Formal Security Analysis for Blockchain-based Software Architecture

Nacha Chondamrongkul*, Jing Sun†, Ian Warren‡
Department of Computer Science
The University of Auckland
Auckland, New Zealand
*ncho604@aucklanduni.ac.nz † jing.sun@auckland.ac.nz ‡ i.warren@auckland.ac.nz

***Abstract*—During the design phase, security as a non-functional requirement needs to be analysed to address vulnerabilities in the architecture design. Without such analysis, security vulnerabilities can be propagated to the implementation. However, security analysis is an error-prone task, especially in complex systems that apply blockchain technology. Without proper security controls applied, the interaction among software components and the blockchain may pose security risks. This paper presents a security analysis approach based on a formal model of blockchain-based architecture design. Our approach can automatically identify specific security vulnerabilities and generate informative scenarios that show how attacks may impact the blockchain. We have evaluated our approach with an example system and found it performs well in identifying an extensible class of security vulnerabilities.**

***Index Terms*—Software Architecture, Security Analysis, Blockchain, Formal Method**

I. Introduction

Blockchain is an emerging technology that many software engineers apply to secure data and resources in software systems. To protect data against tampering, blockchain is usually applied as software components that provide data storage, computation services and control functions [1]. Designing the software architecture for blockchain-based systems involves off-chain components that are outside the blockchain network and on-chain components that have access to a node in the blockchain network. With the combination of on-chain and off-chain components, protecting a blockchain-based system is not limited to the blockchain, but the structure and behaviour of other components that interface with the blockchain should also be considered to eliminate risks of attacks [2]. At the design stage, beside verifying the design against the functional requirements, the architecture security analysis task is conducted using software architecture design models [3]. This task involves tracing through the components and security configuration to pinpoint any security flaws in the architecture design. It helps to prevent the architectural security flaws propagating to the implementation.

There are approaches proposed to verify security in the software architecture design in general, such as [4] and [5]. These approaches apply logical rules and metrics that are hard-coded in source code. Almorsy et al. [6] presented an extensible tool to identify different security attacks based on the signatures representing security metrics and vulnerabilities. However, the blockchain-based system has specific interaction behaviour between on-chain and off-chain components that requires tracing through different attack scenarios to find both direct and indirect impact. The scenario-based traceability is not yet addressed by the existing approaches. There are approaches particularly proposed to analyse security in the blockchain. Luu et al. [7] enhanced the operational semantics of Ethereum to prevent security bugs in the smart contract. Chaieb et al. [8] proposed a verifiable protocol that applies encryption to ensure privacy and security properties. Some approaches [9] [10] [11] have been proposed to verify security in the smart contract by analysing source code. However, these approaches focus either on the structure within the blockchain or the implementation of the blockchain that does not yet exist at the design stage. There is still lack of approaches that can analyse security in blockchain-based application at the software architecture level.

This paper presents an approach that supports architecture security analysis based on a formal model of blockchain-based architecture design. At the design stage, our analysis approach aims at verifying security as a non-functional requirement after the design has complied with functional requirements. Our approach can automatically identify security characteristics and generate scenarios that show how attacks may have direct or indirect impact on the blockchain. The contribution of our approach can be summarised as follows. First, the formal modelling of blockchain-based software architecture design is proposed to describe the structural and behavioural aspect of blockchain-based systems. Second, a set of formally described security characteristics representing security metrics and vulnerabilities is presented. This set is extensible to support other characteristics not addressed in this paper. Last, our approach has been implemented as a tool. This tool allows users to seamlessly perform modelling and security analysis of the blockchain-based system at the architectural level.

The rest of this paper is organized as follows. Section II explains a motivating example of blockchain-based system and its security challenges. Section III presents the modelling and analysis approach. Section IV presents the evaluation of our approach. This paper is concluded in Section V.

DOI reference number:10.18293/SEKE2020-024

II. Security in Blockchain-based Architecture

This section discusses an architecture design for an example application that applies blockchain. In addition, this section identifies the security threats in the architecture design.

