5-2019

Security Protocol Analysis and Blockchains

Benjamin T. Temple
Trinity University, ben@benlinuxguy.com

Security Protocol Analysis and Blockchains

Benjamin Temple

A departmental senior thesis submitted to the Department of Computer Science at Trinity University in partial fulfillment of the requirements for graduation.

May 16, 2019

Seth Fogarty	Yu Zhang
Thesis Advisor	Department Chair

[illegible]

Michael Soto, AVPAA

Security Protocol Analysis and Blockchains

Benjamin Temple

May 2019

1 Introduction

Cryptography is essential to the modern-day notion of information security and has long been used to ensure the security and confidentiality of information being transmitted over an unsecured, potentially hostile medium. The use of cryptography aims to solve multiple different security goals. Security goals are commonly divided into secrecy, authentication, and integrity [Ins]. Secrecy, also called confidentiality, refers to the ability to obscure the contents of a message from those for whom the message was not intended, ensuring only authorized parties gain access to the message. Authentication ensures that two parties have the same information and can also be used to ascertain certainty as to the identity of the other party in a communication. A simple example of this is username and password authentication to a Web service. In order to ensure that only authorized users have access to the Web service, the Web service compares what the user has entered for username and password information to what has been stored in the database. Only if the two sets of information match does the authentication goal succeed. It is important to note that authentication goals need not only be between a user and a service, as many protocols require authentication between two non-user entities, such as between a client computer and a server. Integrity goals ensure that the data being transmitted has not been tampered with in any way. This can be accomplished by appending a cryptographic signature to the data being transmitted. A signature is generally comprised of a hash of the data being transmitted, which is then encrypted by the sender's private key. The receiver can then decrypt the hash and compute their own hash.

If the two hashes match, the integrity of the data can be assured.

Because cryptographic protocols must maintain these properties, or, at the very least, the property of secrecy, at all times, it is difficult to design a cryptographic protocol that fully satisfies these security goals without any exploits. While the goals of a cryptographic protocol are easy to understand, writing cryptographic protocols which satisfy and maintain these goals is surprisingly nontrivial. Many protocols have vulnerabilities in either their design or implementation that, in most cases, are not discovered until the protocol has been released and is in wide use. A famous example of a protocol broken in its design is the Needham-Schroder Public Key protocol, which was intended to be used to establish keys between parties but is vulnerable to a replay attack [God05].

One of the reasons that cryptographic protocols are so difficult to properly and securely develop is quite trivial - cryptographic protocols must be secure when an attacker is already present in the network in which the protocol is being run. All networks must be treated as hostile environments that have already been compromised. This idea forms the foundation of the Dolev-Yao model for a communications channel, which states that an intruder can have full control over any messages being sent over a network and can choose to read and/or modify any messages being sent or received [DY83]. Therefore, encryption and other protections against message modification must be put into place in order for a protocol that is to be run under a Dolev-Yao channel to remain secure.

As the secrecy, authentication, and integrity goals inherent in security protocols are key in ensuring their fitness for implementation, it is important to verify that the protocol does, in fact, satisfy these goals. This is where protocol verification tools become useful. These types of tools have been proposed as being a solution to determining the security of protocols by checking encoded models of these protocols against certain security goals. There are many different protocol analyzer utilities available, and protocol verification is of academic and commercial interest, so there are many different protocol analysis suites available, some free and some commercial, at various states of development, such as AVISPA [Avi06], ProVerif [Bla18b], and CryptoVerif [Bla18a].

This thesis will focus on the AVISPA [Avi06] and SPAN [Gen17] protocol verification tools. These tools take in protocol specifications in the CAS+ and HLPSL formats, discussed in Sections 3.1 and 3.2, and output whether or not a vulnerability can be found in the protocol. Additionally, SPAN generates message sequence charts that simulate the flow of the protocol, both with and without an intruder present. If an attack is found, SPAN can generate a message sequence chart which portrays where the attack can be found. This greatly improves the ability of one who is writing a security protocol to ensure that the protocol does not have any vulnerabilities innate to how messages are being passed.

The AVISPA and SPAN protocol verification tools are used to verify security protocols, and we wish to determine if these utilities can verify protocols which involve blockchains. A blockchain is a distributed and secure ledger which can only be modified by adding more information to the ledger, providing resistance against tampering and changes to currently-existing data [Pec17]. The distributed nature of the blockchain means that every computer which participates in the blockchain network contains a full copy of the blockchain, increasing fault tolerance per [Kou18]. It is this fault tolerance and tamper resistance that differentiate a blockchain ledger from a standard database or ledger that does not rely on blockchain.

The utility of blockchains for security-sensitive applications is dependent on the security of protocols involving blockchains. There are two classes of interesting protocol involving blockchain. First of all, there are protocols about blockchains, such as those protocols necessary to establish keys to communicate on a blockchain or add a block of information to the blockchain. Second, there are protocols that use blockchains as a platform. These protocols run on top of blockchains to accomplish certain tasks, such as leveraging a blockchain to store information as explained in [Kou18] or collecting votes anonymously as explained in [LW17]. Smart contracts, which are another blockchain technology, and will not be covered in depth for this thesis, can be used for role-based authentication [CKY17] or for other generic purposes which require preservation of privacy [KMS$^+$16]. These protocols use blockchains as a platform. When compared to more traditional forms of security protocol, security concerns for both kinds of protocols involving blockchain have increased complexity. As blockchains are publicly-readable data structures [Kou18], it is absolutely critical to ensure that protocols that run on top of blockchains are secure. This is because the results of the protocol's run are public on the blockchain and attacks on the protocol could be discovered long after the protocol's execution.

AVISPA and SPAN are designed to model security protocols with a fixed number of actors. It is unclear if these verification tools can be used to express and verify blockchain-based protocols. By design, blockchains have an arbitrary number of nodes which need to interconnect and communicate with one another. This peer-to-peer nature of the blockchain makes it difficult, although not impossible, to reduce protocols running on a blockchain to a set number of actors.

This thesis will determine if it is possible to utilize standard model-checking utilities and modeling languages to verify the security of blockchain-based security protocols, if there is a limited extent to which these protocols can be modeled and verified, or if the tools can be extended somehow to allow these types of protocols to be verified successfully and accurately. In this thesis, I investigate the viability of AVISPA [Avi06] and SPAN [Gen17] protocol verification tools in verifying protocols presented in two case studies: one example of both kinds of blockchain protocols. We consider a protocol used to establish compound blockchain identities in the context of a blockchain storage and access control system. While this protocol does not run on top of a blockchain, leveraging the blockchain as a platform, it is critical to establishing identities which can be used to improve security of protocols that do leverage the blockchain as a platform. We then examine an electronic voting (E-Voting) protocol that runs on top of a blockchain, leveraging properties of the blockchain to ensure security and ensure that the election is auditable. This auditability is provided by the fact that blockchains are a public record and, therefore, anyone can read the blockchain to determine how votes were recorded. Additionally, the tamper resistance of a blockchain as provided by Byzantine Fault Tolerance is key in ensuring that votes are not tampered with over the course of an election. These properties make a blockchain an ideal platform for an electronic voting protocol. Attempting to encode and verify these case studies is used to determine with reasonable certainty if protocol verification utilities such as AVISPA and SPAN can be used to verify the security of protocols involving blockchains.

For protocols about blockchains, such as the compound identity protocol used to establish identities used for transactions on a blockchain, we discover that it is possible to verify their security using AVISPA and SPAN and that vulnerabilities introduced into the protocol by manipulating the protocol specification are properly recognized. This is not surprising as the compound identity protocol does not run on a blockchain and is a variation on a standard security protocol of the type AVISPA and SPAN were designed to verify.

For protocols running on top of blockchains, however, the results of verification after encoding these protocols is not as clear. It is possible to implement these protocols in AVISPA three different ways: by encoding the blockchain as a Dolev-Yao channel, by encoding the blockchain as a write-protected channel, and by encoding the blockchain as an actor within the protocol. We implement the electronic voting protocol two ways: representing the blockchain as a channel and as an actor. We discover no significant differences in how the protocol is interpreted by AVISPA and SPAN whether the blockchain is represented as a channel or as an actor. However, we determine that AVISPA and SPAN do not appear to be able to detect vulnerabilities in protocols running on top of a blockchain, at the very least in the electronic voting protocol we tested as a case study. The SPAN tool does not generate message sequence charts that are proper to the protocol, only displaying one of the messages involved in the execution of the protocol when the blockchain is represented as an actor, and no messages whatsoever when the blockchain is represented as a channel. Additionally, when vulnerabilities or bugs are purposefully introduced into the protocol, the protocol is still interpreted as being safe by the model checkers in AVISPA. This may be an issue with the model checking backends included within the AVISPA and SPAN toolchains not being able to detect the types of vulnerabilities inherent in a blockchain protocol; however, this may also be an issue with the manner in which we encoded the blockchain protocol for verification by these tools.

Section 2 introduces blockchains and cryptographic protocols involving blockchains. Section 3 introduces the AVISPA and SPAN modeling tools we use for verification of protocols, as well as the protocol specification languages used by these tools. Sections 4 and 5 present our work, introducing the sample protocols, encoding these protocols, and discuss the efficacy of the verification toolchain in verifying these protocols. Section 6 concludes.

2 Blockchains

Before we investigate a protocol that runs on top of a blockchain, it is important to understand the basics behind how a blockchain operates and the implications of this design. According to Koutoupis, a blockchain is a collection of records of various transactions organized into blocks of transactions [Kou18]. These blocks are then hashed and each previous block is referenced by SHA-256 hash within the current block. Because of these hashes, it is not possible to modify a block in a blockchain without controlling a majority of the nodes in the blockchain network. This is because the block hashes must agree for the majority of the nodes in a blockchain in order for a transaction to be accepted. This idea that one more than 50% of the nodes in a blockchain network must agree in terms of their copies of the blockchain and any transaction performed is known as Byzantine Fault Tolerance [Kou18]. It is this property of blockchain networks that provides both the security advantages of blockchain, in the form of tamper resistance, as well as the advantage of resiliency against failure of one or more nodes in the network. This resiliency against failure is necessary because, as blockchains are on a peer-to-peer network, it is not possible to guarantee that all nodes in the network will be online and healthy at all times. Therefore, blockchain technology is suitable for applications which require a permanent resilient record recorded in a distributed database, such as applications regarding financial transactions, as in Bitcoin or similar cryptocurrencies or applications regarding authentication of users against a tamper-resistant database as in [CKY17]. As blockchain can be used in these types of sensitive applications in which the integrity of the data and tamper resistance is paramount, it is important to be able to verify the security of any protocols running on top of a blockchain.

