

ETH zürich

Verification² of the Authentic Digital EMblem

Master's Thesis

Lasse Meinen

April 11, 2024

Advisors: Prof. Dr. Peter Müller, Linard Arquint, Felix E. Linker

Department of Computer Science, ETH Zürich

Abstract

A digital *emblem* is a cryptographically signed message that marks a set of assets as protected under International Humanitarian Law (IHL). Authorities such as nation-states create *endorsements* to attest to the authenticity of an emblem's signature; In turn, endorsements can themselves be endorsed. Assets actively distribute emblems and endorsements using UDP, TLS, or DNS. Parties can thus independently run the protocol's verification procedure, which determines an emblem's *security level* by asserting the existence of a valid chain of endorsements leading to a trusted party. [1]

The protocol has been modeled in Tamarin — a protocol verification tool used to automatically prove (or disprove) security properties for a given protocol model [2] — and a prototype has been implemented in Go. [3]

In this thesis, we formally verify the verification component of the an Authentic Digital EMblem (ADEM) codebase. That is, we verify memory safety (e.g. that there are no null-pointer dereferences), crash safety (e.g. no division by zero), and data race freedom of the implementation. Then, we use the methodology presented in [4], to show that the implementation is a refinement of the aforementioned Tamarin model. That is, the implementation inherits all security properties shown to be provided by the protocol's model.

Contents

Contents	iii
1 Introduction	1
1.1 Outline	2
2 Background	3
2.1 An Authentic Digital Emblem	3
2.2 ADEM implementation	4
2.3 Gobra	5
2.3.1 Locks	6
2.3.2 Channels	7
2.3.3 Waitgroups	7
2.4 Tamarin	8
3 Safety Verification	11
3.1 Verification keys and promises	12
3.2 Asynchronous token verification	14
3.3 Global parsing constraints	15
3.4 Set iteration	17
3.5 Validation constraints	18
3.6 Summary	20
4 Functional Verification	21
4.1 Tamigloo	22
4.1.1 I/O specifications	22
4.1.2 Model constraints	22
4.1.3 Term-level representation	24
4.2 Tamarin model	24
4.2.1 Changes to the model	25
4.3 I/O permissions	27

4.4	Internal permissions	29
4.4.1	Arbitrary numbers of input facts	29
4.4.2	Organizational endorsement chain	31
4.4.3	Library interfaces	33
4.5	Pattern requirements	33
5	Discussion	37
5.1	Verification overhead	37
5.2	Found bugs	38
5.3	ADEM security proof	40
5.4	Summary	41
6	Conclusion	43
6.1	Future work	43
A	Code excerpts	45
A.1	Generated I/O specifications	45
	Bibliography	47

Chapter 1

Introduction

In times of international or domestic armed conflict the Red Cross, Crescent, and Crystal have served as a means for protected parties, such as members of medical sectors and humanitarian organizations, to physically mark their personnel, facilities, or objects as protected under International Humanitarian Law (IHL). However, in recent years cyber operations have increasingly become part of armed conflict, moving conflict to a realm in which physical markings bear no meaning. The International Committee of the Red Cross (ICRC) therefore decided to investigate the idea of a ‘digital emblem’ — a digital equivalent of the physical markings. [5]

In partnership with the Centre for Cyber Trust (CECYT), the ensuing research project led to the development of an Authentic Digital EMblem (ADEM).

A digital *emblem* is a cryptographically signed message that marks a set of assets as protected under IHL. Authorities such as nation-states create *endorsements* to attest to the authenticity of an emblem’s signature; In turn, endorsements can themselves be endorsed. Assets actively distribute emblems and endorsements using UDP, TLS, or DNS. Parties can thus independently run the protocol’s verification procedure, which determines an emblem’s *security level* by asserting the existence of a valid chain of endorsements leading to a trusted party. [1]

The protocol has been modeled in Tamarin — a protocol verification tool used to automatically prove (or disprove) security properties for a given protocol model [2] — and a prototype has been implemented in Go. [3]

In this thesis, we formally verify the verification component of the ADEM codebase. That is, we verify memory safety (e.g. that there are no null-pointer dereferences), crash safety (e.g. no division by zero), and data race freedom of the implementation. Then, we use the methodology presented in [4], dubbed *Tamigloo*, to show that the implementation is a refinement of the aforementioned Tamarin model. That is, the implementation inherits all security properties shown to be provided by the protocol’s model.

Normally, Tamigloo shows a refinement by associating permissions with I/O operations: A protocol role can only perform a particular I/O operation when it holds the required permissions. Until now, these I/O operations have been understood to be network operations. However, in our case, the component under analysis receives its input from and sends its output to the command line. To that end, we assume that parts of the program's command line input, that is, the emblem and endorsements, are provided by the adversary and thus correspond to I/O operations. We make further assumptions regarding the communication between protocol parties, which are outlined in Sec. 4.2.1.

