VoCEM: Verification of Cryptography in Element/Matrix

Robert Herndon
rherndon@ucsd.edu
University of California San Diego
La Jolla, California, USA

Huanlei Wu
huw012@ucsd.edu
University of California San Diego
La Jolla, California, USA

Abstract

Element is an instant messaging application that provides end-to-end encryption to ensure messages are only looked at by the intended recipient(s). We analyze Element's encryption scheme, which turns out to be the Matrix open standard using the Olm and Megolm libraries. VoCEM models Matrix by using an Encrypt-then-MAC scheme which is used within encryption schemes for Matrix. These models work by using verification languages to ensure that the base component of Matrix is protecting the key used to derive the encrypting and signing keys used for message communication. The VoCEM models show that the Encrypt-then-MAC representation of Matrix, and therefore Element, are securing the private key which in turn secures messaging.

Keywords: security, cryptography, verification, end-to-end encryption, Alloy, ProVerif, Matrix, Element

1 Introduction

Security and privacy are one of the many concerns of those who use the Internet. Important sensitive information can be leaked through various of techniques, such as malware attacks, man-in-the-middle attacks, and phishing. Just recently, the FBI arrested more than 800 criminals in an operation dubbed "Operation Trojan Shield". To obtain evidence of illegal activities, the FBI used an instant messaging platform called Anom. Anom was advertised as an application that used end-to-end encryption to avoid being monitored by law enforcement. However, it turned out that the FBI had a master key to decrypt the chat messages sent on the app. [15] While the FBI used this power to punish criminals, their operation also brings up a reasonable doubt among the average citizen of whether or not their messages could be viewed by unintended sources.

As such, our problem and interest lies in the security aspects of an application, specifically the end-to-end encryption of messages. However, many instant messaging applications are not open source, creating a black box between the the application and the user. Fortunately, the instant messaging application Element has its codebase updated publicly to Github. Element is an instant messaging platform that uses end-to-end encryption to ensure messages safely reach the intended recipients. The confidence in privacy and security is extremely important, especially if an application is advertised as such, and Element claims to be a highly secure encrypted chat application. Furthermore, what makes analyzing Element even more relevant is the fact that there were some findings [2] in 2016 by the NCCgroup that presented some security problems with Matrix, the encryption protocol Element uses, and the Olm and Megolm libraries. Matrix claimed it fixed some of these problems by revising the libraries used by Matrix and the client SDK itself.

In this paper, we will explain how Matrix implements its end-to-end encryption mechanism using the Olm and Megolm library as well as describe the models we created in ProVerif and Alloy. These models, which we call VoCEM models, can be found on https://github.com/mcnerd16/VoCEM. Unfortunately, we were not able to completely model the end-to-end encryption protocol; the models in this paper represents the Encrypt-then-MAC approach that Element uses to encrypt its messages. We also impart some insights about implementing a cryptographic protocol using ProVerif and Alloy, providing both benefits and drawbacks of the two modeling tools.

The paper will be organized as follows.

- Section 2 will briefly introduce the relevant applications and softwares mentioned in the paper.
- Section 3 will discuss the literatures we referred to when understanding and creating the Matrix models.
- Section 4 will talk about our process of understanding the encryption mechanism in Matrix as well as describe the encryption mechanism itself. Furthermore, we talk about our process of learning and creating the Matrix model in both ProVerif and Alloy.
- Section 5 will explain our results from our model.
- Section 6 will discuss our experience with the two modeling tools, elaborating on the benefits and drawbacks of each.
- Section 7 will conclude our paper and provide ideas for future works.

2 Background

2.1 Element

Element is an instant messaging app on desktop, Android, iOS, and web. It started off as an application called Vector

Authors' addresses: Robert Herndon, rherndon@ucsd.edu, University of California San Diego, La Jolla, California, USA; Huanlei Wu, huw012@ucsd.edu, University of California San Diego, La Jolla, California, USA.

but changed its name to Riot, and finally ended up as a secure messaging application called Element. To protect its users privacy and live up to what it claims, Element lets its users choose the geographical location of their stored data, allowing them to change it if they wish. The website also claims that Element uses "true decentralized end-to-end encryption," indicating that Element has no way of knowing or decrypting the users' messages as only those in the conversation has the decryption key.

The Element web code base, which we look at in our project, can be found at https://github.com/vector-im/element-web. As mentioned in the introduction, Element is only a wrapper for the end-to-end encrpytion open standard Matrix.