2.1 Blockchain Identities

Cryptographic protocols are central to the concept and functionality of blockchains, as the blockchain itself takes advantage of cryptographic hashes for write protection and, additionally, many protocols that run on top of a blockchain have a security goal that necessitates the use of cryptography. We focus of a protocol used to create blockchain identities. A blockchain identity is a pseudo-anonymous data structure that contains the keys and identifiers necessary to communicate on a blockchain and ensure the secrecy and integrity of any sensitive data posted on the blockchain. Different blockchain standards have different standards for establishing a blockchain identity, and, additionally, blockchain identities can be different across different use cases for the same underlying blockchain. Each blockchain has its own way of identifying users, most commonly by an address used in the blockchain network that corresponds to a public key [Kou18]. Therefore, the identity can be represented as a tuple of the blockchain address, or public key of the user, and the user's private key. This identity is used to sign transactions placed on the blockchain and produce a permanent record as to who placed data on the blockchain. It is important to note, however, that because blockchains are public by their nature, data is often not encrypted by these blockchain identities, only signed to prevent tampering.

Zyskind [ZNP15] uses blockchains for private storage of information. In Section 4, we will study the first protocol presented in his work to enroll users in a compound identity, which lies at the heart of this application. This protocol allows for multiple users to read and write to the same data on a blockchain while still remaining somewhat anonymous. The use of these pseudo-anonymous identities presents one of the major advantages of blockchain in that the identities can preserve the privacy of the user leveraging the blockchain.

2.2 Blockchains As A Platform

Blockchains can be used as a platform to host various different types of applications, most commonly applications in the realms of cryptocurrency. However, there are many different types of applications that can be run on top of a blockchain Commonly, these non-cryptocurrency applications include the execution of smart contracts on a blockchain, which according to [Kou18] allow for the blockchain to evaluate conditions and execute code stored on the blockchain. This could then change whether the contract is executed or not, based on the conditions provided. Then, executed contracts would be stored on the blockchain, providing a tamper-resistant record of all executed contracts. These smart contracts can be used to enforce access control, such as in Jason Cruz et. al's paper entitled "RBAC-SC: Role-Based Access Control Using Smart Contract." This paper outlines a protocol by which smart contracts can be used to enforce access control for users of a system [CKY17]. This use of smart contracts is quite novel because their properties of immutability are desirable in a situation in which users are being authenticated to a secured system. It is desirable when designing an authentication scheme to ensure that users can only access what is permitted by their role and that the user cannot modify their access permissions. The permissions check algorithms presented in Cruz et al's paper provide for this and the nature of a blockchain network provides for the tamper resistance of the protocol.

The analysis of protocols which leverage smart contracts is beyond the scope of this work; however, there are many other protocols which leverage a blockchain as a platform. In Section 5, we investigate a protocol used for electronic voting that utilizes blockchain as a platform as found in [LW17]. This protocol will be used as a case study in an attempt to determine the effectiveness of the AVISPA and SPAN model checkers in verifying the security of protocols which leverage a blockchain.

2.3 Security Concerns for Protocols Running On Blockchains

Blockchain protocols share many of the same properties as security protocols which do not run on a blockchain. First and foremost, in both a traditional security protocol as well as a security protocol running on a blockchain, it is critical to ensure the secrecy of the data being passed over an insecure network using the protocol. As a blockchain is, by its nature, a distributed database with "public" access to any node present on the blockchain network, it is important to ensure that any data stored on a blockchain is either intended to be known by any node on the blockchain network or is adequately encrypted in addition to being hashed by the blockchain network. Additionally, depending on the type of protocol being discussed, authentication may be a goal, that is, determining if the actors in a specific run of the protocol are who they say they are. However, blockchain protocols in particular have certain security concerns that are not inherent to traditional security protocols. These include concerns regarding the establishment of keypairs used to communicate on a blockchain, as these keys must remain private. Also, as blockchain networks require that over 50% of the nodes in the blockchain network agree with one another as to the state of the blockchain, this opens up an interesting possibility known, somewhat inaccurately, as a 51% attack [Kou18]. If an attacker were to control 50% of the nodes plus one, the attacker would hold quorum as to the state of the blockchain network and could modify and data on the blockchain by simply changing any desired data and recomputing the hash used to ensure block integrity. It is important to note, however, that, under normal circumstances, data modified on a single node of the blockchain network would not be of consequence as the majority of the network would have the correct data, causing the incorrect data to be ignored. Therefore, while this type of attack is unlikely, this is something that is a theoretical concern when designing protocols built on top of a blockchain. This type of attack can likely be mitigated at a protocol level by adding additional integrity checking or private-key encrypted signatures to whatever is being stored on the blockchain. Then, even if an adversary has control of the blockchain and modified data and the native blockchain checksums, the modified data would still be recognized as invalid at a higher protocol level. This takes care of the integrity goals inherent in a security protocol running on a blockchain.

Due to the Byzantine Fault Tolerance afforded by a blockchain network as referenced by [Kou18], availability is not much a concern to a blockchain network as each node in the blockchain contains all the data present on the blockchain and the loss of one node would not cause any operational concerns for a blockchain network.

These sorts of security concerns are also evident in protocols which do run on top of a blockchain, such as electronic voting protocols. These protocols must be secure against attack while maintaining integrity and secrecy of the votes being cast.

3 Protocol Verification Tools

It is difficult to determine if a protocol is secure against attack by simply looking at the protocol, as vulnerabilities at the protocol level often involve very subtle differences in messages being passed, such as the lack of encryption on a single message, that can be easily missed by manually reviewing protocols. Therefore, it is important to find a way to verify the security of protocols in an automated fashion given the specification for a particular protocol. This automated verification of security protocols is possible by using a utility which serves as a model checker, which takes in a specification of a protocol encoded in a protocol specification language and statically analyzes the protocol, looking for the flow of messages in the protocol and determining if there is any vulnerability present in the protocol.

This section covers the tools, AVISPA and SPAN, which will be used over the course of this thesis to determine if blockchain-based security protocols can be adequately modeled using utilities intended for the verification of security protocols.

These tools take protocol specifications, which are descriptions of protocols including information on the roles or actors involved in the functioning of the protocol as well as the messages being passed between actors that make up the protocol. These specifications are written in high-level languages which are relatively simple to define, which allows these model checking utilities to be used by even those with little programming experience. These tools have been used to verify protocols in the areas of computer and network security, more specifically protocols between clients and servers on the Internet or on an internal network. Once the tools run on a specified protocol, they will produce either a response that the protocol is safe, that is, no attack has been found, or they will output a trace on whatever attack has been found. These tools both operate on the same backend protocol analyzers, as SPAN is built upon AVISPA and extends the functionality of AVISPA. and they can both be used to verify protocol in areas of computer security such as authentication and encryption. More concretely, these protocol analyzer utilities have been used to analyze and prove the security of TLS for Web encryption and Kerberos for authentication, among other related protocols in these arenas [Avi06]. Additionally, both AVISPA and SPAN accept protocol specifications in HLPSL (High-level Protocol Specification Language) [Pro06] and return the same type of attack trace if an attack is found. The specifics of the HLPSL protocol specification language will be discussed in Section 3.2. The protocols being encoded for verification within AVISPA and SPAN must be able to be represented as interactions between different actors, which can be users of the protocol or servers involved in the processing of the protocol.

The model checking backends included in the AVISPA toolchain are OFMC (On-the-Fly Model-

Checker) [MV09], Cl-AtSe (Constraint-Logic-based Attack Searcher) [Tur06], SATMC (SAT-based Model-Checker) [ACC16], and TA4SP (Tree Automata based on Automatic Approximations for the Analysis of Security Protocols) [BHK09]. These model checking backends each take slightly different approaches to searching for attacks. Therefore, some of these backends can theoretically find vulnerabilities in protocols that others do not and, conversely, it is theoretically possible for one of these backends to encounter a false positive in which the model checker backend identifies a vulnerability where there is none. However, it is more likely, especially in the case of a false positive, that said error was caused by an error in defining the protocol for the model checker as opposed to an error with the protocol itself. For this thesis, we will be focusing on the OFMC [MV09] model checker as it appears to be the most well-documented and mature of the model checkers included in the AVISPA toolchain. However, we have been able to utilize the Cl-AtSe [Tur06] model checker option in AVISPA and SPAN to verify the same protocols as verified by the OFMC model checker. In contrast to the successes in verifying protocols with the OFMC [MV09] and Cl-AtSe [Tur06] model checking backends, the SATMC [ACC16] and TA4SP [BHK09] model checking backends encountered errors when attempting to verify protocols we had defined. It is unclear as to if this is because the SATMC and TA4SP model checking backends can only verify specific types of protocols or if the protocols we verified were not in a proper syntax for these model checkers. The protocols which can be verified by OFMC but cannot be verified by the other model checkers are decidable, as there is an algorithm, namely the algorithm used by OFMC, that can verify the protocols. Therefore, this inability of SATMC and TA4SP to verify certain protocols which can be verified by OFMC indicates the issue is with the maturity of the non-OFMC model checkers in the toolchain, not an issue with the protocols being verified. Per our research and experimentation, we have determined that the OFMC model checker is the most complete of the model checkers in the AVISPA toolchain in that it does enforce goals properly, unlike Cl-AtSe, and does not throw meaningless errors as was our experience with SATMC and TA4SP.

The SPAN (Security Protocol Animator) tool builds on AVISPA by adding three additional functions which increase its utility: support for an additional file format in which protocols can be encoded, an integrated development environment to streamline writing of the protocol specifications, and a visualizer which can prove useful in debugging protocol specifications or simply determining the flow of messages in a protocol, both in normal operation and when a vulnerability is found. SPAN not only accepts files in the HLPSL [Pro06] file format accepted by AVISPA, it also accepts files formatted in the CAS+ [SG11] file format, which provides an easier-to-understand method in which protocols can be encoded. The CAS+ file format will be discussed further in Section 3.1. These CAS+ specifications are then translated on-the-fly into HLPSL files which can be interpreted by the model checking backends. The integrated development environment function within SPAN makes protocol verification more streamlined as a protocol can be defined in the built-in editor then immediately run through one of the model checking backends. The output of the protocol verification can then be analyzed in textual form or a visualization of the protocol can be launched as well as a visualization of any attack that is possible in the protocol. The visualizer allows for the generation of Message Sequence Charts which graphically represent each message in a protocol as it is sent and received, allowing for easier debugging of protocol specifications and better understanding of the functioning of any given protocol. These charts can provide a visual reference to the "normal" functioning of a protocol without an attacker and can also provide a visual interpretation regarding the protocol with an attacker or intruder present as an actor. Lastly, if an attack is found, the SPAN tool can visualize the steps required for the attack in terms of messages passed.