We evaluate the effectiveness and efficiency of our analysis. This includes collecting and evaluating verification time measurements, reasoning about the overhead in terms of lines of code introduced by formal code verification, and a depiction of our experience using Gobra and Tamigloo, including a summary of found bugs. We conclude that Tamigloo provides a way to bridge the gap between a highly abstract, formal model of a protocol and its concrete implementation. However, the approach is highly non-trivial and requires a deep understanding of both the model and the code. To illustrate, the model needed to be iterated multiple times until we could successfully relate it to the implementation. In our opinion, this reflects a discrepancy in the abstraction of a protocol model and the complexities of its real-world code implementation.

To summarize, we present the following contributions. First, as is standard, we verify the memory safety of the implementation. Second, we show that it is a refinement of the protocol's model, written in Tamarin, using the approach outlined in [4]. To apply the approach to our purposes, we make assumptions regarding the format of I/O operations and the communication between protocol parties. Finally, we evaluate our analysis and summarize our findings.

1.1 Outline

We proceed as follows. First, we summarize the required background knowledge to understand the results presented in this thesis (ch. 2). Next, we present our work and results relating to the safety verification (ch. 3) and refinement proof (ch. 4) of the implementation. Finally, we describe our evaluation (ch. ??) and conclusions (ch. 6), including suggestions for future work.

Chapter 2

Background

In this section, we present background on the ADEM protocol, and on used tools and methodologies.

2.1 An Authentic Digital Emblem

In ADEM a digital emblem marks a set of assets as protected. On a technical level, an emblem is a signed JSON Web Token (JWT). A JWT is an encoding of claims as a JSON object that is used as the payload of a JSON Web Signature (JWS) or as the plaintext of a JSON Web Encryption (JWE). [6] An emblem, or, more specifically, the public key used to create the emblem's signature, can be *endorsed*. An endorsement is used by party P_1 to attest that a particular public key K belongs to party P_2 and that P_2 may signal protection under IHL. Furthermore, an endorsement may introduce constraints regarding the issued emblems. For example, an endorsement can constrain the lifetime of an issued emblem or only allow a single website to be protected.

Note that, technically, a party can endorse itself. Such *internal endorsements* could, for example, be used to maintain more fine-grained control over which keys are used to protect which assets. A chain of internal endorsements eventually ends in a public key that is not further internally endorsed. Such a public key is called a *root key*. To provide accountability, a party can commit to a root key by encoding it as a Web PKI certificate and successfully submitting the certificate to Certificate Transparency (CT) logs. The certificate is signed by standard certificate authorities. The root key itself might still be endorsed by other parties. It is expected that most verifiers will only accept emblems endorsed by parties they trust. That is, there is a valid chain of endorsements, starting at the emblem, possibly including an endorsement of the Protected Party (PP)'s root key, and ending in the root key of a trusted party.

An agent that wishes to attack lawful targets will pay attention if an asset presents

a digital emblem to determine its protection status under IHL. We call such agents *verifiers*.

The verification of an emblem, and possibly an accompanying set of endorsements, returns a set of *security levels*. Depending on what additional information is available to a verifier, they may require varying levels of security to deem an emblem trustworthy. However, it is expected that in practice most verifiers will only accept emblems with the security level *endorsed-trusted*, which is returned only if there is a valid chain of endorsements ending in the root key of a party trusted by the verifier. The chain of endorsements cannot have any arbitrary shape: An emblem lists an issuing PP whose root key was, possibly indirectly through a chain of internal endorsements, used to create the emblem's signature, and only that root key can be endorsed by other authorities.

ADEM defines the following three security requirements, the latter two of which have been verified using the protocol verifier Tamarin [2].

- (1) *Covert inspection* requires that an agent who wishes to verify whether an asset is protected under IHL must be indistinguishable from agents who interact with that asset for other purposes.
- (2) *Verifiable authenticity* requires that agents must be able to correctly associate emblems to the issuing PP and the respectively marked assets.
- (3) *Accountability* requires that independent parties must be able to identify misbehaving parties. For example, parties might misbehave by issuing a fraudulent emblem, where a fraudulent emblem is to be understood as marking an asset not protected under IHL, akin to the illicit display of a physical red cross.

A prototype implementation of ADEM has been written in Go and can be found on GitHub [3]. It includes components for emblem and endorsement generation and verification, command line support, utilities for root key registration, and example usage snippets.

2.2 ADEM implementation

In this section, we present a brief, high-level overview of the ADEM codebase, and highlight some of its complexities.