2.2 Matrix

Matrix is an open standard for interoperable, decentralized, real-time communication over IP. Interoperability refers to Matrix's ease of combining Matrix with other communication systems such as bridging for Slack and Discord; decentralization refers to the fact that any user can create and connect to any other peer without a central server. Like Element, it is an open source project and can be found at https://github.com/matrix-org. Due to Matrix being open source, anyone can use it; in 2019, the French government is using a modified version of Matrix for its communication system [19].

2.3 ProVerif

ProVerif is a software that uses the programming language OCaml for modeling and automatically analyzing the security of cryptographic protocols [5]. It provides support for cryptographic primitives such as symmetric and asymmetric encryption and digital signatures, which is necessary to model Matrix. For the analysis portion, it is capable of proving reachability properties, correspondence assertions, and observational equivalence in an unbounded session and message space.

2.4 Alloy

The Alloy Analyzer is an open source software for modeling different kinds of applications using the Alloy specification language. Its use is very flexible, ranging from analyzing security mechanisms to designing telephone switching networks [1]. The way Alloy works is by taking in a set of constraints written in some form of first order logic and finding structures to satisfy them. From these structures, Alloy can find counterexamples and create diagrams that help the user detect flaws and bugs within their system.

3 Related Work

Analyzing the Matrix open standard is not completely new. In 2016, the NCC Group's Cryptography Services Practice reviewed the Olm and Megolm library used by Matrix to implement their end-to-end encrpytion. They published their findings [2] on their website. From their tests, the NCC group found that:

- Both Olm and Megolm are weak to unknown key-share attacks, a type of attack where one person (Alice) believes she shares a key with another person (Bob), but Bob believes he shares a key with yet another person (Eve) when in actuality, he shares a key with Alice.
- An attacker or group member can replay chat messages of any user, invading the privacy of that user.
- There are concerns for future and backward secrecy as chats are "long-lived." While these chats are protected by ephemeral keys, compromise of these keys can result in past messaged being retrieved.

In order to mitigate these potential attack vectors, the paper suggested a few ideas, such as signing ephemeral keys, adding in unique identifying elements into the MAC of the initial message, and erasing randomness buffers.

Furthermore, in 2020, Floris Hendriks published a paper [8] discussing in great lengths how Matrix and its key management system worked. For his analysis, he looked at forward and backward secrecy, deniability, confidentiality, integrity, and authentication, similar to the security features the NCC group looked at. From the paper's description of Matrix's infrastructure, Matrix took the NCC group's advice and added signing to the one-time keys, which is confirmed by our findings.

Another paper studying the Matrix open standard and Element (called Riot.im in the paper) [17] discovered that while the end-to-end encryption is "good," data stored by the application on the hard drive of a device could be accessed and analyzed. Therefore, the paper focused on discovering forensic artifacts rather than Element's cryptographic components. However, the paper did provide more insight into how Element and Matrix interacted with each other.

Outside of understanding Matrix, [10, 11] helped us build our Matrix model in the Alloy Analyzer. The goal of the two literatures was to model attacks, but they were still useful in our case. The former literature is the main paper that describes its implementation of modeling DES triple modes in Alloy, and the latter literature is an extension to the first that contained the Alloy implementation. Matrix uses AES-256 in counter mode to encrypt and pad the message, and the papers aided in our Alloy design.

4 Methodology

In this section, the process for validating the Matrix encryption code will be discussed. First, the manual code review of both Element and Matrix repositories lead to the encryption implementation code. Second, verification tools needed to be decided and models created, ultimately using ProVerif and Alloy to validate our models. Third, developing the models

to properly represent the Matrix implementation code to validate both security and correctness.

4.1 Code Review

The repositories for both Element and Matrix needed to be reviewed to identify which pieces of code contained cryptographic code to be used to encrypt or decrypt messages primarily. The search started at the repository for Element Web eventually ending in Matrix JS.

4.1.1 Element Web. Starting in Element's web repository [7], the source was looked through thoroughly to try to identify code relating to cryptography and message sending, where calls to one function, *_ipcCall*, are made involving messages. This function calls *window.electon.send('seshat',...)* with its arguments, appearing to pass the message and encryption to another process. It should be noted that Electron [16] is used to perform this communication between server and client. Looking into seshat, it is a database tool for Matrix for events, including messages, are indexed. However, no cryptographic code exists here, but it revealed that the search should be shifted toward Matrix. Since Element Web is utilizing Matrix React for most of its function, the search continued with Matrix's repository.