A. Motivating Example

AgriDigital is a motivating example that we use in this work. In this paper, we briefly introduce the system, but more details can be found in [12]. The system applies blockchain technology to build digital trust between parties in the agriculture supply chain such as farmers, suppliers and transporters. Digital trust allows different parties to track and transfer commodities while financial transactions can be made transparently. The architecture design of AgriDigital can be found in Figure 1. The *AD Web App* and *Provenance Web App* are web applications that allow users to perform administration operations and enter the record of commodities. After a user enters or updates information, *Provenance Integration* publishes it to involved components such as *AD Message Bus* and *Blockchain Message Bus*. *Blockchain Message Bus* notifies *Blockchain Integrator* about updated information. This design applies the Oracle pattern [13] that allows the off-chain component to push information from the external world to the blockchain. Therefore, when *Blockchain Integrator* is notified about the new information, it creates a new block that keeps financial transactions on the *Public Blockchain*, while another block is created and appended on *Private blockchain* for an updated status of commodities. Some status of commodities can be detected and updated through *IOT Sensor* that helps to measure the condition of commodities such as temperature and humidity. These records of updated status are also created as a block on *Private Blockchain*. Reverse Oracle pattern has been applied between *Digital Wallet* and *Public Blockchain*. This pattern allows *Digital Wallet* to fetch financial information from the blockchain for processing.

Some off-chain components, such as *AD AppServer*, *AD WebServer* and *IOT WebApp* are deployed on local dedicated servers, while others, such as *Provenance Integration*, *Provenance WebApp* and *Blockchain Integrator*, are deployed on the public cloud infrastructure. The deployment configuration of these components poses security threats to the on-chain components such as *Private Blockchain*, implemented with Quorum, and *Public Blockchain* implemented with Ethereum.

B. Attack Scenarios

The on-chain components are vulnerable to attack as they are in the request flow triggered by the off-chain components. In this work, we focus on prominence attack scenarios that usually have security impacts on the blockchain, namely data disclosure and data tampering.

1) Data Disclosure: A key decision in designing blockchain-based software is determining whether the data should be placed on-chain or kept off-chain [1]. The same copy of data in the blockchain is shared among all nodes that run in the blockchain network. If adversaries have access to any node, they can also access the data stored in the blockchain.

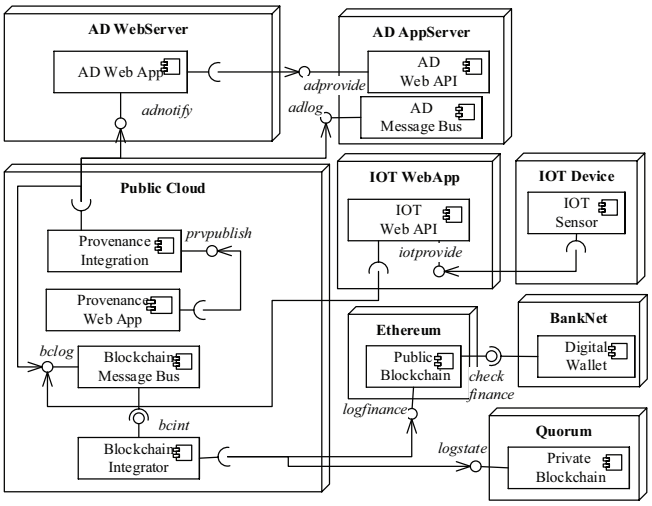

Fig. 1. AgriDigital System

Even though data on the blockchain can be encrypted, the data could be disclosed if the private key is stolen [2]. Securing the connections and components that access the blockchain is therefore important. In our example system, if *Digital Wallet* or its connection to *Public blockchain* is compromised, the financial transaction can be disclosed.

2) Data Tampering: Data in the blockchain is known to be nearly immutable and tamper-proof since hash functions are one-way and collision resistant [14]. However, when the oracle component feeds data to the blockchain, the data are assumed to be trusted by all participants. If the oracle is compromised, the adversaries could modify data before it is stored on the blockchain. The oracle should be safeguarded by appropriate security controls to prevent this scenario. In our example system, the *Blockchain Integrator* serves as an oracle component. If the *Blockchain Integrator* or the components that feed data to it such as *IOTWebAPI* and *Provenance Integrator* are compromised, the data on *Private Blockchain* and *Public Blockchain* can be tampered with.