The codebase is available open-source [3] and consists of 9 packages, 6 of which are relevant to the verification procedure. We treat the `VerifyTokens` function in `pkg/vfy/vfy.go` as the entry point for our verification. The function accepts two arguments: a slice of byte slices where each byte slice represents a single ADEM token, that is, an emblem or an endorsement, and a set of trusted signature verification keys. Consider the file `cmd/emblemcheck/main.go` for an example of how to use the function.

Upon its invocation, the `VerifyTokens` function stores all trusted public keys in the `keyManager`, a struct used to store and supply verified verification keys. Next, `VerifyTokens` starts a goroutine for each byte slice by calling the `vfyToken` function, giving each a reference to the `keyManager` and to a shared channel `results`, used to communicate verification results.

In short, the `vfyToken` function parses a provided token into a JWT and verifies its signature. Most of the parsing and signature verification is handled by a call to `jwt.Parse`, which is provided by the external `lesstrat-go/jws` library. The `FetchKeys` function in `key_manager.go` is invoked to provide the verification keys used in the signature verification.

If the token encodes a root public key, `FetchKeys` verifies that a corresponding, correctly-configured X.509 certificate is logged in the CT logs. Otherwise, `FetchKeys` will check if a corresponding endorsement of the verification key has been verified by another thread or if its verification key is contained in the set of trusted keys. If no verification key is available at the time of invocation, the function call will block until one becomes available. Only once these checks have been performed will the `FetchKeys` return a verification key to the `jwt.parse` function.

There are two important observations we make here. First, this implementation creates dependencies between threads: The verification of a token can only proceed when the verification of the preceding token has successfully terminated, or when its public key commitment has been verified. Second, we note that any verified token is transitively endorsed either by a root key with a corresponding public key commitment or by a trusted key.

If a JWT's signature has been verified, it is stored in an `ADEMTOKEN` struct, along with its verification key and metadata. Finally, the struct, or an error in the case of unsuccessful verification, is sent on the `results` channel.

After creating a goroutine for each byte slice, the `results` is used by the main thread to collect verification results. For each verified endorsement, the endorsed verification key is passed to the `keyManager`, allowing any waiting threads to continue execution.

The main thread checks the metadata of each collected token and splits them into endorsements and emblems. If it encounters multiple or no emblems, it terminates, outputting security level `INVALID`.

Finally, it proceeds to determine the emblem's security levels as described in Sec. 2.1.

2.3 Gobra

To further increase confidence in the aforementioned ADEM implementation, in particular the verification component, the implementation's core components are

```
1 requires acc(x) && acc(y)
2 ensures acc(x) && acc(y)
3 ensures val == *x + *y
4 func Sum(x *int, y *int) (val int) {
5     return *x + *y
6 }
```

Figure 2.1: Gobra code to verify a function `Sum` that returns the sum of the two values pointed to by `x` and `y`.

formally verified. To that end, we make use of Gobra [7], a modular, deductive program verifier for Go.

Gobra uses separation logic style field permissions [8]. In separation logic, field permissions specify the heap locations that a statement or an expression may access, enabling local reasoning about their effects on the heap. To that end, separation logic introduces an assertion language, encoded in Gobra as accessibility predicates. For example, `acc(p)` denotes permission to access a pointer `p`. Gobra encodes annotated Go programs into the intermediate verification language Viper. [9] Viper provides two verification backends, one based on symbolic execution and one based on verification condition generation. By default, Gobra uses the former. Either way, both backends ultimately use the SMT solver Z3 [10] to discharge proof obligations. Gobra supports many of Go's language features, including interfaces, structs, packages, and concurrent primitives such as *goroutines* and *channels*.

In Gobra, we annotate functions with pre- and postconditions. If the function is called in a state in which the precondition is satisfied, the postcondition must hold after it returns. This is proven modularly by only once verifying that the function body guarantees this. Consider the code in Fig. 2.1 for a simple example.

Gobra allows the verification of memory safety by verifying that there are sufficient permissions for all heap accesses. This also implies crash safety and the absence of data races. Afterward, user-provided specifications can be specified to verify the functionality of the program. All of the previous points will be considered in the verification of the ADEM codebase.

Next, we will briefly present how we can express permissions and invariants for relevant language features such as channels, waitgroups, and locks.

2.3.1 Locks

The Go standard library enables locking with `sync.Mutes`. It provides the standard operations associated with a mutex: `Lock`, `TryLock`, and `Unlock`. In Gobra, a mutex must be initialized with an invariant which is returned upon successfully acquiring the lock. This allows us to, for example, model the memory access permissions required for synchronized access to shared data.