I present the CAS+ and HLPSL formats using a very simple protocol with a known vulnerability, in this case a man-in-the-middle attack, as a running example. This protocol is not related to blockchains, but is used to introduce the modeling languages and discuss the features and strengths of this toolchain. Section 3.1 encodes the protocol in CAS+, introduces goals for the protocol, and demonstrates SPAN's ability to detect vulnerability. Section 3.2 discusses HLPSL, the modeling language to which CAS+ files are translated which is the language used natively by the AVISPA toolchain.

3.1 CAS+

The CAS+ protocol specification language supported by SPAN is more concise, but potentially not as expressive as the HLPSL file format natively supported by AVISPA. However, it is also easier to understand and write than HLPSL files. These files are formatted in a slightly simpler fashion with more well-defined sections than HLPSL files. The files start with an identifiers section in which the different identifiers (users/agents, private keys, numbers, and other arbitrary objects needed for the functioning of the protocol) are defined. Then, the messages section is defined. This section includes the sequence of messages that make up the protocol specification. Then, a knowledge section including each user/agent followed by the other objects to which it should be privy are defined. The following sections define the sessions of the protocol that will be simulated, the knowledge of any intruder who may be in the system (likely a public key or any other public information in a well-designed protocol), and any goals, which, similarly to the HLPSL file format, are evaluated by the protocol verification backends of the AVISPA toolchain to ensure protocol security.

One disadvantage of the CAS+ format is that the goal section does not always get translated properly into the HLPSL format that is actually executed, so it may be necessary to add a goal section to the generated HLPSL file when utilizing a CAS+-defined protocol file. These goal sections are necessary for some of the protocol-analyzing backends to be able to generate a definitive result as to the safety or vulnerability of a specified protocol.

The section uses a basic protocol as a running example to demonstrate the capabilities of the AVISPA and SPAN tools. The protocol consists of one party sending a message to another party and the receiving party acknowledging this by sending a confirmation message back to the sending party. This was the simplest example of a peer-to-peer protocol type using public-key encryption I could formulate using CAS+ that is "interesting," that is, actually exchanged messages between multiple actors with some level of security. The protocol simulates message exchange between two actors with a message being passed from actor A to actor B and actor B then sending a confirmation message back to actor A once the message has been received. The full CAS+ specification for this

```
protocol basicProtocol;

identifiers
A, B : user;
Msg, Confirm : number;
Ka : public_key;

messages
1.A -> B : {Msg}Ka'
2.B -> A : {Confirm}Ka

knowledge
A: A, B, Ka;
B: A, B;

session_instances
[A:alice,B:bob,Ka:key]
[A:i,B:bob,Ka:key]
;

intruder_knowledge
alice, bob;

goal
secrecy_of Msg [A,B];
B authenticates A on Confirm;
```

Figure 1: The Basic Protocol as Represented in CAS+

protocol is listed in Figure 1.

Without a goal section, the basic protocol example provided in Figure 1 is of limited utility as, although some protocol analyzer backends of AVISPA, most notably the Cl-AtSe backend, can generate a result, others, most notably OFMC, cannot. This inability of the OFMC model checker to verify this protocol in its current state is directly related to the fact that the protocol does not have any security goals defined in this iteration. Therefore, as OFMC only supports the verification of protocols with goal sections, the basic protocol cannot be verified by OFMC until a goal section is added to the protocol. Additionally, results on a protocol without any goals do not make much sense semantically, as a protocol cannot be declared as being safe without knowing from which attacks the protocol should be safe. This could indicate that OFMC is a more thoroughly sufficient model checker for protocols, as it does enforce the explicit definition of security goals for protocols. This shortcoming of the protocol definition is easily overcome by adding such a goal section, as in the following, which was appended to the end of the CAS+ specification for the protocol, as indicated by the last three lines in Figure 1.

By simply adding these three lines, which specify that the security goals of this protocol are to maintain the confidentiality of the `Msg` as transmitted between roles A and B and that Bob should authenticate Alice on the Confirm message, the verification backends, most notably OFMC for the purposes of this thesis, which previously were unable to return a result are able to return a result. The authentication goal prevents replay and man-in-the-middle attacks from occurring. With all three goals in place, the protocol is identified as unsafe by the OFMC backend within AVISPA. However, without the authentication goal, the protocol is erroneously marked as safe by the OFMC model checking backend.

The vulnerability in the protocol requiring the authentication goal to detect is as follows. If Alice were to send Bob a message, Bob could send back any arbitrary Confirm message. Therefore, there is no authentication that Bob actually receives the message Alice sent, opening up the possibility of a man-in-the-middle attack in which an intruder intercepts a message and sends a confirmation without passing the message on to Bob. It is important to note, however, that the intruder having intercepted the message does not mean that the intruder can read the message, as it is encrypted with the private key to which the intruder has no access. The entire purpose of this attack is to intercept the communication between Alice and Bob in order to discard the message being sent to Bob and send a false confirmation back to Alice. This dissonance between Alice's expectation and reality has interesting implications in that, if the message were some sort of funds transfer message or other financial transaction, this type of attack could be used to steal money or otherwise leave a critical transaction unfinished. However, this vulnerability can be detected by the OFMC model checker within AVISPA due to the authentication goal placed within the CAS+ protocol specification file as indicated by the last line in Figure 1.

This additional authentication goal can also be represented in HLPSL as supported by both AVISPA directly as well as SPAN. The process to implement this will be discussed in Section 3.2.

With both secrecy and authentication goals defined for the basic protocol using CAS+, for our purposes, CAS+ is mostly sufficient to encode the protocols that will be defined in Section 4 and Section 5. Importantly, however, CAS+ does have certain limitations that force the use of HLPSL to fully define certain types of goals, namely a secrecy goal for a private key. Therefore, we begin by encoding protocols within CAS+ and utilizing SPAN to translate these protocols into HLPSL for protocol analysis. Then, we review the generated HLPSL code and make any necessary changes to resolve syntactic and semantic errors introduced either by limitations of the CAS+ language itself or errors in the translation from CAS+ to HLPSL for processing and verification of the protocol specification.

3.2 HLPSL

While CAS+ fully suffices to encode and verify the basic protocol presented in Section 3.1, that is not always the case. Thus this section presents HLPSL. The protocol created in Section 3.1 is run through the CAS+ to HLPSL converter provided by SPAN to generate an HLPSL specification for the same protocol.

The HLPSL file format consists of definitions of the various roles of which a protocol consists, most often Alice and Bob (the two users or actors between whom the protocol is being executed and the messages are supposed to be being passed), a session role which contains the various sessions for which the protocol will be simulated, and an environment in which the other roles are instantiated. As with CAS+, the HLPSL file format also provides for a goal section in which the overall secrecy and/or authentication goals of the protocol are defined. It is these goals that the AVISPA and/or SPAN tools utilize as a baseline to determine whether or not a protocol is secure. Put differently, the protocol is validated against the defined goals and any attacks (sequences of events which violate these goals) can be found.

The role blocks within HLPSL, which are identified by strings defining the name of the role, such as "alice" or "bob," begin with definitions of the various constants used by the role, such as public keys, hash functions, and state variables, among other references such as references to all other actors within the protocol. For example, an HLPSL protocol between two actors, Alice and

Bob, would reference agents A and B within both the Alice and Bob roles. Within each role, there exists a `played_by` directive to indicate which agent constant defined earlier is used by this role. By using this syntax, it is simple to define each role and ensure that roles can be aware of other roles. This awareness between roles serves the same purpose as the knowledge and `intruder_knowledge` sections of CAS+ files, which will be explored in the next section. The role definition also contains variables local to the role itself which would not be considered outside knowledge or accessible by any other roles present in the protocol definition. The "init" section of the role allows for the definition of initial values for local variables present in the role.

Within HLPSL, it is the case that roles are referenced by other roles in every protocol implemented with this specification language, forming a sort of hierarchy. As everything in HLPSL is represented as a role, role blocks cover both the agents, such as Alice and Bob, as well as other non-agent aspects of the protocol, including each session for which the protocol is run as well as the environment. The environment is an all-encompassing role in which the other roles, such as the agents and sessions, are instantiated. This hierarchical design of HLPSL has both advantages and disadvantages in that, because everything in the protocol is an object, the operations of the protocol can be represented as a composition of other objects. However, this "objects within objects" approach can be difficult to understand when reading or writing protocols by hand, hence why the visualizer and CAS+ translator functions within the SPAN utility prove useful.

It is surprising to note that the HLPSL conversion was done which preserving variable names that made sense based on the variable named present in the CAS+ file. This makes it easier to understand the generated HLPSL without needing to reverse engineer which variable name in the HLPSL corresponds to which variable in the original CAS+ file. As HLPSL is more complex to initially write, the translator from CAS+ to HLPSL made available by SPAN proves useful so a CAS+-defined protocol, which has limited capability in terms of defining security goals for protocols, can be translated into HLPSL. The protocol can then be modified to include a goal specification, which is necessary for some of the model checking backends within AVISPA and SPAN, such as the OFMC model checker, to be able to determine the security of a protocol.

The HLPSL protocol specification for this basic protocol we have defined starts with a definition

```
role role_alice(A:agent,B:agent,Ka:public_key,SND,RCV:channel(dy)) played_by A
def=
    local
        State:nat,Msg:text,Confirm:text
    init
        State := 0
    transition
        1. State=0 /\ RCV(start) =|> State':=1 /\ Msg':=new()
        /\ secret(Msg',sec_1,{A,B}) /\ SND({Msg'}_inv(Ka))
        2. State=1 /\ RCV({Confirm'}_Ka)
        =|> State':=2 /\ request(A,B,auth_2,Confirm')
end role
```

Figure 2: The Alice Role as Represented in HLPSL

of the roles for each agent in the protocol, namely `role_alice` for the Alice role and `role_bob` for the Bob role, see Figures 2 and 3. These roles include parameters, local variables, and state transition information. The parameters of these roles for each agent of the protocol are passed into the role and used in each transition statement. These parameters include `A` and `B`, the two agents performing these roles when the protocol is run. Each role must be referenced as a parameter to each other role so that each role is aware of every other agent in the protocol. A shared public key is passed in as `Ka`. This key is used to encrypt the message as well as the confirmation of the message in the protocol. Lastly, send and receive channels are provided, named `SND` and `RCV`. It is through these send and receive channels that all messages exchanged in the protocol are sent and received. The local variables include `State`, which encodes the current step in which the protocol currently resides, `Msg`, a text value which stores the message to be transmitted, and `Confirm`, a different text value transmitted by Bob after receiving the the `Msg` sent by Alice role. The `State` variable, which indicates the state in which the protocol currently resides, is local to each role, not global to the protocol. This would become more critical when examining protocols of more complexity in which the state transitions and the state in which the protocol currently resides may not be immediately obvious. Because of this state variable, it becomes immediately obvious when looking at a transition where in the protocol sequence the change in state related to the transition occurs. This makes the protocols relatively simple to both write and understand.