C. Security Metrics

Different security metrics are used to measure the security in the software architecture design. In this work, we focus on the security metrics that suit assessing blockchain-based software architecture.

1) Attack Surface: This metric measures the number of weaknesses in the system that the adversaries can use to attack the system. The attack surface is usually where the system is open to the external environment such as where data are entered or the components that are publicly accessible. The lower the number of attack surface, the more secure the system is. In our example, *Public Blockchain* is an attack surface as it allows any entity to join and run a node in the blockchain network. *BCIntegrator* and *IOTWebAPI* are also attack surfaces as they are accessible from the public network.

2) Least Priviledge: This metric ensures that minimal access to critical data or operations is granted to users or other components in the system. From an architectural perspective,

the number of components that can access critical data should be limited. In blockchain-based software, the on-chain components are critical components that should be accessed by no more than necessary off-chain components. In our example, *BCIntegrator* and *DigitalWallet* are the only components that have direct access to the on-chain components.

3) Defense In Depth: This metric measures how security controls are applied at different points in the system. To protect data in the blockchain, the components that access the blockchain should employ security controls at the component, host and network layers. For architectural analysis, we can calculate the ratio of off-chain components that access on-chain components and which apply security controls compared to the total number of off-chain components that access on-chain components. The higher the ratio value is, the more secure the system is. In our example, *BCIntegrator* and *DigitalWallet* should apply authentication and authorization controls, as well as a firewall that prevents incoming malicious traffic.

III. Formal Security Analysis

Our formal security analysis for blockchain-based software architecture combines ontology reasoning and model checking techniques, as shown in Figure 2. First, the architecture design is formally modelled as the component and connector (C&C) view and the deployment view. Second, the ontology reasoner is used to identify security vulnerabilities in the model, based on the ontology description of architecture patterns and security characteristics. These ontology descriptions are defined as classes kept in the ontology library. Third, assertions are inserted into the behavioural model based on the identified vulnerabilities. Finally, the model checker processes the assertions against the model and generates the security scenarios.

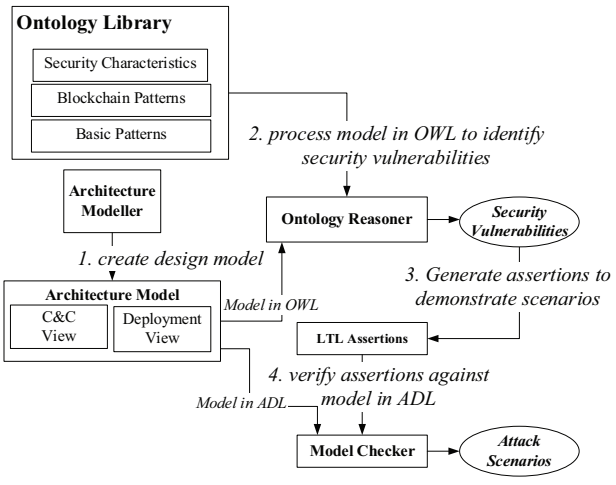

Fig. 2. Overall Analysis Process

A. Blockchain Architecture Modelling

Both structure and behaviour of the model are essential to analyse the security in the design phase. Structure in architecture design can be formally defined as ontology representation using the Ontology Web Language (OWL). Behaviour respect to interaction among components, can be defined using the Architecture Description Language (ADL).