Consider the invariant in Fig. 2.2. It shows a simplified version of the lock invariant used to provide memory permission to shared data structures in the `keyManager`. It is attached to a globally accessible lock upon initialization as `km.lock.SetInv(LockInv!<&km!>)` and is inhaled after a `km.lock.Lock()` statement.

```

1  pred LockInv(km *keyManager) {
2      acc(&km.keys) &&
3      acc(km.keys) &&
4      (forall k string :: k in km.keys ==> KeyMem(km.keys[k], k)) &&
5      acc(&km.listeners) &&
6      acc(km.listeners)
7  }

```

Figure 2.2: A simplified version of the invariant for the lock used to control shared access to the `keyManager`. It encodes memory permissions to the two globally accessible maps `km.keys` and `km.listeners`, and their elements.

2.3.2 Channels

The Go language provides another synchronization mechanism with channels, which allow the sending and receiving of values between threads. Channels can be buffered or unbuffered, and a channel can have multiple senders or receivers.

In Gobra, a channel is initialized with two invariants. First, a predicate of type `pred(T)`, where `T` is the type of the value sent on the channel. If a sender wishes to send a value `v`, an instance of `pred(t)` is exhaled at the sending site. An identical instance is inhaled at the receiver site. Second, a complementary predicate with arity zero. It is exhaled at the receiving site and inhaled at the sending site.

The channel initialization of a channel `c` returns permissions `c.SendChannel()` and `c.ReceiveChannel()`, read fractions of which must be held in order to send to or receive from `c`, respectively. To close a channel, full write permission to `c.SendChannel()` must be exhaled.

The predicate in Fig. 2.3 shows the send invariant used in the `NewPromise` function, which returns a new instance of type `Promise`. The `Init` statement in line 3 initializes the channel `p.ch` correspondingly, with `PredTrue` as the receive invariant.

2.3.3 Waitgroups

`sync.WaitGroup`, also included in the Go standard library, provides a mechanism to wait for a collection of goroutines to finish. For a waitgroup `wg`, the size of the collection is specified by calling `wg.Add(n)` for some integer `n`. `wg.Done()` decrements the counter by one, and `wg.Wait()` blocks until the counter reaches zero.

```

1 func NewPromise() Promise {
2   p := promise{ch: make(chan jwk.Key, 1)}
3   p.ch.Init(SendInv!<_!>, PredTrue!<!>)
4   ...
5   return &p
6 }

7 pred SendInv(val jwk.Key) {
8   acc(val.Mem(), _)
9 }

```

Figure 2.3: The predicate `SendInv` is specified as the send invariant of the channel underlying the `Promise` interface. It encodes a read fraction of the memory permission of a value of type `jwk.Key`.

In Gobra, a waitgroup, similar to locks and channels, must be initialized. An initialization statement `wg.Init()` returns a permission `wg.WaitGroupP()`, which is required to perform any `Wait` or `Add` operations. An add statement `wg.Add(n)` returns `n` instances of a `wg.UnitDebt(PredTrue!<!>)` predicate, one instance of which is consumed in the precondition of `wg.Done()`.

Furthermore, Gobra allows for more complex definitions of `UnitDebt(P)` predicates (Fig. 2.4). In rough terms, a goroutine must first pay its debt by providing an instance of `P` along with `UnitDebt(P)`. The two predicates are consumed in the precondition of `wg.PayDebt()`. In return, an instance of `UnitDebt(PredTrue!<!>)` is produced, allowing for the goroutine to call `wg.Done()`. Along with the more complex debt predicate, a token predicate `Token(P)` may be generated. These are exchanged for instances of `P` in the postcondition of `wg.Wait()`.

```

1 km.init.Init()
2 km.init.Add(numThreads, 1/2, PredTrue!<!>)
3 invariant 0 <= i && i <= numThreads
4 invariant acc(km.init.WaitGroupP(), 1/2) && !km.init.WaitMode()
5 invariant acc(km.init.UnitDebt(PredTrue!<!>), numThreads - i)
6 invariant acc(km.init.UnitDebt(WaitInv!<!>), perm(i))
7   && acc(km.init.Token(WaitInv!<!>), perm(i))
8 for i := 0; i < numThreads; i++ {
9   km.init.GenerateTokenAndDebt(WaitInv!<!>)
10 }
11 km.init.Start(1/2, WaitInv!<!>)
12 km.init.SetWaitMode(1/2, 1/2)

```

Figure 2.4: The `km.init` waitgroup is used to ensure that all signature verification threads (Sec. 2.2) have obtained a `jwk.Key` promise before `VerifyTokens` begins to collect results. It is initialized to a counter of `numThreads`. For each thread, we generate an instance of `UnitDebt(P)` and `Token(P)`, where `P` is set to the `WaitInv` predicate.

2.4 Tamarin

The ADEM protocol has been modeled using the protocol verifier Tamarin [2]. Tamarin is a protocol verification tool used to automatically prove (or disprove)

trace-based security properties for a given model. A Tamarin model is represented as a multiset rewriting system (MRS) consisting of rules and an equational theory. The rules typically represent the steps taken by a protocol participant or the attacker, and the equational theory encodes message semantics.