The transition block in each role specification is composed of a series of statements. Each statement describes a single step in the execution of the protocol. Transitions take the form of `preconditions =|> post-conditions`. Each statement usually receives a message in the precondition, and sends a response in the post-condition. Other preconditions might include the contents of of the message and the value of local variables such as `State`. As an example of this, transition 1 of the `alice` role within the basic protocol starts with the `State` variable at zero. Then, a special `start` message is received through the `RCV` channel. This message signifies the start of a single run of the protocol, as implied by the name of the message. Once this start message is received, the postconditions of this transition begin. In the case of this first transition within the `alice` role, these postconditions involve setting the `State` to 1, generating a new value for the `Msg` and sending the `Msg` over the Dolev-Yao channel named `SND` encrypted by the private key which corresponds with `Ka`, which, in the case of HLPSL, is represented by `inv(Ka)`. The `SND()` and `RCV()` functions present in each transition statement allow for the sending and receiving of any specified message respectively, optionally encrypted by a key previously defined. These functions are implicitly defined with the channel definitions present in each agent role.

Variables within HLPSL are primed whenever a new value is being sent or received, such as the newly-generated message and confirm text variables in the case of this basic protocol. In general, primed variables are sent in the postcondition of a transition and received in the precondition of a transition. The `new()` keyword within HLPSL signifies the initialization of a variable with a arbitrary new value, and, as such, the first use of a variable after `new()` in both the send and receive case should be primed. The contents of a variable to be sent can optionally be encrypted by appending an `_` character followed by the name of a key to the send statement, as follows: `SND{Variable'}_Key`, as each variable name referenced by a send or receive is enclosed in `{}`. If encryption using a public key's corresponding private key is desired, the syntax is as follows: `SND{Variable'}_inv(Key)`. The `inv` keyword within HLPSL signifies that a message is to be encrypted or decrypted using the private key associated with a public/private keypair. In addition to the previous constructs within HLPSL that are present within the `alice` role as depicted in Figure 2, such as the send and receive functions, transitions, and local variables, the `bob` role

```
role role_bob(A:agent,B:agent,Ka:public_key,SND,RCV:channel(dy)) played_by B
def=
    local
        State:nat,Msg:text,Confirm:text
    init
        State := 0
    transition
        1. State=0 /\ RCV({Msg'}_inv(Ka))
        =|> State':=1 /\ secret(Msg',sec_1,{A,B})
        /\ Confirm':=new() /\ witness(B,A,auth_2,Confirm')
        /\ SND({Confirm'}_Ka)
end role
```

Figure 3: The Bob Role as Represented in HLPSL

depicted in Figure 3 also provides an instantiation of the `secret()` construct, which is critical in establishing secrecy goals and allowing the model checker to verify secrecy of a message between two actors in a protocol. The `secret()` construct is formulated as follows. The first parameter within the `secret` block indicated that `Msg'`, which indicates the newly-assigned value of the message being sent between actors, should be a secret. The second parameter, `sec_1`, provides a name, known as a `protocol_id` to this secret. This `sec_1` constant will be used later in the goal section within the HLPSL file. Finally, the comma-separated list of actors within brackets, `{A,B}` indicates that this piece of data, the `Msg`, should remain private between actors A and B, or Alice and Bob.

The transitions between two roles are symmetric. Each message sent in one transition is received by the other role in another transition. This becomes evident when looking at the transition sections for both roles in the example protocol provided. For each `SND` operation in Role A, there is a corresponding `RCV` operation in Role B and vice versa. More concretely, in transition 1 of the Alice role, the Start sentinel message is received, indicating the start of the protocol with the state initially at state zero in the precondition for the transition. Then, in the postcondition for the transition, the state transitions to 1 and a new message is generated and sent over the `SND` channel, encrypted by the private key associated with `Ka`. In the second transition, the precondition indicates that the state is currently set to 1 and a confirm message encrypted by the public key `Ka` is received. Then, in the postcondition, the state transitions to 2, indicating the end of this particular protocol. When

```
role session(Ka:public_key,A:agent,B:agent)
def=
    local
        SND2,RCV2,SND1,RCV1:channel(dy)
    composition
        role_bob(A,B,SND2,RCV2) /\ role_alice(A,B,Ka,SND1,RCV1)
end role
```

Figure 4: A Session as Represented in HLPSL

the state variable in the postcondition matches the number of transitions present in the protocol, the protocol has completed and there is no further work specified by the protocol.

HLPSL uses a session role to represent the instantiation and run of the protocol.. An example of a session is in Figure 4. The local variables for each session include the send and receive channels in which messages are sent, one set for each agent which participates in the protocol, as well as the overall `channel(dy)` declaration, indicating that this protocol is to be run with the Dolev-Yao model dictating the rules of the protocol. These local variables which denote send and receive channels in each session are passed in to each agent role within the composition section of the session role. Each role being instantiated within the session has its own set of send and receive channels passed in, which, in this case, are `SND1` and `RCV1` for `role_alice` and `SND2` and `RCV2` for `role_bob`. The session, in its header within the role declaration, also references each agent, in this case `A` and `B` and any other parameters that are passed in for the protocol, in this case, only a keypair known as `Ka`. These parameters which are shared between sessions as they are instantiated, are passed in at the declaration of each session role, similar to the instantiation of a class in an object-oriented programming language. This is in contrast to the other variables which are local to each agent, such as their state, the `Msg` variable, and the `Confirm` variable. These variables are not referenced in the session at all. Communication channels are created locally to the session, and are not connected by the specification. Each role has its own send and receive channels which are interconnected by the environment in which the roles are instantiated.

After the sessions are defined as roles, one more role is defined - the environment: see Figure 5. Just as the session is a composition of agents with their requisite parameters, the environment is

```
role environment()
def=
    const
        key:public_key,hash_0:hash_func,
        alice:agent,bob:agent,sec_1:protocol_id,auth_2:protocol_id
    intruder_knowledge = {alice,bob}
    composition
        session(key,alice,bob) /\ session(key,alice,bob)
end role
```

Figure 5: The Environment Role as Represented in HLPSL

a composition of sessions with their requisite parameters. Because of this, the environment is at the top of the hierarchy of the protocol simulation, as the sessions for the protocol simulation are defined in the environment, and the roles for the protocol simulation are defined in the session. The environment also ties together the arbitrarily-named send and receive channels for each role as defined in the session used to represent an actor in the protocol. While these are different channels through which the messages are sent, the channels are aggregated semantically within the environment so the protocol behaves as expected, with one actor in the protocol able to communicate with the other and vice versa.

The representation of a goal within HLPSL is quite similar to the representation of a goal within CAS+ as evidenced by the following goal section which was added to the HLPSL version of the basic protocol once the protocol was translated from CAS+ with a goal section in place:

```
goal
    secrecy_of sec_1
    authentication_on auth_2
end goal
```

It may not be clear immediately to which message the `sec_1` constant is referring or the roles to which the message should be secret. By looking at the HLPSL protocol specification, it only becomes clear that `sec_1` refers to a "protocol ID," therefore, it is clear that the goal has something to do with the secrecy of the protocol, which, in this case, refers to the secrecy of the messages passed between the two roles in this protocol, alice and bob. Based only on this secrecy goal, the protocol can be marked as safe because the `Msg` is encrypted and cannot be read by any party who may be able to intercept the message, as the intercepting party is not privy to the private key necessary to decrypt the message as it has been sent. In fact, each goal specified in HLPSL may have its own unique protocol identifier, as different goals require different types of protocol specifications. As an example, an authentication goal intended for anti-replay protection would have a different protocol identifier defined than a secrecy goal. In this way, the protocol identifier construct in HLPSL serves as a sort of constant which defines each instance of a security goal for a protocol, for example `sec_1` or `auth_2`. The goal section in HLPSL is located as the second to last portion of the HLPSL protocol specification, right before the final line which instantiates the environment. This goal section can include goals for secrecy, such as the `secrecy_of sec_1` goal noted in the goal section for this basic protocol. However, it can also include authentication goals in the following format `authentication_on auth_2` where `auth_2` is a protocol identifier. This allows for the model checker to verify goals not only of secrecy but also of the integrity of the protocol and any messages passed.

The vulnerability the basic protocol within the HLPSL protocol specification can be recognized by AVISPA by simply revising the goal section to include an authentication goal, making a change to the environment to add the `auth_2` protocol ID constant, and making changes to the transition statements in the `alice` and `bob` roles to add a `request()` directive to the postcondition of the transition on the `alice` role and a `witness()` directive to the postcondition of the transition on the `bob` role. The `request()` indicates that the variable in question is to be used to authenticate the roles indicated in such a request. The `witness()` directive in the opposite role indicates that the `bob` role is the role performing the actual authentication by witnessing the passing of the proper `Confirm` message to the `alice` role.

3.3 Comparison of CAS+ and HLPSL

CAS+ files do not appear to be as expressive as HLPSL files, as HLPSL files allow for roles to be defined with certain parameters, similar to how classes and objects work in object-oriented programming languages, whereas CAS+ files are written in a more imperative sense with clearly-defined sections defining sessions, actors, knowledge, and everything else required to simulate the protocol. The advantages of HLPSL files are that, because everything including the actors and the environment are defined as roles, this can increase flexibility in how a protocol is written and it may be easier in some cases to write protocols in this syntax. However, for the purposes of most, if not all, protocols, these two file formats appear to be equivalent and equally viable for use. Therefore, for the purposes of this thesis, CAS+ files will be used to describe protocol specifications unless use of HLPSL provides a tangible benefit. However, as CAS+ does not fully support the recognition of goal sections, specifically some authentication goals [SG11], through its translation into HLPSL, it is critical to open the generated HLPSL file after compiling the CAS+ file into HLPSL and add or modify a goal section if the goal section is not able to be properly translated.