4.1.2 Matrix React. Briefly looking into Matrix React[14], similar to the review of Element Web, most of the code within the repository does not deal with cryptography and message sending. While some messaging components exist within this codebase, most of the cryptographic seems to be offloaded to an older repository by Matrix, Matrix JS [13], the older engine. Several files import from the Matrix JS's crypto folder and files within it thus implying that most of the cryptography described by Matrix and Element are likely within this folder, which was found to be true. This lead to a thorough investigation of the crypto folder within Matrix JS's repository.

4.1.3 Matrix JS. Looking into the crypto folder of this repository [13], all the cryptography code implementations were found within this folder among several files. The primary encryption schemes used are Olm and Megolm for devices.

Olm/Megolm Encryption Schemes. Olm is a peer-to-peer encryption scheme [18], while Megolm is a session-encryption scheme [6], which is used more widely across Matrix. Both are ratchet schemes that provide the keys for encrypting and signing. Ratchet schemes constantly change the keys [3] in order to provide encryption for the two users, who share a secret. This secret is typically set up with a key exchange or peer-to-peer connections, happening in Olm and Megolm respsectively. Both utilize AES and SHA256 in order to achieve their ciphertext and MAC components for the payload to be sent to a recipient or group.

Reducing the Schemes. Using the Olm and Megolm schemes, the main symmetric encryption scheme of Matrix boils down to an Encrypt-then-MAC scheme using AES and SHA256 [12], which is known to be semantically secure. Using the name and key(s) provided to the algorithm, the AES and HMAC key will be created to ensure that both can be recreated for the recipient of the message. Using AES in counter mode (AES-CTR), the message is encrypted using the AES key and a random IV to form a ciphertext. This ciphertext is signed with the HMAC key using an HMAC construction relying on the SHA256 algorithm to create a signature for this ciphertext. The IV, ciphertext, and signature are sent as the event for the message to the recipient. The recipient can then perform similar actions to return the ciphertext to the message, using the signature to validate the message. The recipient recreates the keys used by performing the same operations as the sender to obtain the same keys. The recipient then verifies that the signature matches that of the ciphertext. If it does not, it rejects the ciphertext as it is likely not from the sender. Otherwise, it continues to decrypt the message using AES-CTR to decrypt the message to plaintext, allowing the recipient to read the message of the sender.

4.2 Validation Languages

Two validation languages were heavily considered during this project, being Alloy and ProVerif. Both tools readily available with different capabilities and are used to create the models of Matrix. Both are useful tools for verifying, where code can be tested without an implementation.

4.2.1 Alloy. Alloy is used to verify and test code by writing a simple script. This script must contain signatures, or types, for each object to be used in the verification process. These can then be used in either predicates, facts, or functions to produce the desired output for specific functionality. Assertions can be checked along with predicates and functions ran to validate the assumptions within the functionality code. If assertions are true, the check can be passed with no counterexamples, where otherwise Alloy will provide a specific counterexample to show that the assumptions are not correct. This can also be done with running predicates, where an instance can be provided by Alloy if one exists to prove the conditions within.

4.2.2 ProVerif. ProVerif is a powerful verification language. To develop a model in ProVerif, declarations, process macros, and a main process can be defined to create a well defined model. The declarations are generic functions that can be used to describe an algorithm without any implementation. They are simple to write and can be used to ensure that two functions are inverses of each other. Additionally Process Macros will allow a process to perform multiple actions to simulate multiple function calls, which are useful for protocol attacks. The process then performs the verification,

ensuring that the declarations and process macros satisfy the requirements of the process.

4.3 Model Development

For model development for ProVerif, the manual and online examples were used as a reference to develop the model for verification and security. For model development with Alloy, their online tutorial and previous research helped with developing the model.

4.3.1 ProVerif Maunal. The ProVerif manual contains information of how to build a model using ProVerif's syntax and keywords [5]. This contains enough information for the model, but it only provides a few examples, mostly of handshaking protocol and public key encryption. Since these are more popular topics within the field of cryptography, this makes sense, but some more information was needed as well as more examples to help the process of model development.