Protocol messages are modeled as ground instances of terms from a term algebra $\mathcal{T} = \mathcal{T}_{\Sigma}(\mathcal{N} \cup \mathcal{V})$. \mathcal{T} is built over a signature Σ of function symbols, a set of fresh and public names \mathcal{N} , and a set of variables \mathcal{V} . That is, cryptographic messages \mathcal{M} are elements of \mathcal{T} without any variables. Terms are given semantic meaning with an equational theory E , where we denote equality modulo E by $=_E$.

Facts are atomic predicates applied to message terms, modeling partial state. They are constructed over a fact signature $\Sigma_{facts} = \Sigma_{lin} \uplus \Sigma_{per}$, partitioned into linear, single-use facts, and persistent facts.

A MRS \mathcal{R} consists of rewriting rules $l \xrightarrow{a} r$ which act on the global protocol state \mathcal{S} , that is, on multisets of ground facts. A rule is applicable if $l \subseteq^m \mathcal{S}$, where \subseteq^m expresses the subset relation on multisets, and updates the protocol's state by removing any linear facts in l from \mathcal{S} and adding any facts in r . Facts in a are called actions and constitute the modeled protocol's state, labeling a sequence of transitions.

A MRS \mathcal{R} together with an equational theory E constitutes a labeled transition system (LTS) acting on the global protocol state \mathcal{S} , starting from the empty state \square .

maybe add an example? Should I also introduce In, Out and Fr facts, or maybe just more notation in general?

Chapter 3

Safety Verification

The goal of this thesis is to build confidence in the ADEM codebase by formally verifying its verification component. As is standard, we first establish memory safety before proving functional properties.

In this chapter, we start by explaining what safety verification entails and its relevance to the implementation of security protocols. We've given a high-level overview of the ADEM codebase and highlighted some of its complexities in Sec. 2.2. Here, we delve deeper and provide in-depth descriptions of technical details and challenges of the performed safety verification.

Proving a program's memory safety shows that memory accesses are valid thus preventing issues such as buffer overflows, race conditions, and other memory-related bugs. It is distinct from *functional* verification which reasons about a program's correctness in the context of a user-provided specification. Note, however, that functional properties may often be necessary to prove memory safety, for example, to show that array accesses are within bounds.

In the context of security protocol implementations, memory safety verification is paramount. Memory-related bugs in security protocol implementations can lead to severe vulnerabilities, as was showcased, for instance, by the infamous Heartbleed vulnerability in the OpenSSL library. [11] Despite their notoriety, memory safety vulnerabilities continue to be among the most prevalent known exploited vulnerabilities, as illustrated by the CWE Top 10, which lists the most exploited vulnerabilities. [12]

To illustrate how memory safety verification works in practice, consider the code snippet in Fig. 3.1 which contains a trivial race condition.

The function `foo` starts 1000 goroutines, each of which increments the same heap location by one. The goroutine's implementation is in the function `bar`. The increments of the shared variable `counter` are not synchronized in any way and therefore may result in a race condition.

```

1 func bar(counter *int) {
2     // Unsafe concurrent write access
3     *counter++
4 }

5 func foo() {
6     counter := 0

7     for i := 0; i < 1000; i++ {
8         go bar(&counter)
9     }
10 }

```

Figure 3.1: A simple code snippet demonstrating improper synchronization: Multiple goroutines may attempt to write to `counter` concurrently.

```

1 requires acc(counter)
2 func bar(counter *int) {
3     *counter++
4 }

5 func foo() {
6     counter@ := 0

7     invariant acc(&counter, (1000 - i) / 1000)
8     for i := 0; i < 1000; i++ {
9         // precondition of call might not hold.
10        go bar(&counter)
11    }
12 }

```

Figure 3.2: Now annotated with memory access permissions, Gobra will tell us that the precondition of `bar` may not hold. In this case, the permission to `counter` may not suffice.

We might annotate the code with permissions as shown in Fig. 3.2. We see that `bar` requires write access permission to the `counter` variable through its precondition. Gobra tells us that the call to `bar` may fail, as the permission to `counter` may not suffice. While trivial, this example demonstrates how permissions are a way of tracking ownership of heap locations, ensuring that either multiple goroutines can concurrently read or a single one can write. This allows us to effectively rule out data races.

3.1 Verification keys and promises

In Sec. 2.2, we describe that verification of an endorsement may block until a preceding verification key becomes available. The codebase uses *promises* to await verification of a particular key by another goroutine.