4 Verifying a Protocol Involving Blockchain Identities

When working with a protocol, whether or not it runs on a blockchain, but especially when running on a pseudo-anonymous blockchain network, it is critical to be able to establish identities used within the communication of the protocol. This is especially important on a blockchain network because users on some blockchain networks may not have any natural identifiers, such as IP addresses, that can be traced back to the user and used for identification purposes. Zyskind [ZNP15] presents protocols to generate a compound identity, check a user's permissions against the permissions stored for the user in the blockchain network, make an access control decision, and store or retrieve data from the blockchain network. As an example of one of these protocols, the protocol presented to generate a compound identity operates as described in the next paragraph. The purpose of a compound identity is to provide an identity which is able to be shared among multiple parties [ZNP15]. One party would "own" the compound identity and essentially serve as its administrator, and this administrator or owner of the compound identity can delegate access to it by defining other users with access, known as guests [ZNP15]. The compound identity is composed of keypairs for each user who has access to the compound identity as well as a symmetric key used to encrypt and decrypt data that should be shared among the users of the compound identity. Because of this sort of double encryption, it is theoretically possible for users to be added to or removed from the compound identity at any time without compromising security. Therefore, desired properties for the compound identity include secrecy of the data protected by the compound identity from actors who are not members of the compound identity. There does not appear to be any sort of authentication goal inherent in this compound identity protocol. In this work, we focus on the protocol used to generate the compound identity, which, while not specifically a protocol that runs on a blockchain, is still instrumental in generating the keys necessary to securely interact with a blockchain in the context of other protocols that may run on top of a blockchain. Put differently, the compound identity protocol allows for access control on the blockchain for the blockchain protocols presented in Zyskind's paper. Additionally, this protocol could conceivably be used to generate identities for other blockchain protocols not presented in Zyskind's paper, as the blockchain protocol would simply need to perform its encryption using a symmetric key. The access control and management of keypairs for each authorized user is performed entirely within the compound identity protocol, so protocols leveraging a compound identity only need to be aware of and utilize the symmetric key which was generated by the compound identity protocol. In this work, we focus on the protocol as presented in [ZNP15] to generate a compound identity.

To generate a compound identity, according to [ZNP15], a secure channel is first generated between the two actors (in this case, a user and a server) which will be sharing said compound identity. This can be done by AVISPA and SPAN easily, by either utilizing a secure channel in CAS+ with the `=>` operator or encrypting all messages passed between actors with an additional `Ksecure` symmetric key which is known to both the user and the server. This compiles down to HLPSL as channel specifications passed into each role in the case of a standard Dolev-Yao channel defined within CAS+, or channel specifications in addition to pre-generated symmetric keys in the case of a secure channel defined within CAS+. If a standard channel is used, it is critical to ensure that all messages passed between the user and the server and encrypted by a symmetric key to secure the channel against eavesdropping and modification by an attacker, ensuring security of the protocol. Without this in place, the protocol would be verified as unsafe by the AVISPA model checkers, such as OFMC. To began the actual execution of the protocol, after the channels are set up, the user generates a public/private keypair as well as a symmetric key and sends the public key and symmetric key to the server. In return, the server generates a public/private keypair and a symmetric key and sends its public key and symmetric key back to the user. Therefore, after one run of the protocol, both the user and the server have each other's public key as well the symmetric key shared between them. It is important to note that the symmetric key and user's public key are sent in one message in the original compound identity protocol; however, due to limitations within AVISPA and SPAN, we have represented this as two messages. This protocol is used to enroll new users in a compound identity, so multiple users can submit the same symmetric key with their own public key to enable the "compound" portion of the compound identity, enabling the multiple-user access benefits afforded by this compound identity protocol.

Figure 6 is a representation of the CAS+ version of the compound identity protocol.

This protocol is relatively simple in that it only involves the transmission of public and symmetric keys between the user obtaining a compound identity and the server and the return of a public key generated by the server back to the user. This is all encrypted by a `Ksecure` key to enable the channel through which these keys are passed to be secure.

Based on experimental evidence in attempting to add a goal section to this initial revision of the compound identity protocol of either a secrecy or authentication goal type, it does not appear to be possible to add a secrecy or authentication goal if the keypairs are included in the knowledge section of the protocol specification. This results in the AVISPA application assuming that these keys were pre-generated as opposed to being generated as a part of the run of the protocol. This pre-generation of keys would defeat the purpose of secrecy goals, therefore, it is critical to ensure the public/private keypairs and symmetric keys do not appear in the knowledge section of the protocol specification.

Goals can be defined by simply removing the keys, in this case `Ksu`, `Kus`, and `Ksymmetric` from the knowledge section of the CAS+ specification, indicating to the model checkers that these keys are to be generated at the time of the protocol's execution. Then, two secrecy goals, one for the secrecy of `Kus` between U and S and one for the secrecy of `Ksu` between S and U can be defined.

```
protocol compoundIdentity;

identifiers
U, S : user;
Kus, Ksu : public_key;
Ksymmetric : symmetric_key;
Ksecure : symmetric_key;

messages
1.U -> S : {Kus}Ksecure
2.U -> S : {Ksymmetric}Ksecure
3.S -> U : {Ksu}Ksecure

knowledge
S: S, U,Ksecure;
U: S, U,Ksecure;

session_instances
[S:s,U:u,Ksecure:key];

intruder_knowledge
s, u;

goal
secrecy_of Ksu [S, U];
secrecy_of Kus [U, S];
secrecy_of Ksymmetric  [U, S];
```

Figure 6: A representation of the Compound Identity protocol as represented in CAS+

This goal setting caused an interesting result in that the protocol is shown to be insecure even after a single session with these goals in place. This would indicate that these public/private keypairs which are not supposed to be shared between S and U are shared between S and U.

To step through the CAS+ encoding of the compound identity protocol as represented in Figure 6, the specification begins with the identifiers section of the CAS+ specification file. This includes the user and server roles U and S, which are declared to be identifiers of type user. While the type declaration in CAS+ is of type user, at this point we are declaring the user and server to be actors in the protocol. CAS+, somewhat confusingly, uses the terminology of "user" rather than the terminology of "actor." This is misleading because an actor in a protocol can be a user, a server, or an intermediary. Then, the `Ksu` and `Kus` keys are declared as `public_key` datatypes, which will instantiate the public/private keypairs to be negotiated between the user and user, in the case of `Kus`, and vice versa, in the case of `Ksu`. The symmetric key that forms part of the compound identity, known as `Ksymmetric` as well as the symmetric key that simply exists to enforce a secure channel, `Ksecure`, are both also defined in this section.

Then, the messages section of the CAS+ protocol specification is defined. This section is relatively straightforward with the user sending the server the `Kus` public key and the symmetric key in two separate messages. Both of these messages are encrypted by the `Ksecure` private key to enforce that these messages are passed over a secure channel. Then. the server responds back to the user with the `Ksu` public key, also encrypted with `Ksecure`, which forms the final portion of the compound identity. It is important to note that, instead of explicitly defining a `Ksecure` symmetric key and using it to encrypt all communications between the user and server explicitly to create a secure channel, each transition could also have been written similar to the following: `1.U => S : Kus`. This use of the `=>` operator in CAS+ instead of the `->` operator creates a secure read-and-write protected channel implicitly, through which messages can be securely passed.

After the messages are defined, a knowledge section is defined. In the case of the compound identity protocol, both the user and the server have knowledge of themselves and each other, as well as the key necessary to generate a secure channel, prior to the beginning of the protocol. The `Kus`,`Ksu`, and `Ksymmetric` keys, which are generated over the course of the protocol's execution, are not included in the knowledge section. This is because the knowledge section should only contain pre-existing knowledge of the parties involved in the protocol prior to the execution of the protocol.

The `session_instances` section of the protocol specification only contains a single session instance for the sake of simplicity. This session instance simply defines that `S`, `U`, and `Ksecure`, which will be defined as `s`, `u`, and `key` respectively for purposes of message sequence diagram generation, will be pre-generated as a part of the protocol session being run.

The `intruder_knowledge` section of the protocol specification is also quite self-explanatory in that this defines what an intruder would have knowledge of prior to the protocol being run in the previously defined session instance. This is also defined using the alternative names for each entity in the protocol that were defined earlier in session instances, such as `u` instead of `U`. In this case, the intruder only knows about the user and the server prior to the start of the protocol, and, as the protocol is verified as safe by AVISPA and SPAN, it can be said with reasonable certainty that the intruder does not gain any further knowledge of any portions of the compound identity that is generated over the course of the protocol being run.

The goal section of the CAS+ file for the compound identity protocol is also self-explanatory, as it contains the secrecy goals needed for the protocol. The goal section as defined in CAS+ only enforces secrecy of the public keys between actors as well as secrecy of the symmetric key used for the compound identity. When the version of the compound identity protocol encoded in CAS+ is run through the OFMC model checker option in SPAN, the protocol is verified to be safe with no attacks found.

However, importantly, with the compound identity protocol, there are some limitations that exist within CAS+ which force the generated HLPSL code to be modified manually in order to accurately represent all of the security goals of the protocol, as both the public and private keys need to be ensured to be secret. In the case of this protocol, the limitations of CAS+ surround the inability to place a secrecy goal on a private key, which is defined as the primed counterpart of a public key in CAS+ or the inverse of a public key in HLPSL. Therefore, this protocol was started in CAS+ as per Figure 6 but the generated HLPSL was modified to include secrecy goals for the private keys in addition to the `Kus` and `Ksu` public keys and `Ksymmetric` symmetric key.

To generate this corrected HLPSL version of the compound identity protocol, the corrected CAS+ code is first compiled down to HLPSL, then the inverses of the public keys, which represent private keys, are manually added as secrecy goals by adding the necessary secret predicates to the HLPSL roles and adding the added secrecy `protocol_id` constants to the goal section for the protocol. This increases the number of secrecy goals from three to five. Whether running the protocol with only the public and symmetric keys as secrecy goals or running the protocol with the public, private, and symmetric keys as secrecy goals, the protocol is verified as safe by AVISPA using the OFMC backend. Additionally, when a vulnerability, such as a purposeful leak of key information outside of a secure channel, is introduced, the protocol is correctly identified as unsafe. This strongly indicates that the AVISPA toolset can properly verify the compound identity protocol.

The use of a symmetric key within the compound identity protocol which is associated with the user as opposed to using the public/private keypair directly for encryption is key to this notion of a compound identity, as this allows for multiple people to share use of the same symmetric key, if the key is encrypted separately with each authorized user's public key. Then, each user could decrypt the symmetric key with his or her private key, and this enables the "compound" portion of the compound identity protocol. Then, once the user-to-server keypair and symmetric key are submitted to the server by the user, the server responds with a server-to-user public key and the server retains control of its associated private key.