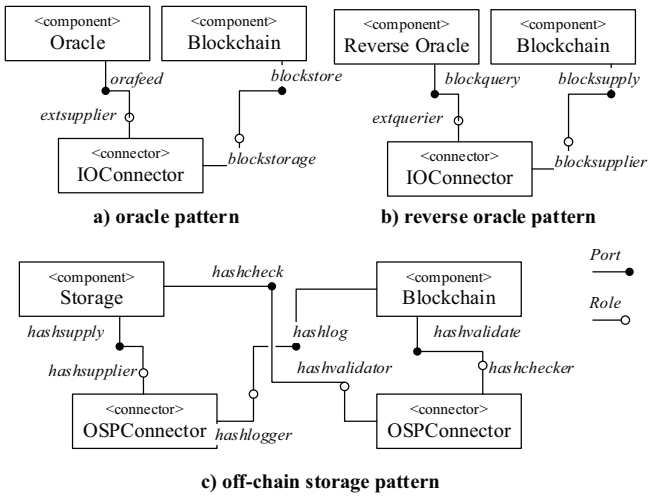

Fig. 3. Architecture Patterns for Blockchain

Architecture security analysis is usually performed on the component and connector view (C&C) and the deployment view of software architecture design. These views are therefore semantically described in the design model. Ontology classes are predefined to support the modelling of the architecture design [15]. As shown in Figure 3, we have defined four connector types to support three architecture patterns, namely Oracle, Reverse Oracle and Off-Chain Storage. These patterns, proposed by Xu et al. [13], have been applied to blockchain-based software. In AgriDigital, we applied Oracle and Reverse Oracle, as well as other basic patterns such as Client-Server and Publish-Subscribe (more details can be found in [15]).

The design model of a blockchain-based system can be defined by creating ontology individuals based on defined classes to represent different entities in the C&C view, such as Component, Connector, Port and Role. Another set of individuals is created to represent different entities in the deployment view such as device, execution environment and communication link. These entities are linked to describe how components are deployed within the infrastructure. Due to the page limit, we present a subset of the AgriDigital model as shown below[1].

$$
\begin{aligned}
&Individual(ex : pubwire\\
&\quad value(ex : hasRole\ ex : pubextsupplier)\\
&\quad value(ex : hasRole\ ex : pubblockstorage))\\
&Individual(ex : walletwire\\
&\quad value(ex : hasRole\ ex : wlextquerier)\\
&\quad value(ex : hasRole\ ex : wlblocksupplier))\\
&Individual(ex : pubextsupplier\ type(ex : Extsupplier))\\
&Individual(ex : pubblockstorage\ type(ex : Blockstorage))\\
&Individual(ex : wlextquerier\ type(ex : Extquerier))\\
&Individual(ex : wlblocksupplier\ type(ex : Blocksupplier))
\end{aligned}
$$

The *pubwire* individual represents a connector that links *Public Blockchain* and *Blockchain Integrator*. The *walletwire* individual represents a connector that links *Public Blockchain* and *Digital Wallet*. The code below shows some individuals

[1]The complete OWL model of AgriDigital can be found at http://bit.ly/3bm18YB

representing components and ports. The individual representing the component *BlockchainIntegrator* has *bcint* port that attaches to *pubextsupplier* defined above. This definition applies the Oracle pattern. The *checkfinance* port attached to the *wlextquerier* role applies the reverse oracle pattern.

Individual(*ex* : *BlockchainIntegrator*
  value(*ex* : *hasPort ex* : *bcint*))
Individual(*ex* : *PublicBlockchain*
  value(*ex* : *hasPort ex* : *logfinance*)
  value(*ex* : *hasPort ex* : *checkfinance*))
Individual(*ex* : *bcint*
  value(*ex* : *hasAttachment ex* : *pubextsupplier*))
Individual(*ex* : *logfinance*
  value(*ex* : *hasAttachment ex* : *pubblockstorage*))
Individual(*ex* : *checkfinance*
  value(*ex* : *hasAttachment ex* : *wlextquerier*))

For the deployment view, another set of individuals are defined partially shown below. *Docker1* represents the container situated on *PublicCloud* where *BlockChainIntegrator* is deployed. *Port12037* represents a communication port that *bcint* uses to communicate to *Public Blockchain*. *Link6* represents the network communication that links the communication ports that the ports of *BlockchainIntegrator* and *Public Blockchain* are bound to.

Individual(*ex* : *Docker*1
  value(*ex* : *isNodeOf ex* : *PublicCloud*)
  value(*ex* : *hasDeployment ex* : *BlockchainIntegrator*))
Individual(*ex* : *Port*12037
  value(*ex* : *hasBind ex* : *bcint*))
Individual(*ex* : *Link*6
  value(*ex* : *hasCommPort ex* : *Port*12037)
  value(*ex* : *hasCommPort ex* : *Port*8889))

The interaction behaviour of components can be defined in ADL as presented in our previous work [16]. This behaviour model allows the model checker to trace through different states occurring in the blockchain-based software. The tracing helps to generate a scenario that shows how an attack happens. Below is part of model in ADL describing the behaviour of the Oracle and Reverse Oracle pattern[2]. The ADL model also includes definition of the component and system configuration, which defines role and port attachment, as well as how they are executed at runtime.