In programming, a promise is a structure that represents the eventual completion of an asynchronous operation. Typically, a promise is in one of three states: pending, fulfilled, or rejected. The Go standard library does not provide promises; instead, an implementation is included in the codebase. Here, promises are implemented on top of channels, providing the following three methods:

- `Fulfill`: Fulfills the promise by sending a value of type `jwk.Key` and subsequently closing the channel.
- `Reject`: Rejects the promise by closing the channel without sending a value.
- `Get`: Awaits the completion of the promise by blocking on the underlying channel. If the promise is fulfilled, the call receives a non-nil value `v` on the channel. If the promise is rejected, it receives a `nil` value `v`. In either case, `Get` unblocks and returns `v`.

Goroutines that are waiting to receive a particular verification key create a promise and append it to a slice of promises in the `keyManager`, all of which are waiting for that same key.

There are two complications to note in this setup. First, the same verification key is sent to an arbitrary number of goroutines and each corresponding promise must produce a non-zero fraction of that key's access permission so that it may later be used for signature verification. Second, the channel is closed after a single send operation. Note that, as sending on a closed channel results in a panic, closing a channel consumes full send permission to that channel. Consequently, the `Fulfill` and `Reject` methods must require full send permission to the channel as a precondition.

To resolve the first complication, we specify a wildcard permission, instead of a full permission, to the sent value as a send invariant on the promise's underlying channel (Fig. 3.3). Access permissions to values of type `jwk.Key` are specified as an abstract predicate `Mem`.

```

1  ensures res != nil && res.ProducerToken() && res.ConsumerToken()
2  func NewPromise() (res Promise)

3  requires p.ProducerToken()
4  requires acc(val.Mem(), _)
5  func (p *promise) Fulfill(val jwk.Key)

6  requires p.ProducerToken()
7  func (p *promise) Reject()

8  requires p.ConsumerToken()
9  ensures res != nil ==> acc(res.Mem(), _)
10 func (p *promise) Get() (res jwk.Key)

```

Figure 3.3: Specifications for the functions to create, fulfill, reject and await (`Get`) a promise. We omit the function bodies here for brevity.

We tackle the second complication by creating high-level predicates `ProducerToken` and `ConsumerToken` that contain all necessary permissions to send and receive on the channel, respectively. Consider the specifications in Fig. 3.3. An instance of each predicate is created upon the instantiation of a promise. The `Fulfill` and `Reject` methods require a `ProducerToken` in their precondition, while

the `Get` method requires a `ConsumerToken`. Thereby, we ensure that no value is ever sent on a promise's closed channel.

Importantly, this implies that a sending goroutine requires full access to the `ProducerToken` predicate. We achieve this by requiring that for any unresolved promise in the `keyManager`, a corresponding `ProducerToken` must be available. This is ensured by encoding corresponding permissions in the invariant associated with the shared lock `keyManager.lock`. We do not need to do the same for `ConsumerToken` predicates, as they are only ever held locally by threads that created a corresponding promise.

3.2 Asynchronous token verification

An invocation of `VerifyTokens` starts `n` verification threads, one for each token in its input. Each thread is in charge of verifying exactly one token. As mentioned in Sec. 2.2, the verification results are communicated via the `results` channel, that is, a channel with exactly `n` senders and a single receiver. The sent values are structs of type `VerificationResult`, which has two fields: a field `Token` of type `*ADEMToken` and a field `err` of type `error`. A non-nil `err` corresponds to a failed verification. The main thread waits to receive exactly `n` results, after which it closes `results`.

To send its verification result to the `results` channel, a goroutine requires a non-zero fraction of the `results.SendChannel()` predicate, representing the permission for being able to send on the channel. Having send permission to a channel guarantees that sending on that channel does not panic. However, to close the channel, the main thread requires full permission to that predicate and, thus, needs to collect and combine each goroutine's fractional predicate instance. As the channel send operation is a verification thread's final operation before terminating, there is a priori no obvious way for the verification threads to transfer their send permission fractions to the main thread.

To address this issue, one could adapt the specification of channels to allow the transfer of the `SendChannel` predicate as part of its send invariant. However, such an adaptation would be highly non-trivial and we decided to adopt another approach.

We address this issue by introducing a waitgroup to explicitly reflect that the main thread waits to receive a value from every goroutine before closing the channel. By requiring that every thread sends exactly one result, and calls the waitgroup's `Done` method immediately after that send operation, that is, without any further modifications of the shared state, we ensure that this waiting behavior is already achieved by the channel receive operations. Therefore, the waitgroup does not introduce any additional synchronization behavior and we can mark it as `ghost`.