4.3.2 ProVerif Examples. ProVerif additionally has examples on their website, containing more generic examples and more ideas to develop off of [4]. Using these examples helped with understanding how models were to look and function as a unit, being more helpful than the manual in some cases. While some examples appeared in the manual with detailed description, the rest of the examples were little documented, requiring more detailed analysis to identify the nature of the model represented. Using the manual to identify the keyterms and inference to determine the function and use the output to analyze what makes a model well defined.

4.3.3 Alloy Tutorial. On Alloy's website, they provide online tutorials and examples which are easy to follow and help develop an understanding on how to create models for a given application [9]. They provide tutorials for file systems and the river crossing problem, and additional examples for their publication, which was not reviewed. These tutorials provided the structure and the basis of knowledge required to develop the model of Matrix in Alloy and have similar qualities to the ProVerif Model.

4.3.4 Alloy Operational Modes. Alloy has been used to validate attacks against certain cryptographic schemes [10, 11]. Using previous work as a guideline, it made developing a basic model possible. This work provided semantics on how to create other signatures using functions within to develop a sound model. For example, a signature Message can have a member decrypt to allow a key and ciphertext to be converted into a Message signature, matching decryption. This is used among most types to provide similar function between the ProVerif and Alloy code such that they can be compared to each other within Section 6.

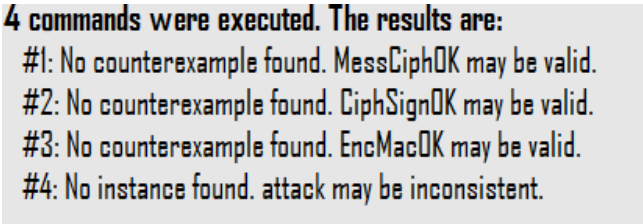

Figure 1. Alloy Model output showing not attack can be performed to achieve the private key. Additionally showing that encryption, signing and Encrypt-then-MAC are secure functions.

ProVerif text output:

```
-- Process 1-- Query not attacker(privateKey[]) in process 1
Translating the process into Horn clauses...
Completing...
Starting query not attacker(privateKey[])
RESULT not attacker(privateKey[]) is true.

------------------------------------------------------------
Verification summary:

Query not attacker(privateKey[]) is true.

------------------------------------------------------------
```

Figure 2. ProVerif Model output showing that the attacker is unable to acquire the private key.

4.4 VoCEM Model

Our models, called VoCEM, describe the basic Encrypt-then-MAC scheme and test if an attacker is able to determine a private key, that is the key used for generating. Appendix A contains the script for the ProVerif model and Appendix B contains the script for the Alloy model. Both perform similar functions, having encryption, signing, and Encrypt-then-MAC available, testing an adversary's ability to obtain the private key for the given encrypting key and signing key.

5 Evaluation

Figure 1 shows the output from our Alloy model. We incrementally tested our Alloy code, as seen in execution results #1-3, ensuring that our predicates and facts were declared correctly. When we ran the attack command, Alloy returned execution result #4, indicating that the attack model we declared could not retrieve the private key. Figure 2 shows the output for our ProVerif model. Similar to result #4 in the Alloy Analyzer output, ProVerif also found that the attacker could not retrieve the private key. Therefore, from running the model scripts for ProVerif and Alloy in their respective interpreters/compilers, we can conclude that the attacker is unable to obtain this private key from the encryption and signing keys, even if they can query the Encrypt-then-MAC functions.

6 Discussion

6.1 Exact Model

The VoCEM models created are not perfect models of Matrix itself, but can be used to ensure that its base component is secure. A more specific model can be made at a later time, fully ensuring that each step of the Olm or Megolm encryption steps is properly secure. The VoCEM models show that the Encrypt-then-MAC steps are secure for maintaining the private key used to generate the other keys. This private key is produced from the ratchet algorithms of Olm and Megolm and would allow an adversary to have a state within the ratchet, which would compromise the security of Matrix. For this reason, the VoCEM models present that the private key is secure even when querying the functions to encrypt and decrypt.