We will now step through the HLPSL specification of the compound identity protocol, presented in Figure 7, paying special attention to the differences in semantics between it and the CAS+ version of the same protocol. These differences in semantics are a direct result of the addition of additional secrecy goals for the private keys involved in the protocol, otherwise known as the inverses of the public keys being transmitted.

The user role, known as `role_U` in the HLPSL specification, is a fairly standard HLPSL role with creation of new public/private keypairs and symmetric keys using the `new()` operator, the sending of these keys, and establishment of the necessary secrecy goals for the protocol. It is important to note that the secrecy goals are defined by the `secret()` predicate within HLPSL and take in the key being marked as secret, an arbitrary protocol ID referenced later in the session and goals, as well as the entities to which the variable should remain a secret, in this case U and S for all secrets except for the private keys, which only have the entity that initially generated the key, U or S, listed under the parties to whom the key should remain a secret. This ensures that the private key remains private. Even though the private key is never transmitted over the channel between the user and the server, this goal is included for the sake of completeness in terms of the necessary security goals for any protocol involving private keys. Whether or not the private keys are included as secrecy goals. the compound identity protocol is verified as safe by the OFMC model checking backend within AVISPA and SPAN. However, without the private keys included as security goals, it is not possible to detect any vulnerabilities that may exist in the protocol if the protocol is implemented poorly and private keys are leaked. Therefore, we have included the private keys as security goals to ensure all possible vulnerabilities, whether intentional or unintentional, with this protocol can be properly identified by the model checkers.

The server role is known as `role_S` in the HLPSL specification. As with the user role, this role is a fairly standard role for an actor as represented in HLPSL and presents what is essentially the inverse of the user role. That is, where the user role sends a message, the server role receives the message, and vice versa. The secrecy declarations are also the same among both the user and server roles, as, in order for the HLPSL specification to be coherent, the secrecy declarations must be identical between all actors in the protocol, which, in the case of this compound identity protocol, consist of only the user and server roles.

The session role within the protocol specification does not encode any of the transitions for the protocol, with messages being sent or received; however, it does serve as a composition of the roles necessary to complete one run of the protocol. In this case, the session role takes in the agents or actors within the protocol as well as the Ksecure symmetric key which is pre-generated prior to the run of the protocol. Then, it establishes local variables for send and receive channels used for the run of the protocol. Finally, the session is defined as the composition between `role_U` and `role_S`.

The environment role for the protocol is the highest-level role in the HLPSL protocol specification which encapsulates the session for however many runs of the session are being performed in the simulation. In this case, the session is only being run once. Additionally, all the agents, the symmetric key, and the `protocol_id` constants used for the secrecy goals are established within the environment. It is also here within the environment role that the intruder's knowledge is defined as being only the server and the user.

The goal section for the HLPSL encoding of the compound identity protocol is the simplest of all the sections within the HLPSL file in that it only instantiates `secrecy_of` goals for all five of the `protocol_id` constants `sec_1` through `sec_5` that were defined earlier in the environment and defined to be secrets in the roles for the user and the server.

The HLPSL specification for this compound identity protocol represents all the necessary goals for the compound identity protocol, including the goals related to secrecy of private keys. All goals for this protocol are fully satisfied by the protocol, as the AVISPA and SPAN model checkers are able to verify that this protocol is safe with the OFMC model checker. Also, the message sequence chart generated by SPAN includes all messages sent or received over the course of this protocol's execution, which lends credibility to the verdict rendered by the model checking backend.

5 Verifying a Protocol Hosted On A Blockchain

After verifying a basic protocol as well as a protocol about blockchain identities, namely Zyskind's compound identity protocol [ZNP15], we will verify a protocol which leverages a blockchain as a platform. While it is difficult to verify the entirety of a protocol hosted on a blockchain, because a blockchain network has an indeterminate number of nodes, it is possible to verify a blockchain protocol if the indeterminate number of nodes in a blockchain network are somehow abstracted out. This difficulty in verifying blockchain-hosted protocols using AVISPA and SPAN arises because blockchain-hosted protocols are inherently many-to-many in that many nodes communicate with many other nodes within the blockchain network. This is at odds with the mechanics of AVISPA and SPAN which are inherently one-to-one with one message being passed from one actor to another actor. This limitation is due to the limitations in the semantics of HLPSL and CAS+, in which a message can only pass from one actor in a protocol to another actor in a protocol and all actors must be known and defined in advance within the protocol specification. Transitions can only occur

```
role role_U(S:agent,U:agent,Ksecure:symmetric_key,SND,RCV:channel(dy))
played_by U
def=
local
        State:nat,Kus:public_key,Ksymmetric:symmetric_key,Ksu:public_key
    init
        State := 0
    transition
        1. State=0 /\ RCV(start) =|> State':=1 /\ Kus':=new() /\ secret(Kus',sec_2,{U,S})
        /\ secret(inv(Kus'),sec_4,{U})/\ SND({Kus'}_Ksecure) /\ Ksymmetric':=new()
        /\ secret(Ksymmetric',sec_3,{U,S}) /\ SND({Ksymmetric'}_Ksecure)
        3. State=1 /\ RCV({Ksu'}_Ksecure) =|> State':=2 /\ secret(Ksu',sec_1,{S,U})
        /\ secret(inv(Ksu'),sec_5,{S})
end role
role role_S(S:agent,U:agent,Ksecure:symmetric_key,SND,RCV:channel(dy))
played_by S
def=
    local
        State:nat,Kus:public_key,Ksymmetric:symmetric_key,Ksu:public_key
    init
        State := 0
    transition
        1. State=0 /\ RCV({Kus'}_Ksecure) =|> State':=1 /\ secret(Kus',sec_2,{U,S})
        /\ secret(inv(Kus'),sec_4,{U})
        2. State=1 /\ RCV({Ksymmetric'}_Ksecure) =|> State':=2 /\ secret(Ksymmetric',sec_3,{U,S})
        /\ Ksu':=new() /\ secret(Ksu',sec_1,{S,U}) /\ secret(inv(Ksu'),sec_5,{S})
        /\ SND({Ksu'}_Ksecure)
end role
role session1(S:agent,U:agent,Ksecure:symmetric_key)
def=
    local
        SND2,RCV2,SND1,RCV1:channel(dy)
    composition
        role_U(S,U,Ksecure,SND2,RCV2) /\ role_S(S,U,Ksecure,SND1,RCV1)
end role
role environment()
def=
    const
        hash_0:hash_func,u:agent,s:agent,key:symmetric_key,sec_1:protocol_id,sec_2:protocol_id,
        sec_3:protocol_id,sec_4:protocol_id,sec_5:protocol_id
    intruder_knowledge = {s,u}
    composition
        session1(s,u,key)
end role
goal
    secrecy_of sec_1
    secrecy_of sec_2
    secrecy_of sec_3
    secrecy_of sec_4
    secrecy_of sec_5
end goal
environment()
```

Figure 7: The Compound Identity protocol as represented in HLPSL

in the form of "actor A sends message to/receives message from actor B", so, unless a protocol has a defined number of actors, it is not possible to define the protocol for verification using the semantics provided by the AVISPA and SPAN protocol verification tools without any abstraction.

However, if these blockchain protocols can be simplified into interactions between a fixed number of actors, it should be possible to define these protocols in HLPSL and CAS+ for verification using AVISPA and SPAN. This can be done in a number of ways. The simplest way to reduce this indeterminate number of actors to a single or otherwise determinate number of actors is to treat the blockchain network as a whole to be one of the actors in the protocol. This would allow for a protocol to be represented as interactions between a user on a blockchain network and the network itself. Additionally, the blockchain can be treated as a sort of "black box" through which messages transparently pass without the blockchain being explicitly referenced. This could work to represent protocols in which a user stores or retrieves information from a blockchain network. This treatment of the blockchain network as a black box would be most easily done by treating the blockchain as a channel, which will be discussed in Section 5.1. The treatment of the blockchain as a channel fits for this treatment of the blockchain as a black box because the channels in AVISPA and SPAN are implicit and all messages pass through a channel. For more complex protocols in which the nodes within the blockchain network need to communicate, it would be necessary to restrict the number of nodes being looked at for each run of the protocol and simply pass messages between the nodes which will be represented in the protocol. Messages could then be relayed between actors by passing messages from actor A to actor B, who then passes the message to actor C and so forth. This does introduce an aspect of complexity, however, because it could be difficult to determine which nodes to include and which to exclude when writing up the protocol specification. Therefore, this option would not fully be able to represent a blockchain and this could affect the accuracy of any protocol definition created using this methodology, potentially causing a false result as to the security of the protocol in question. Because of these inherent limitations to this method of relaying messages between an arbitrary number of nodes in a blockchain network, this method of encoding the protocol will not be explored in this work.

As an alternative to the straight peer-to-peer networking that would require the number of nodes in a blockchain to be either known or restricted or the complete abstraction of the blockchain inherent in treating the blockchain as a channel through which messages are passed, it is also possible to model the blockchain as an actor in the protocol specification, as described in Section 5.2. This provides the advantage of not assuming that the blockchain acts as a channel in all cases while potentially avoiding the disadvantages inherent in a purely peer-to-peer passing and relaying of messages.

It appears to be the case that blockchain protocols, especially those which deal with initialization of a user onto the blockchain network, can be simplified into interactions between the user and the network as a whole. By treating the blockchain network as a single entity, it is possible to avoid the inherent limitations of AVISPA and SPAN when describing protocols with a variable number of actors as it becomes possible to reduce the variable number of actors to only two types of actor - the users between which the protocol is being run and the network. This reduces a potentially indeterminate number of actors to a number of actors which is finite and tractable when encoding the protocol. For protocols in which the primary goal is communication between users with the blockchain acting as a channel through which messages are sent, the blockchain itself can be abstracted out as well, allowing the protocol to be in a more standard format that can be verified by AVISPA.