The waitgroup consumes a verification thread's send permission fraction through its `UnitDebt` debt predicate. Send permissions are wrapped in a high-level predicate `SendFraction`, shown in Fig. 3.4. An instance of `SendFraction` must be consumed by the waitgroup's `PayDebt` method, producing an instance of `UnitDebt(PredTrue!<!>)` before a thread can call `Done` (Fig. 3.5).

```

1 pred SendFraction(results chan *TokenVerificationResult, n int) {
2     0 < n && acc(results.SendChannel(), 1 / (2 * n))
3 }

```

Figure 3.4: The `SendFraction` predicate contains a channel send permission fraction of size $1 / (2 * n)$, where n is the number of verification threads. A fraction `acc(results.SendChannel(), 1 / 2)` is held by the main thread.

```

1 requires ...
2 requires SendFraction!<results, threadCount!>()
3 requires vfyWaitGroup.UnitDebt(SendFraction!<results, threadCount!>)
4 func vfyToken(
5     rawToken []byte,
6     results chan *TokenVerificationResult,
7     ghost vfyWaitGroup *sync.WaitGroup,
8     ...
9 ) {
10     result@ := TokenVerificationResult{}
11     ...
12     unfold SendFraction!<results, threadCount!>()
13     results <- &result
14     fold SendFraction!<results, threadCount!>()
15     vfyWaitGroup.PayDebt(SendFraction!<results, threadCount!>)
16     vfyWaitGroup.Done()
17 }

```

Figure 3.5: A verification thread must provide its `results` send permission fraction to increment the waitgroup's counter.

Once the main thread has received n results, it calls `vfyWaitGroup.Wait`, passing the sequence of all expected send fractions as an argument. Afterward, it will have full send permission to the `results` channel and can therefore safely call `close`.

3.3 Global parsing constraints

In ADEM, a token, that is, an emblem or endorsement, is encoded as a JWS or, in case the signature is omitted, as an unsigned JWT. [1][6] A payload consists of registered and unregistered, that is, custom, claims. ADEM tokens are parsed via the external `lestrrat-go/jwx` library, which is used throughout the project to handle JWT-related functionality such as parsing and signature verification. Custom JWT claim definitions are registered by calling the `RegisterCustomField` function. Such a registration specifies that a claim name will be decoded as a particular type. If a claim name cannot be decoded to its specified type, parsing fails, thereby providing custom parsing behavior and type guarantees. Importantly,

this has a global effect.

The `tokens` package, as part of its initialization logic, registers multiple custom claim names. Therefore, thanks to the guarantees provided by the parsing, given a parsed JWT token `t` and a custom claim name `key`, any later call to `t.Get(key)` returns values of a particular type. Thus, we want to leverage this information for verification, which requires us to keep track of which custom field maps to which type and suitable specification stating that we obtain permissions for the value returned by `Get` (Fig. 3.6).

```

1  jwt.RegisterCustomField("key", EmbeddedKey{})
2  ...
3  token, err := jwt.Parse(someBytes)
4  if err != nil {
5      return
6  }
7  if k, ok := token.Get("key"); ok {
8      key := k.(EmbeddedKey).Key
9  }

```

Figure 3.6: An example usage of custom claims. "key" is registered as a custom claim name with type `EmbeddedKey`. Therefore, thanks to the guarantees provided by the successful parsing of the bytestring `someBytes`, the type assertion `k.(EmbeddedKey)` will not trigger a panic.

In other words, throughout the program, we must propagate a correspondence between claim names and types, and we must be able to correctly determine the memory permissions to those claims. Note that it is possible to call `t.Get(key)` multiple times for the same `t` and the same `key`. In addition, in some cases, the ADEM implementation may require write permission to a value returned by `Get` and distribute that value to multiple goroutines, thereby only retaining a fractional read permission, but continue to use the underlying token, for example, in later `Get` calls. Inspecting the library's implementation reveals that multiple such calls even return references to the same heap location. [13] This complicates permission reasoning significantly. For example, a solution that simply provides a wildcard read permission `acc(v.Mem(), _)` to a value `v` after a call to `Get` would not suffice. We conclude that we must be able to represent the internal memory of a token.

We solved this problem by introducing a global map `Custom` as part of the library stub. It maps a field name to an interface of type `JwtClaim`, enabling the expression of more complex memory permissions for concrete type implementations of `JwtClaim`. The interface `JwtClaim` is empty except for a member `pred Mem()`, representing the memory permission to a returned value. The map `Custom` is initially empty and is modified by calls to the library's `RegisterCustomField` function. The function ensures a mapping from a field name to a type is stored as expected. We propagate the constraints introduced during initialization as a pure ghost function `CustomFields` (Fig. 3.7). Any later call to `Get` requires read permission to `Custom` to ensure that the returned value has the same type as the value stored in `Custom`

if the key is present in both the token and the map.

```

1  ghost
2  requires acc(f, _)
3  pure func CustomFields(f jwt.Fields) bool {
4      return (
5          domain(f) == set[string] {
6              "log", "key", "ass", "emb", "ver"
7          } &&
8          typeOf(f["log"]) == type[[]*LogConfig] &&
9          typeOf(f["key"]) == type[EmbeddedKey] &&
10         typeOf(f["ass"]) == type[[]*ident.AI] &&
11         typeOf(f["emb"]) == type[EmblemConstraints] &&
12         typeOf(f["ver"]) == type[string])
13 }