connector IOConnector {
  role blockstorage() = *token*?*j* → *process*
    → *stored* → *blockstorage*();
  role extsupplier(*j*) = *process*
    → *token*!*j* → *Skip*; }
connector ROConnector {
  role extquerier(*j*) = *request* → *uid*!*j*
    → *res*?*j* → *process* → *Skip*;
  role blocksupplier() = *uid*?*j* → *process*
    → *res*!*j* → *blocksupplier*(); }

B. Security Characteristic Analysis

Beside the ontology classes supporting blockchain-based software architecture modelling, we also define ontology classes for classifying security characteristics in the model. Three characteristics, namely Attack Surface, Defence in Depth and Least Privilege, are used to calculate metric values that measure how secure the system is. Two characteristics, namely Data Tampering and Data Disclosure, are used to trace attack scenarios. These ontology classes are kept in the ontology library. Other characteristics not addressed here can be defined by creating new class inherited from existing classes, or conditionally capturing different properties in the class definition.

1) Attack Surface: To formally define the attack surface, an ontology class called *AttackSurface* is created with a logic to describe the components that are publicly accessible through the internet or public network such as public blockchain. In other words, a component is an attack surface if it has an incoming communication port that binds to the internet link.

$$\begin{aligned} \mathit{AttackSurface} \equiv\ & \mathit{Component} \sqcap \exists\, \mathit{hasPort} \\ & (\mathit{Port} \sqcap \exists\, \mathit{isBindTo}\ (\mathit{IncomingCommPort} \\ & \sqcap \exists\, \mathit{isCommPortOf}\ \mathit{InternetLink})) \end{aligned}$$

2) Least Privilege: In a blockchain-based system, an on-chain component is considered as a critical component. We use an ontology rule to select the components that have access to the on-chain components. This rule is defined in Semantic Web Rule Language (SWRL) as shown below. It describes the connection between two components: *comp1* and *comp2*, which *comp2* is a *Blockchain*.

$$\begin{aligned} & \mathit{hasPort}(\mathit{comp}1, p1) \wedge \mathit{hasPort}(\mathit{comp}2, p2) \\ & \wedge\ \mathit{hasAttachment}(p1, r1) \wedge \mathit{hasAttachment}(p2, r2) \\ & \wedge\ \mathit{hasRole}(\mathit{con}, r1) \wedge \mathit{hasRole}(\mathit{con}, r2) \\ & \wedge\ \mathit{Blockchain}(\mathit{comp}2) \rightarrow \mathit{LeastPriviledge}(\mathit{comp}1) \end{aligned}$$

3) Defence in Depth: To classify the communication ports that use security controls, the ontology class, namely *AuthenticatedCommPort*, *AuthorizedCommPort*, *FirewalledCommPort* and *InputSantizedCommPort* are defined. As we aim to capture the components that have access to the blockchain and apply security controls, *DefenseInDepth* is defined as a subset of *LeastPriviledge* that has its port bound to an incoming secured communication port.

$$\mathit{AuthenticatedCommPort}, \mathit{AuthorizedCommPort} \sqsubseteq \mathit{SecureCommPort}$$

$$\mathit{FirewalledComPort}, \mathit{InputSantizedCommPort} \sqsubseteq \mathit{SecureCommPort}$$

$$\begin{aligned} \mathit{DefenceInDepth} \equiv\ & \mathit{LeastPriviledge} \sqcap \exists\, \mathit{hasPort}\ (\mathit{Port} \sqcap \\ & \exists\, \mathit{isBindTo}(\mathit{IncomingCommPort} \sqcap \mathit{SecureCommPort})) \end{aligned}$$

4) Data Disclosure: Data disclosure occurs on a connection that transfers data as plain text over unencrypted protocols such as *http* and *ftp*. *PlainLink* is defined to represent connectors that communicate using the insecure protocols. An ontology class called *DataDisclosureConnector* is defined as below to describe the connector that is vulnerable to data disclosure, as it transfers data in plain text.