6.2 Alloy Key Generation Fact

Within the Alloy model (see Appendix B), an important fact about key generation was discovered. Key generation for this Encrypt-then-MAC scheme is extremely important to the integrity of the system, where if implemented poorly, can lead to security problems. The following statements allow the model to have no counterexamples and no instances of attacks for the Encrypt-then-MAC.

$$\text{createEK.pk1} = \text{createEK.pk2} \Rightarrow \text{createSK.pk1} \neq \text{createSK.pk2} \quad (1)$$

$$\text{createSK.pk1} = \text{createSK.pk2} \Rightarrow \text{createEK.pk1} \neq \text{createEK.pk2} \quad (2)$$

These state that for any public key that shares a generated key, the other generated key must be different. If both generated keys are the same, the key generation has a weakness that can be exploited, even if it is only for two private keys. As the keys shift using ratcheting, and a finite number of keys can be used, it is likely that some keys repeat, being few and far between ideally. Without these statements, the model found that the Encrypt-then-MAC would not register properly and an attack can be performed.

6.3 Modeling Languages

The differences between the modelling languages ProVerif and Alloy have become very apparent and each has its benefits and drawbacks.

6.3.1 Alloy.

Benefits. Alloy is simple to understand and a great application for a variety of applications. The tutorial used file systems and a puzzle problem, which can be expanded to riddles, graphs, and other complicated problems to help simplify them. These were easy to follow and provided a good amount of information of developing models for Alloy.

Alloy uses a visualization to show counterexamples and instances, allowing the content to be viewed as a table, tree, or graphically. Additional tools are available for visualization and expanding Alloys usefulness that are also open sourced. They endorse some on their website.

Drawbacks. Functions are difficult to write in Alloy, making predicates much simpler to create. These are still convoluted compared to the ProVerif declarations, needing to explicitly define each part of the action, rather than a portion. In fact, facts may need to be used to force certain behavior to adjust the model, showing which areas may have problems, but also restricting the model.

6.3.2 ProVerif.

Benefits. Very simple declarations allow for powerful assumptions. A pair of functions can be written in three lines to define encryption and its basic semantics. ProVerif allows implications to be made based on how the line is written to reduce the amount of lines for a model.

The attacker is also implied in most models, allowing the adversary to attempt any sort of attack rather than a specified one, generated by the Alloy code. This allows for stronger attacking models, as no code needs to be spent describing the attack. The output will show how an adversary may be able to bypass a protocol, helping aid development of implementation.

Drawbacks. The examples and tutorial are much less involved, requiring more analysis to fully understand how each example worked and how to use it to aid development for a new model. It took roughly twice the amount of time to create the ProVerif model than it did to create the Alloy model, albeit the ProVerif model was created first.

ProVerif also is primarily developed for cryptographic work, meaning it will be less applicable for other problem sets. This has a specific use case in which it can succeed, but limits it usage.

7 Conclusion

The VoCEM models describe an Encrypt-then-MAC encryption scheme that provides the security for Matrix and further Element. The models used have been shown to have correctness and properly working encryption and signing when necessary. These models show that this base component of Matrix is secure to keeping the private key secure, so adversaries will be unable to obtain message data. While the Encrypt-then-MAC model is not a perfect representation of Matrix, it suffices as a general approximation for the base security of the most basic function. The important fact discovered is that keys need distinct generated keys to maintain security of the messages.

Due to time constraint, we were not able to model Matrix in full. In the future, we would like to add the low level details of the protocol, such as an increment-by-one counter for the AES-256 counter mode as well as some high level

representations of the Olm and Megolm libraries. From the improved model, we hope to find some more invariants or discover a counterexample in which the process is violated in some way. Another interesting direction we can look at is replicating the Encrypt-then-Mac encryption scheme in other modeling software and comparing the benefits and drawbacks to our VoCEM models.

References

[1] Alloy. About. https://alloytools.org/about.html, 2017.

[2] A. Balducci and J. Meredith. Olm cryptographic review. https://www.nccgroup.trust/globalassets/our-research/us/public-reports/2016/november/ncc_group_olm_cryptogrpahic_review_2016_11_01.pdf, 2016.

[3] M. Bellare, A. C. Singh, J. Jaeger, M. Nyayapati, and I. Stepanovs. Ratcheted encryption and key exchange: The security of messaging. In *Katz J., Shacham H. (eds) Advances in Cryptology – CRYPTO 2017*, volume 10403 of *CRYPTO '17*. Springer, Cham, 2017.

[4] B. Blanchet, V. Cheval, , M. Sylvestrer, and S. Malladi. Welcome to online demo for proverif, 2016.

[5] B. Blanchet, B. Smyth, V. Cheval, , and M. Sylvestrer. Proverif. https://prosecco.gforge.inria.fr/personal/bblanche/proverif/manual.pdf, 2017.