5.1 Modeling the Blockchain as a Channel

The simplest way this issue with AVISPA and SPAN not allowing for the many-to-many message passing necessary for blockchain protocols with an indeterminate number of nodes is to treat the blockchain as a channel, ignoring the many-to-many mesh network properties of a blockchain network. This could possibly work with blockchain protocols intended for security verification, as AVISPA and SPAN have native support for channels in which messages can be sent between two actors. The only type of channel supported by AVISPA and SPAN within HLPSL, is the Dolev-Yao (DY) channel, in which it can be assumed that an intruder can examine and/or modify any messages being passed within the channel [DY83]. However, it is important to note that the CAS+ modeling language supported by SPAN supports the simulation of different channel types, including a fully secure and a write-protected channel. This is accomplished as the CAS+ to HLPSL translator inserts encryption keys and hash functions into the HLPSL code that is generates, ensuring the security goals of a fully secure or write-protected channel are met. If a Dolev-Yao channel is used, the only security for the protocol is provided by cryptographic keys for signing and encryption of messages that are placed manually into the CAS+ or HLPSL specifications. While it is difficult to determine which type of channel should be used to represent a blockchain, this can be easily narrowed down to an unsecured Dolev-Yao channel or a write-protected channel because of a blockchain's nature as world readable without authentication. After testing these options with the case study presented in Section 5.3, we have determined that either option works to represent a blockchain and the theoretical advantages of a write-protected channel do not make a difference in verification for the case of our case study. The encoding of the blockchain as a channel fits because the blockchain is, by its nature, public, and a full copy of the blockchain is present on every node within a blockchain network. While the quorum requirements for a blockchain modification would make unauthorized writes to the blockchain unlikely, it is still critical to ensure a protocol is protected against unauthorized reads as well as unauthorized writes/modifications of the data on the blockchain, through encryption to ensure confidentiality against unauthorized reads and signing to ensure integrity against unauthorized modification of the data being stored on the blockchain by the protocol in question.

5.2 Modeling the Blockchain as an Actor

Another way to model protocols hosted on a blockchain is to incorporate the blockchain into the model specification as another actor. Under this paradigm, one user in the blockchain protocol could send messages to the blockchain, which is represented as another actor in the system, and the blockchain could send the message on to the other peer or user in the blockchain network. As an alternate perspective, one user could write to the blockchain by sending a message and another user could read from the blockchain by receiving the message sent by the blockchain.

As AVISPA and SPAN are not designed to model protocols based on a blockchain, it is not possible to have these tools assume the blockchain itself, when the blockchain is added to the protocol as an actor, is an intruder. For both CAS+ and HLPSL files, actors defined in the scope of the protocol definition are assumed to have good intentions and the intruder is an entirely different construct with its own set of knowledge as defined in a protocol specification. Therefore, attacks regarding a compromised blockchain likely cannot be detected by AVISPA and SPAN when the blockchain is represented as an actor.

5.3 Verifying a Blockchain Election Protocol

Liu and Wang [LW17], in their paper entitled "An E-Voting Protocol Based on Blockchain" propose a protocol for electronic voting that can be abstracted to only include messages being passed between actors in the protocol by utilizing the methods of treating the blockchain as a channel or as an actor. Therefore, this protocol should be able to be modeled within AVISPA without issue. As this is a protocol based on a blockchain, we can likely model the blockchain either as a channel or as an actor in the protocol specification. The representation of the blockchain as a channel appears to be more simple for this protocol, as the blockchain is only tangentially involved in the last step of the protocol, in which the vote is "recorded" on the blockchain. This recording of the vote could then be implicit by the vote's passing over the channel.

However, it is important to note that not every aspect of the E-Voting protocol can be modeled and needed to be abstracted out of the protocol in order to be able to model the protocols in a format compatible with AVISPA and SPAN. One of these aspects of the E-Voting protocol is the blind signature. While it appears that blind signatures have been modeled in HLPSL [SS14], this model adds unnecessary complexity that does not affect the authentication goals inherent to the protocol, so this was abstracted out to a hash of the Vote message that is manipulated wherever a message using a blind signature is needed in the protocol specification. Additionally, when the Vote is to be cast, we encrypted the Vote with a different private key so the Vote could not be traced back to the voter. The hash function was defined in CAS+ by defining a variable called HashFunc of type function, which is translated into a variable of type hash in the HLPSL code. Additionally, within CAS+, it is not possible to notate the "if these two messages match, then send another message" construct necessary for a by-the-letter encoding of Liu and Wang's E-Voting protocol and, while it may be possible to simulate this behavior in HLPSL by changing which message is sent based on what is received, this is not necessary if we presume that both the organizer and the inspector are honest. Therefore, this will be abstracted out as well, as the focus is on the security of the blockchain, not necessarily the honesty of the organizer and inspector involved in each voting transaction. As a workaround to these limitations inherent with AVISPA , mutual authentication goals based on the vote were established as follows:

```
Inspector authenticates Organizer on Vote;
Organizer authenticates Inspector on Vote;
```

This ensures the vote was not tampered with by either the organizer or inspector before being recorded at the last step of the protocol and also allows the protocol to be encoded solely using CAS+, as it is no longer necessary to simulate the equality check for the Vote within the messages being passed over the course of the protocol with these mutual authentication goals in place. If it is true that the Organizer authenticates the Inspector on the Vote, this would imply that the Organizer and the Inspector have the same copy of the Vote and the Vote should therefore be recorded on the blockchain. The ability to utilize CAS+ is key to this protocol for readability due to the number of messages being passed between actors. All necessary aspects of the protocol can be properly represented in either specification language, as both CAS+ and HLPSL support the necessary authentication goals to ensure the Vote is not tampered with. However, it is important to note that neither CAS+ nor HLPSL can guarantee the anonymity of the vote being cast, as anonymity goals are not implemented within CAS+ or HLPSL.

As there are two different paradigms which can be used to encode a protocol which runs on a blockchain, either modeling the blockchain as a channel through which messages are passed implicitly and transparently or as an actor to which messages are passed explicitly and forwarded on to their final destination. When implementing the E-Voting protocol as a channel, it is not important as to the type of channel used, as the additional encryption and hashing inherent in the E-Voting protocol provide the necessary security for the protocols to be analyzed as safe by AVISPA with the OFMC backend. Therefore, while the additional channel types afforded by the use of CAS+ are pedagogically interesting as blockchain, by its nature, is a write-protected medium that would lend itself well to being implemented as a write-protected channel, this is not strictly necessary to ensure the security of the protocol as a result of the protocol's design. Figure 8 is a representation of the E-Voting protocol with the blockchain represented as a channel through which messages are passed. The protocol consists of the following steps. First, the voter sends the organizer of the election their own public key for authentication purposes, a hash of the vote being cast, here represented as HashFunc(Vote), and the organizer's public key, all signed with the voter's private key. Then, the voter sends the same message to the inspector of the election. After this message is sent, the organizer responds with its public key, the hash of the vote being cast signed by the organizer as well as the voter's public key, all signed by the private key of the organizer. Then, the voter sends the inspector their public key, the has of the vote being cast, as well as a copy of the inspector's public key, all signed by the private key of the voter. This prompts the inspector to respond with their public key, another copy of the hashed vote, and the voter's public key, all signed by the inspector's private key. This adds assurance to the protocol that the voter is corresponding with the proper inspector. Finally, the voter sends the vote to the blockchain, represented implicitly in the as-channel version of the protocol as sending the vote message to both the inspector and the organizer. This message contains a new public key for the voter, used to ensure that the Vote cannot be traced back to the voter, ensuring the integrity of the secret ballot. In addition to the new public key, the message contains the vote itself, a hashed copy of the vote signed by the organizer, and a hashed copy of the vote signed by the inspector. This is all signed by the private key corresponding to the secondary public key contained in the message. This structure allows the vote to be validated but not traced back to the voter.

As the HLPSL code for the E-Voting protocol was not modified at all and is simply a direct compilation of the E-Voting protocol CAS+ code already presented in Figure 8, the HLPSL code for this protocol will not be presented in this work.

It is also possible to model this protocol with the blockchain portrayed as an actor within the protocol as opposed to the implicit channel through which the messages of the protocol are passed. This allows the blockchain to be more explicitly involved in the protocol, as the Voter would send

```
protocol eVotingChannel;

identifiers
HashFunc : function;
Voter,Organizer,Inspector : user;
Kvoter,Kvoter2,Kinspector,Korganizer : public_key;
Vote : number;

messages
1. Voter -> Organizer : {Kvoter,HashFunc(Vote),Korganizer}Kvoter'
2. Voter -> Inspector : {Kvoter,HashFunc(Vote),Korganizer}Kvoter'
3. Organizer -> Voter : {Korganizer,{HashFunc(Vote)}Korganizer',Kvoter}Korganizer'
4. Voter -> Inspector : {Kvoter,HashFunc(Vote),Kinspector}Kvoter'
5. Inspector -> Voter : {Kinspector,{HashFunc(Vote)}Kinspector',Kvoter}Kinspector'
6. Voter -> Organizer : {Kvoter2,Vote,{HashFunc(Vote)}Korganizer',{HashFunc(Vote)}Kinspector'}
Kvoter2'
7. Voter -> Inspector : {Kvoter2,Vote,{HashFunc(Vote)}Korganizer',{HashFunc(Vote)}Kinspector'}
Kvoter2'


knowledge
Voter : Organizer, Inspector, Kvoter, Kvoter2, Vote, HashFunc;
Organizer : Inspector, Korganizer, Voter, HashFunc;
Inspector : Voter, Kinspector, Organizer, HashFunc;

session_instances
[Voter:alice,Organizer:bob,Inspector:charlie,Kvoter2:kv2,Kvoter:kv,Kinspector:ki,Korganizer:ko,
HashFunc:hashfunc,Vote:v];

intruder_knowledge
alice;

goal
Inspector authenticates Organizer on Vote;
Organizer authenticates Inspector on Vote;
```

Figure 8: The E-Voting protocol from Liu and Wang with the blockchain represented as a channel as represented in CAS+

the message to the blockchain, and the blockchain would then forward the message onto whomever required the message. This action of forwarding by the blockchain is not precisely how blockchains function, however, the protocol must be encoded this way due to limitations present within AVISPA, whether using CAS+ or HLPSL. The blockchain forwarding the message can be read as the user receiving the message retrieving it from the blockchain, even though these limitations force the interaction with the blockchain to essentially be reversed. Figure 9 reflects the E-Voting protocol with the blockchain represented as an actor encoded in CAS+. As with the representation of the E-Voting protocol with the blockchain represented as a channel, the E-Voting protocol with the blockchain represented as an actor will not be presented in HLPSL as it is a direct conversion from the CAS+ representation of the protocol.

The flow of the E-Voting protocol with the blockchain represented as an actor in the system is similar to that of the protocol with the blockchain represented as a channel; however, there are a few important differences. To start, every message is threaded explicitly through the blockchain. So, instead of the Voter sending a message directly to the Organizer or Inspector, the Voter sends a message to the Blockchain, which then forwards the message on to the party to whom the message is intended. Additionally, in the last stage of the protocol in which the Vote is recorded, the Voter sends the Vote message, which is formatted in the same manner as in the protocol with the blockchain represented as a channel, to the Blockchain only once, and the Inspector and Organizer retrieve the Vote message from the Blockchain.