$$\mathit{HTTPLink}, \mathit{FTPLink} \sqsubseteq \mathit{PlainLink}$$

$$\begin{aligned} \mathit{DataDisclosureConnector} \equiv\ & \mathit{Connector} \sqcap (\exists\, \mathit{hasLinkVia} \\ & (\mathit{PlainLink} \sqcap \mathit{InternetLink})) \end{aligned}$$

5) Data Tampering: When data is transferred over a connector that is vulnerable to data disclosure, the data is also be vulnerable to tampering if the connector is on a communication link that has no input sanitisation or authorisation. Without input sanitisation the data may come from an unknown source, and the data can be changed during the transmission without any authorisation. *DataTamperingConnector* class is defined as below.

$$\mathit{NoInputSanitizedCommPort} \equiv \mathit{hasInputSantization} <= 0$$

$$\mathit{UnauthorizedCommPort} \equiv \mathit{hasAuthorization} <= 0$$

$$\begin{aligned} \mathit{DataTamperingConnector} \equiv\ & \mathit{DataDisclosureConnector} \\ & \sqcap \exists(\mathit{hasLinkVia}(\mathit{CommunicationLink} \sqcap \exists\, \mathit{hasCommPort} \\ & (\mathit{NoInputSanitizedCommPort} \sqcap \mathit{UnauthorizedCommPort}))) \end{aligned}$$

[2]The complete ADL model of AgriDigital can be found at http://bit.ly/2vkmEMK

C. Security Attack Scenarios Analysis

With the ontology classes defined as previously described, the ontology reasoner can pinpoint which connector is vulnerable to data tampering and data disclosure. We use this information to generate attack scenarios by inserting attacker components into the design model. These attacker components represent software components that adversaries use. Then, Linear Temporal Logic (LTL) assertions are generated and inserted into the behavioural model in ADL. This allows the model checker to trace how the components interact with each other in response to the attacker's request. Algorithm 1 shows how the attacker component and LTL assertions are generated. This algorithm loops through *VulnConnSet* that contains inferred individuals that are of type *DataTamperingConnector* or *DataDisclosureConnector*. The attacker component is added to the model, and its *attack* port is attached to the outbound role of the vulnerable connector. The outbound role is where the request is initiated to make system responses. All inbound roles that handle the requests are iterated in the second loop. This iteration finds the port attached to an inbound role and its component to generate a LTL assertion.

Algorithm 1 Attack Scenarios Generation

```
1: Input model is a design model
2: Input VulnConnSet is a set of vulnerable connectors
3: for vulconn ∈ VulnConnSet do
4:     create attacker as an attacker component
5:     create attack port of attacker
6:     attach attack port to outbound role of vulconn
7:     for inRole ∈ vulconn.getInboundRole() do
8:         for comp ∈ model.getComponent() do
9:             for port ∈ comp.getPort() do
10:                if port has inRole attached then
11:                    define a LTL assertion with
12:                    vulconn as vulnerable connector
13:                    comp as target component
```

The LTL assertion that proves the attack scenario is created according to the formula below. The *vevnt* represents the event triggered from the attached outbound role (*outrole*) of the vulnerable connector (*vulconn*). The *cevnt* represents the event triggered by the target component (*targetcomp*) that responds to the request issued by the attacker component (*attacker*). In other words, this LTL assertion checks whether the target component is always eventually invoked when the attacker makes a request.

$$\Box(attacker.vulconn.outrole.vevnt \rightarrow \Diamond\ targetcomp.inport.cevnt)$$

For example, if the connector between *Provenance Integration* and *Blockchain Message Bus* carries the plain text over the internet through the cloud-based container, this is vulnerable to data tampering. The attacker component could be added here. The LTL assertion is defined to generate the scenarios of this attack as $\Box(Attacker.prvmsgwire.publisher.process \rightarrow \Diamond\ BlockchainMessageBus.bclog.evntlogged)$. The model checker verifies this assertion to be valid. The negation of this assertion gives a counterexample showing a state trace, as shown below.

