)
{
    // check if election already exists
    uint32_t election_bytes_len = 0;
    uint8_t election_bytes[MAX_VALUE_SIZE];
    get_state(
        election_name.c_str(),
        election_bytes,
        sizeof(election_bytes),
        &election_bytes_len,
        ctx
    );

    if (election_bytes_len == 0)
    {
        LOG_DEBUG("Election␣needs␣to␣already␣exist!");
        return ELECTION_DOES_NOT_EXIST;
    }

    // check if election is closed
    election_t election;
    unmarshal_election(&election, (const char*)election_bytes,
    election_bytes_len);

    if (!election.status) {
        LOG_DEBUG("Election␣must␣be␣open␣to␣submit␣new␣votes.");
        return ELECTION_ALREADY_CLOSED;
    }

    // Create composite key to encrypt vote
    // If vote already exists, we just overwrite it
    std::string new_key("\u00" + election_name
    + "\u0" + voter_name + "\u0");

    vote_t new_vote;
    new_vote.vote_from = voter_name;
    new_vote.vote_to = vote_to;

    // convert to json and store
    std::string json = marshal_vote(&new_vote);
    put_state(new_key.c_str(), (uint8_t*)json.c_str(),
    json.size(), ctx);

    return OK;
}
```

Listing 6: Code sample for *SubmitVote* function

12 Appendix D

```
std::string evaluateElection(
    std::string election_name, shim_ctx_ptr_t ctx
)
{
    ...

    // the winner of the election
    std::string election_result;

    // get all votes
    std::string composite_key = "\u00" + election_name
    + "\u0";
    int c_one_count = 0;
    int c_two_count = 0;
    int c_three_count = 0;
    std::map<std::string, std::string> votes;
    get_state_by_partial_composite_key(
        composite_key.c_str(), votes, ctx
    );

    if (votes.empty())
    {
        LOG_DEBUG("There␣are␣no␣votes␣submitted.");
        election_result = ELECTION_NO_VOTES;
    }
    else
    {
        // Find candidate w/ most votes
        LOG_DEBUG("All␣considered␣votes:");
        for (auto v : votes)
        {
            vote_t vote;
            unmarshal_vote(
                &vote, v.second.c_str(), v.second.size()
            );

            LOG_DEBUG(
                "Election:␣Voter␣\t%s␣picked␣candidate:␣%d",
                vote.vote_from.c_str(),
                vote.vote_to.c_str()
            );

            if (
                vote.vote_to == election.candidate_one
            ) {
                c_one_count += 1;
            } else if (
                vote.vote_to == election.candidate_two
            ) {
                c_two_count += 1;
            } else {
                c_three_count += 1;
            }
        }

        ...
    }

    ...
}
```

Listing 7: Code sample showing tallying up in *evaluateElection*

```
    }
    else {
        ...

        // Finding candidate with most votes
        candidate_t winner;
        winner.name = "";
        winner.num_votes = -1;
        int draw = 0;

        if (c_one_count > winner.num_votes)
        {
            draw = 0;
            winner.name = election.candidate_one;
            winner.num_votes = c_one_count;
        }

        if (c_two_count > winner.num_votes)
        {
            draw = 0;
            winner.name = election.candidate_two;
            winner.num_votes = c_two_count;
        } else if (c_two_count == winner.num_votes)
        {
            draw = 1;
        }

        if (c_three_count > winner.num_votes)
        {
            draw = 0;
            winner.name = election.candidate_three;
            winner.num_votes = c_three_count;
        } else if (c_three_count == winner.num_votes)
        {
            draw = 1;
        }

        if (draw != 1)
        {
            LOG_DEBUG("Winner␣is:␣%s␣with␣%d␣votes",
            winner.name.c_str(), winner.num_votes);
            election.winner = winner.name.c_str();
            election_result = marshal_candidate(&winner);
        }
        else
        {
            LOG_DEBUG("DRAW");
            election_result = ELECTION_DRAW;
        }
    }

    ...
}
```

Listing 8: Code sample for determining winner in *evaluateElection*

13 Appendix E

```
std::string evaluateElection(
    std::string election_name, shim_ctx_ptr_t ctx
) {
    ...

    if (votes.empty())
    {
        LOG_DEBUG("There␣are␣no␣votes␣submitted.");
        election_result = ELECTION_NO_VOTES;
```