While this protocol is able to be modeled using CAS+, and by extension HLPSL, within AVISPA and SPAN, it is unclear as to the accuracy of the verification performed by the AVISPA backends, as the OFMC backend contends that the protocol is safe, but there is not a proper trace of the protocol generated by SPAN. The message sequence chart which is generated by SPAN only represents one of the transitions present within the protocol as opposed to representing the entirely of the protocol. More importantly, the protocol appears to be verified as safe even when a purposeful error, that of sending different Vote values to the Organizer and Inspector, is introduced into the CAS+ protocol specification. This indicates that the ability to detect whether a Vote is tampered with is beyond the scope of the AVISPA and SPAN tools. This appears to be a result of the encryption and signing

```
protocol eVotingActor;

identifiers
HashFunc : function;
Voter,Organizer,Inspector,Blockchain : user;
Kvoter,Kvoter2,Kinspector,Korganizer : public_key;
Vote : number;

messages
1. Voter -> Blockchain : {Kvoter,HashFunc(Vote),Korganizer}Kvoter'
2. Blockchain -> Organizer : {Kvoter,HashFunc(Vote),Korganizer}Kvoter'
3. Organizer -> Blockchain : {Korganizer,{HashFunc(Vote)}Korganizer',Kvoter}Korganizer'
4. Blockchain -> Voter : {Korganizer,{HashFunc(Vote)}Korganizer',Kvoter}Korganizer'
5. Voter -> Blockchain : {Kvoter,HashFunc(Vote),Kinspector}Kvoter'
6. Blockchain -> Inspector : {Kvoter,HashFunc(Vote),Kinspector}Kvoter'
7. Inspector -> Blockchain : {Kinspector,{HashFunc(Vote)}Kinspector',Kvoter}Kinspector'
8. Blockchain-> Voter : {Kinspector,{HashFunc(Vote)}Kinspector',Kvoter}Kinspector'
9. Voter -> Blockchain : {Kvoter2,Vote,{HashFunc(Vote)}Korganizer',{HashFunc(Vote)}
Kinspector'}Kvoter2'
10. Blockchain -> Organizer : {Kvoter2,Vote,{HashFunc(Vote)}Korganizer',{HashFunc(Vote)}
Kinspector'}Kvoter2'
11. Blockchain -> Inspector : {Kvoter2,Vote,{HashFunc(Vote)}Korganizer',{HashFunc(Vote)}
Kinspector'}Kvoter2'

knowledge
Voter : Organizer, Inspector, Kvoter, Kvoter2, Vote, HashFunc, Blockchain;
Organizer : Inspector, Korganizer, Voter, HashFunc, Blockchain;
Inspector : Voter, Kinspector, Organizer, HashFunc, Blockchain;

session_instances
[Voter:alice,Organizer:bob,Inspector:charlie,Kvoter2:kv2,Kvoter:kv,Kinspector:ki,
Korganizer:ko,HashFunc:hashfunc,Vote:v,Blockchain:b];

intruder_knowledge
alice;

goal
Inspector authenticates Organizer on Vote;
Organizer authenticates Inspector on Vote;
```

Figure 9: The E-Voting protocol from Liu and Wang with the blockchain represented as an actor as represented in CAS+

inherent in this protocol as changes to the encrypted messages in the protocol specification appear to be ignored by the verification framework. Therefore, if everything goes correctly with the run of the protocol, it can be safely stated that the E-Voting protocol is secure; however, any errors in the protocol which occur at runtime, such as the substitution of a different Vote between what is received by the Inspector and the Organizer, do not appear to be detected by the AVISPA toolchain. It should be the case that the Vote is only recorded if the Organizer and Inspector receive the same Vote information from the Voter. Therefore, it appears that the verification of the goals for this protocol is beyond the scope of what can be performed by AVISPA and SPAN. While, theoretically, an authentication goal should enforce that the same Vote was received by the Organizer and the Inspector, as this is the definition of an authentication goal, this is not what is occurring when the E-Voting protocol as encoded in CAS+ or HLPSL is verified using AVISPA and SPAN. However, this could also be the fault of our encoding of the protocol within CAS+ and it is not possible to tell whether this is a limitation of AVISPA or an issue with our representation of the protocol.

6 Conclusion

In conclusion, we have determined that it is possible to use protocol verification tools, such as AVISPA and SPAN in the context of this work, to verify a basic protocol involving communication between two users as well as a protocol used to establish a compound identity for communication on a blockchain. These protocols were verified as being safe by the OFMC default backend within AVISPA and message sequence charts were properly generated for the protocols within SPAN. The results changed entirely when looking at the electronic voting protocol that runs on top of a blockchain, as it appears that AVISPA and SPAN are unable to verify the security or lack thereof inherent in this protocol. The encoded electronic voting protocol was not properly verified by SPAN. This may be a result of a flaw in the protocol, however, it is more likely that this is a result of the limitations of AVISPA or the way we decided to encode the protocol in CAS+. Whether the blockchain is represented as a channel or as an actor, is reported as safe by these tools even when an error in the protocol is purposefully introduced. This appears to be an indication that this type of protocol, with multiple layers of encryption and hashing, is undecidable by the AVISPA toolchain and produces an erroneous result when verified by the OFMC model checking backend which is the focus of this work. Therefore, while the protocol appears to be secure when reasoning about it, because the Votes are signed and it is verified that the inspector and organizer have the same vote prior to the vote being recorded on the blockchain. It is unclear as to why this goal of ensuring the same Vote is received by the Organizer and Inspector is not able to be analyzed within AVISPA and SPAN. When this protocol is analyzed, not only is the protocol recognized as safe even with the error introduced, there is not a full message sequence chart generated by SPAN for the protocol. Only one of the messages, in which the Voter sends the Inspector a copy of the hashed Vote, is represented in the message sequence chart for the protocol with the blockchain represented as an actor. For the protocol with the blockchain represented as a channel, no message sequence chart is generated at all. This could reflect an issue in our encoding of the protocol, which is unlikely as the protocol was encoded directly into CAS+ from the protocol specification in Liu and Wang's work. More likely, this reflects a limitation of the AVISPA and SPAN tools in dealing with protocols in which messages are tupled together and doubly encrypted. Therefore, it appears that model checking utilities, at least AVISPA and SPAN, cannot be effectively used to determine the security of every protocol, especially those running on top of a blockchain with complex double encryption requirements. It is unclear. however, if these limitations are the result of fundamental limitations of model checkers for security protocols or if these limitations are the result of the maturity, or lack thereof, of the toolchain that consists of AVISPA and SPAN.

Additionally, this lack of maturity in the AVISPA and SPAN toolchains also presented technical issues in getting the tools to run properly. These tools are quite old, with AVISPA last being modified around 2006 and SPAN last being modified around 2016 while relying on 32-bit libraries last modified around 2010. No matter which modern Linux distribution I attempted to use when running these tools, I only got as far as an error message indicating a missing library or mismatched library version. Therefore, the only reasonable way I found to run these tools was to utilize a pre-configured VirtualBox VM which consisted of Ubuntu 10.04 with the tools preinstalled. While this was able to get the tools up and running, the fact that this was necessary would serve to indicate that these tools were not designed with future-proofing in mind and were likely designed as a sort of proof-of-concept that is not production ready.

While it is clear that the limitations involving compatibility with modern Linux distributions are a reflection on the lack of maturity of the AVISPA and SPAN tools, it is not as clear as to if the limitations we have discovered in terms of the protocols that can be analyzed and what can be detected by AVISPA and SPAN are fundamental limitations of the model checking technology, indicating that the types of blockchain protocol we have analyzed in AVISPA are undecidable in terms of security without regard to the toolchain used, or are limitations of the AVISPA and SPAN model checkers specifically, indicating that it may be possible to verify these protocols in other model checking utilities. Therefore, because of these limitations within the AVISPA toolchain, it is not production-ready and, while model checkers can be used to verify security protocols, we have concluded that the AVISPA and SPAN tools are not ready for usage in industry, especially as it comes to the verification of blockchain protocols.

References

[ACC16] Alessandro Armando, Roberto Carbone, and Luca Compagna. Satmc: a sat-based model checker for security protocols, business processes, and security apis, April 2016.

[Avi06] Avispa Project. Avispa project web site, 2006.

[BHK09] Yohan Boichut, Pierre-Cyrille Heam, and Olgsa Kouchnarenko. Tree automata for detecting attacks on protocols with algebraic cryptographic primitives, July 2009.

[Bla18a] Bruno Blanchet. Cryptoverif: Cryptographic protocol verifier in the computational model, 2018.

[Bla18b] Bruno Blanchet. Proverif: Cryptographic protocol verifier in the formal model, 2018.

[CKY17] Jason Paul Cruz, Yuichi Kaji, and Naoto Yanai. Rbac-sc: Role based access control using smart contract. *IEEE Access*, 6:12240–12251, 2017.

[DY83] Danny Dolev and Andrew C. Yao. On the security of public key protocols, March 1983.

[Gen17] Thomas Genet. Span - security protocol animator for avispa, 2017.

[God05] Jens Chr. Godskesen. The needham-schroeder public-key protocol–how to break it, 2005.

[Ins] Infosec Institute. Cia triad.

[KMS$^+$16] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 839–858, May 2016.

[Kou18] Petros Koutoupis. Blockchain, part i: Introduction and cryptocurrency, 2018.

[LW17] Yi Liu and Qi Wang. An e-voting protocol based on blockchain, 2017.

[MV09] Sebastian Modersheim and Luca Vigano. The open-source fixed-point model checker for symbolic analysis of security protocols, 2009.

[Pec17] Morgen E. Peck. Blockchains: how they work and why they'll change the world. *IEEE Spectrum*, October 2017:28–35, 2017.

[Pro06] Avispa Project. Hlpsl tutorial, 2006.

[SG11] Ronan Saillard and Thomas Genet. Cas+, March 2011.

[SS14] Neetu Sharma and Birendra Kumar Sharma. New provably secure blind signature scheme with weil pairing, May 2014.

[Tur06] Mathieu Turuani. The cl-atse protocol analyser, 2006.

[ZNP15] G. Zyskind, O. Nathan, and A. '. Pentland. Decentralizing privacy: Using blockchain to protect personal data. In *2015 IEEE Security and Privacy Workshops*, pages 180–184, May 2015.