init $\rightarrow$ *Attacker_ attack_ attacked*
$\rightarrow$ *Attacker_ prvmsgwire_ publisher_ process*
$\rightarrow$ *prvmsgwire_ pevt*!87 $\rightarrow$ *prvmsgwire_ pevt*?87
$\rightarrow$ *BlockchainMessageBus_ bclog_ evntlogged ...*
$\rightarrow$ *BlockchainIntegrator_ bcint_ sendtobc*
.. $\rightarrow$ *PublicBlockChain_ logfinance_ finlogged*
$\rightarrow$ *PrivateBlockChain_ logstate_ statelogged ...*

This state trace illustrates the sequence of how components and connectors are involved in the scenario. It shows that both *Public Blockchain* and *Private Blockchain* are affected as they are consequently invoked by *Blockchain MessageBus* and *Blockchain Integrator* respectively. This information supports software engineers to analyse and fix the configuration in the design model. In this case, the communication link to the *Blockchain Message Bus* should employ an encrypted protocol like https. Furthermore, an authorisation control should be applied to *Blockchain MessageBus* and *Blockchain Integrator* to prevent data tampering. Hence, the state trace helps to pinpoint where the security configuration should be fixed.

IV. Evaluation

This section presents how we evaluated our approach using the motivating example. The detail of how we conducted the evaluation is explained, followed by the results and discussion.

A. Experimental Setup

We have implemented our approach as a software framework to automate the security analysis process. *Arch Modeller*[3] is implemented as a graphical user interface tool to support modelling the architecture design and performing security analysis. This tool allows users to draw the graphical diagrams representing the architecture design model using the Eclipse Modelling Framework (EMF), and converts them into the structural model in OWL and the behavioural model in Wright#. The model in OWL can be processed by the ontology reasoner to identify the security characteristics. Then, the attacker components are automatically inserted and linked to the vulnerable connectors; at the same time the LTL assertions are inserted into the behavioural model. The behavioural model is processed by PAT ADL [16] and returns state traces as output, which is not possible using results from ontology reasoning alone.

The model of AgriDigital[4] has been created using Arch Modeller. In our evaluation, we assume the design model serves functional requirements correctly before conducting the security analysis. We aim to assess the completeness and soundness of the security analysis approach. After the model has been processed, the result is analysed to determine whether there are true-positives (TP) or false-positives (FP). We also analysed the design model to find false-negative (FN) results that are missing from the results given by the ontology reasoning. The precision and recall rate has been calculated

[3]Arch Modeller can be found at http://bit.ly/2m3LITT
[4]The model of AgriDigital can be found at http://bit.ly/2SfxHjE

according to [6]. This evaluation was carried out using an Intel Core i7 CPU with 8.00 GB Ram computer.

B. Evaluation Result

The evaluation result is summarised in Table I. The ontology reasoner took 9,123 milliseconds to detect all five security characteristics in the AgriDigital model. We have calculated the precision and recall rate to prove the soundness and completeness of the results. It can be seen that most of the detection could achieve a 100% recall rate for all characteristics. The precision rate can achieve 100% for most characteristics except data tampering, as we have found a false-positive result. It is important to note that no false-negative has been found, as the breach to the whole system may be caused by a missing security flaw. However, the accuracy of the detection relies on how accurately the security characteristics are defined. Also, the design model needs to be checked against functional requirements before the security analysis is performed.

TABLE I
EVALUATION RESULTS

Characters	TP	FP	FN	Precision	Recall
Attack Surface	7	0	0	1.0	1.0
Least Priviledge	2	0	0	1.0	1.0
Defence in Depth	2	0	0	1.0	1.0
Data Disclosure	5	0	0	1.0	1.0
Data Tampering	3	1	0	0.75	1.0

Table II shows the statistics when the scenarios have been generated based on identified connectors that are vulnerable to different scenarios. It can be seen that some assertions give state traces that show the on-chain components have indirect impacts (as indicated in the last column). These on-chain components are not the target directly connected to the vulnerable connectors, but they have a consequent impact from the attacks as some part of the request flow leads to them. In addition, the time taken by the model checker to process is reasonable for this size of model, however more comprehensive evaluation is required to better understand its performance.