[6] H. Chathi. Megolm group ratchet. https://gitlab.matrix.org/matrix-org/olm/-/blob/master/docs/megolm.md, 2021.

[7] Element. element-web. https://github.com/vector-im/element-web, 2021.

[8] F. Hendriks. Analysis of key management in matrix. https://www.cs.ru.nl/bachelors-theses/2020/Floris_Hendriks___4749294___Analysis_of_key_management_in_Matrix.pdf, 2020.

[9] D. Jackson, R. Seater, G. Dennis, D. L. Berre, and F. Chang. Tutorial for alloy analyzer 4.0, 2019.

[10] C. Kalyvas, E. Konstantinou, and G. Kambourakis. Implementation of cryptographic modes in alloy (online material). http://www.icsd.aegean.gr/postgraduates/icsdm10002/, 2012.

[11] C. Kalyvas, E. Konstantinou, and G. Kambourakis. Modeling multiple modes of operation with alloy. In T.-h. Kim, A. Stoica, W.-c. Fang, T. Vasilakos, J. G. Villalba, K. P. Arnett, M. K. Khan, and B.-H. Kang, editors, *Computer Applications for Security, Control and System Engineering*, pages 78–85, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[12] Matrix-org. Aes.js. https://github.com/matrix-org/matrix-js-sdk/blob/develop/src/crypto/aes.js, 2021.

[13] Matrix-org. matrix-js-sdk. https://github.com/matrix-org/matrix-js-sdk, 2021.

[14] Matrix-org. matrix-react-sdk. https://github.com/matrix-org/matrix-react-sdk, 2021.

[15] U. S. D. of Justice. Lancement de tchap : la messagerie instantanée des agents de l'État, 2021.

[16] OpenJS. Electron, 2013.

[17] G. C. Schipper, R. Seelt, and N.-A. Le-Khac. Forensic analysis of matrix protocol and riot.im application. *Forensic Science International: Digital Investigation*, 36:301118, 2021. DFRWS 2021 EU - Selected Papers and Extended Abstracts of the Eighth Annual DFRWS Europe Conference.

[18] R. van der Hoff. Olm: A cryptographic ratchet. https://gitlab.matrix.org/matrix-org/olm/-/blob/master/docs/olm.md, 2020.

[19] R. Wadoux. Lancement de tchap : la messagerie instantanée des agents de l'État, 2019.

A ProVerif Code

```
free c : channel.
(*Encryption Declarations*)
type enckey.
fun senc(bitstring,enckey):bitstring.
reduc forall x:bitstring, k:enckey; sdec(senc(x,k),k) = x.
(*Signature Declarations*)
type signkey.
fun sign(bitstring,signkey) : bitstring.
reduc forall x:bitstring, k:signkey; verify(sign(x,k),k) = x.
(*Key Gen Declarations*)
type privkey.
fun enckeygen(privkey): enckey.
fun signkeygen(privkey): signkey.
(*Our Private Key*)
free privateKey: privkey [private].
query attacker(privateKey).
(*Main Process*)
process
  let ek = enckeygen(privateKey) in
  let sk = signkeygen(privateKey) in
  (*Perfrom encryption*)
  in(c,y:bitstring);
  let ciph = senc(y,ek) in
  out(c,(ciph ,sign(ciph,sk)));
  (*Perfrom decryption*)
  in(c,(cipher:bitstring,sig:bitstring));
  if (verify(cipher,sk) = sig) then
  out(c,sdec(cipher,ek))
```

B Alloy Code

```
//cipher objects
sig PrivKey { }
sig EncKey { createEK : one PrivKey }
sig SignKey { createSK : one PrivKey }
sig Message { dec: EncKey one -> one Ciphertext }
sig Ciphertext { enc: EncKey one -> one Message, verifying : SignKey one -> one Signature }
sig Signature { signing : SignKey one -> one Ciphertext }
//Encryption Standards
pred EncryptionValid{
  all m:Message,k:EncKey | lone c:Ciphertext | enc.m.k = c
  all c:Ciphertext,k:EncKey | lone m:Message | dec.c.k = m
  all m:Message,k:EncKey | dec.(enc.m.k).k = m
  all c:Ciphertext,k:EncKey | enc.(dec.c.k).k = c
}
//Signature Standards
pred SigningValid{
  all c:Ciphertext, k:SignKey | lone s:Signature| signing.c.k = s
  all s:Signature, k:SignKey | lone c:Ciphertext | verifying.s.k = c
  all c:Ciphertext, k:SignKey | verifying.(signing.c.k).k = c
}
//Key Generation Facts (pred creates counterExample)
fact KeyGen {
```

```
  all pk:PrivKey | lone ek:EncKey | createEK.pk = ek
  all pk:PrivKey | lone sk:SignKey | createSK.pk = sk
  all pk1,pk2:PrivKey | createEK.pk1 = createEK.pk2 => createSK.pk1 != createSK.pk2 //Strongest
  all pk1,pk2:PrivKey | createSK.pk1 = createSK.pk2 => createEK.pk1 != createEK.pk2 //Strongest
}
//Checks if encryption is valid
MessCiphOK: check {
  all m1,m2:Message,k:EncKey | m1 != m2 => enc.m1.k != enc.m2.k //no repeated ciphertext per message
  all m:Message,k1,k2:EncKey | k1 != k2 => enc.m.k1 != enc.m.k2 //not repeated ciphertext per key
  all c1,c2:Ciphertext,k:EncKey | c1 != c2 => dec.c1.k != dec.c2.k //no repeated message per ciphertext
  all c:Ciphertext,k1,k2:EncKey | k1 != k2 => dec.c.k1 != dec.c.k2 //not repeated message per key
}
//Checks if signing is valid
CiphSignOK: check {
  all c1,c2:Ciphertext,k:SignKey | c1 != c2 => signing.c1.k != signing.c2.k //no repeated signature
                                                                            //per ciphertext
  all c:Ciphertext,k1,k2:SignKey | k1 != k2 => signing.c.k1 != signing.c.k2 //no repeated signature
                                                                            //per key
  all s1,s2:Signature,k:SignKey | s1 != s2 => verifying.s1.k != verifying.s2.k //no repeated ciphertext
                                                                              //per signature
  all s:Signature,k1,k2:SignKey | k1 != k2 => verifying.s.k1 != verifying.s.k2 //no repeated ciphertext
                                                                              //per key
}
//Encrypt then MAC Standards with Key Generation
pred EncThenMAC {
  all pk:PrivKey,m:Message | lone c:Ciphertext | enc.m.(createEK.pk) = c
  all pk:PrivKey,c:Ciphertext | lone m:Message | dec.c.(createEK.pk) = m
  all m:Message,pk:PrivKey | dec.(enc.m.(createEK.pk)).(createEK.pk) = m
  all pk:PrivKey,c:Ciphertext | lone s:Signature | signing.c.(createSK.pk) = s
  all pk:PrivKey,s:Signature| lone c:Ciphertext | verifying.s.(createSK.pk) = c
  all c:Ciphertext,pk:PrivKey | verifying.(signing.c.(createSK.pk)).(createSK.pk) = c
}
//Checks if Encrypt-then-MAC is valid repeating test for encryption and signing
EncMacOK : check {
  all m1,m2:Message,k:PrivKey | m1 != m2 => enc.m1.(createEK.k) != enc.m2.(createEK.k)
  all m:Message,k1,k2:PrivKey | k1 != k2 => enc.m.(createEK.k1) != enc.m.(createEK.k2)
  all c1,c2:Ciphertext,k:PrivKey | c1 != c2 => dec.c1.(createEK.k) != dec.c2.(createEK.k)
  all c:Ciphertext,k1,k2:PrivKey | k1 != k2 => dec.c.(createEK.k1) != dec.c.(createEK.k2)
  all c1,c2:Ciphertext,k:PrivKey | c1 != c2 => signing.c1.(createSK.k) != signing.c2.(createSK.k)
  all c:Ciphertext,k1,k2:PrivKey | k1 != k2 => signing.c.(createSK.k1) != signing.c.(createSK.k2)
  all s1,s2:Signature,k:PrivKey | s1 != s2 => verifying.s1.(createSK.k) != verifying.s2.(createSK.k)
  all s:Signature,k1,k2:PrivKey | k1 != k2 => verifying.s.(createSK.k1) != verifying.s.(createSK.k2)
}

pred attack [ek:EncKey,sk:SignKey] {
  all m:Message,c:Ciphertext | enc.m.ek = c <=> dec.c.ek = m
  all c:Ciphertext,s:Signature | signing.c.sk = s <=> verifying.s.sk = c
}

run attack
```