TABLE II
SCENARIO GENERATION

Assertion	Scenario	Time(ms)	State#	Impact?
#1	Data Tampering	687	22772	None
#2	Data Tampering	930	38014	Indirect
#3	Data Tampering	19	781	Indirect
#4	Data Disclosure	565	22772	None
#5	Data Disclosure	937	38014	Indirect
#6	Data Disclosure	213	781	Indirect
#7	Data Disclosure	5	13	Direct
#8	Data Disclosure	3462	79	Indirect

V. CONCLUSION

This paper presents a security analysis approach for blockchain-based software architecture design. Based on the ontology description of the blockchain architecture pattern and security characteristics, our approach can identify vulnerabilities in the design model. Attack scenarios can be generated based on the identified vulnerabilities using the model checking technique. The result can be used to determine whether the blockchain has any impact from the attacks. We have evaluated our approach with an example system, and the results showed that it performs reasonably well. Our set of ontology descriptions can be extended by inheriting or defining ontology classes to describe other security metrics or vulnerabilities. Security engineers can extend our ontology library, which allows software engineers to analyse security in the blockchain-based system in a standardised way.

For future work, we plan to conduct a more comprehensive evaluation of our approach and explore how it can be applied in the software construction stage.

REFERENCES

[1] X. Xu, I. Weber, and M. Staples, *Blockchain in Software Architecture*. Cham: Springer International Publishing, 2019, pp. 83–92.

[2] M. Saad, J. Spaulding, L. Njilla, C. A. Kamhoua, S. Shetty, D. Nyang, and A. Mohaisen, "Exploring the attack surface of blockchain: A systematic overview," *CoRR*, vol. abs/1904.03487, 2019.

[3] W. D. Yu and K. Le, "Towards a secure software development lifecycle with square+r," in *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*, July 2012, pp. 565–570.

[4] J. Gennari and D. Garlan, "Measuring attack surface in software architecture (cmu-isr-11-121)," 2012.

[5] R. Vanciu and M. Abi-Antoun, "Finding architectural flaws using constraints," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2013, pp. 334–344.

[6] M. Almorsy, J. Grundy, and A. S. Ibrahim, "Automated software architecture security risk analysis using formalized signatures," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 662–671.

[7] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS 16. New York, NY, USA: Association for Computing Machinery, 2016, p. 254269.

[8] M. Chaieb, S. Yousfi, P. Lafourcade, and R. Robbana, "Verify-your-vote: A verifiable blockchain-based online voting protocol," in *Information Systems*, M. Themistocleous and P. Rupino da Cunha, Eds. Cham: Springer International Publishing, 2019, pp. 16–30.

[9] S. Bragagnolo, H. Rocha, M. Denker, and S. Ducasse, "Smartinspect: solidity smart contract inspector," in *2018 International Workshop on Blockchain Oriented Software Engineering*, March 2018, pp. 9–18.

[10] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara, "Security assurance for smart contract," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, Feb 2018, pp. 1–5.

[11] Á. Hajdu and D. Jovanovic, "solc-verify: A modular verifier for solidity smart contracts," *CoRR*, vol. abs/1907.04262, 2019. [Online]. Available: http://arxiv.org/abs/1907.04262

[12] X. Xu, I. Weber, and M. Staples, *Case Study: AgriDigital*. Cham: Springer International Publishing, 2019, pp. 239–255.

[13] X. Xu, C. Pautasso, L. Zhu, Q. Lu, and I. Weber, "A pattern collection for blockchain-based applications," in *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, ser. EuroPLoP 18. New York, NY, USA: Association for Computing Machinery, 2018.

[14] F. Chen, Z. Liu, Y. Long, Z. Liu, and N. Ding, "Secure scheme against compromised hash in proof-of-work blockchain," in *Network and System Security*, M. H. Au, S. M. Yiu, J. Li, X. Luo, C. Wang, A. Castiglione, and K. Kluczniak, Eds. Cham: Springer International Publishing, 2018, pp. 1–15.

[15] N. Chondamrongkul, J. Sun, and I. Warren, "Ontology-based software architectural pattern recognition and reasoning," in *30th International Conference on Software Engineering and Knowledge Engineering*, June 2018, pp. 25–34.

[16] N. Chondamrongkul, J. Sun, and I. Warren, "Pat approach to architecture behavioural verification," in *31th International Conference on Software Engineering and Knowledge Engineering*, July 2019, pp. 187–192.