```

Figure 3.7: The package initialization of tokens populates the `jwt` library's Custom map. We keep track of these mappings throughout the program by using the ghost pure function `CustomFields`.

```

1  pred FieldMem(ghost fields dict[string]JwtClaim) {
2      forall k string :: { fields[k] } k in domain(fields)
3          ==> fields[k].Mem()
4  }
5
6  // JwtToken represents a generic JWT token.
7  type JwtToken interface {
8      pred Mem()
9
10     ghost
11     requires acc(Mem(), _)
12     decreases _
13     pure Values() (ghost r dict[string]JwtClaim)
14     ...
15 }

```

Figure 3.8: For a token `t` of type `JwtToken` token, we summarize its fields' memory permissions in a predicate instance `FieldMem(t.Values())`.

Due to the possibility of multiple, subsequent `Get` calls, we decided to omit any memory permissions to the returned field value from the postcondition of `Get`. Instead, we opted to keep these separate in a `FieldMem` predicate, allowing more fine-grained and transparent memory permission specification. We couple the `FieldMem` predicate to a token by introducing another ghost pure function `Values()`, which returns a mathematical map of the fields stored in the token, as shown in the code snippet in Fig. 3.8. An instance of a `FieldMem` predicate is returned when a token is first instantiated by `jwt.Parse`.

3.4 Set iteration

The `VerifyTokens` function accepts a set of trusted keys of type `jwt.Set` as its second input argument. `jwt.Set` is, as the name implies, a set-like datastructure that stores values of type `jwt.Key`. `jwt.Set` offers standard insertion and

lookup functionality in addition to a method `Keys`, which returns an iterator of type `arrayiter.Iterator` over the set's elements.

At this point, it is important to note two things. First, both the lookup operation and the set iteration return references to the stored keys. The permission to access the heap location referenced by a value of type `jwt.Key` is represented by an abstract, high-level predicate `Mem()`. Second, the described operations can be arbitrarily interleaved. In particular, it is possible to create multiple set iterators at a time. Therefore, we need to be able to reason about access permissions of a set's elements in the presence of such interleavings where multiple references to the same heap location may exist simultaneously.

Similar to the problem presented in Sec. 3.3, this leaves us with a situation in which a naive solution, such as providing a wildcard read permission to each iterated key, would not suffice.

To that end, we introduced two proof utilities to the `Iterator` interface: A pure method `GetIterSeq`, which returns a mathematical list representing the sequence of elements that are iterated, and a pure method `Index`, which returns an index into the sequence returned by `GetIterSeq`. In addition, we add a pure ghost function `Elms()` to the `jwt.Set` interface, which returns a mathematical sequence representing the contents of the set. This allows us to constrain the elements of an iterator created by a call to `s.Keys()`, for `s` of type `jwt.Set`, to be equal to the elements in `s`.

In addition, we wrap access permissions to the keys in a set `s` in a predicate `KeySeq(s.Elms())`. That is, the memory permissions to the set's keys are contained in `KeySeq`, while the methods of `jwt.Set` and `arrayiter.Iterator` only return information that a key is contained in the sequences returned by `Elms()` resp. `GetIterSeq()` (Fig. 3.9).

3.5 Validation constraints

After all signatures have been verified and the tokens collected, `VerifyTokens` splits the tokens into endorsements and emblems, and discards any that do not fit in either category. Before determining an emblem's security levels, each token is validated against a set of constraints, differing depending on whether the token is an endorsement or an emblem. This ensures that a token's payload constitutes a valid ADEM token. For example, an emblem's payload is required to include the "ass" claim, which lists the asset identifiers that are marked as protected. Any tokens that fail the validation check are discarded. Naturally, any knowledge acquired as a result of a successful token validation is later propagated through the proof.

Similarly to the `FetchKeys` method described in Sec. 3.1, the constraint validation is handled by implementing the `jwt.Validator` interface, once as an

```

1 // Set represents JWKS object, a collection of jwk.Key objects.
2 type Set interface {
3     pred Mem()
4
5     ghost
6     requires acc(Mem(), _)
7     pure Elms() (r seq[Key])
8
9     requires acc(Mem(), _)
10    ensures res != nil && res.IterMem() &&
11        res.GetIterSeq() == Elms() && res.Index() == 0
12    Keys(context.Context, ghost p perm) (res KeyIterator)
13
14    ...
15 }
16
17 pred KeySeq(ghost keys seq[Key]) {
18     forall i int :: { keys[i].Mem() } 0 <= i && i < len(keys)
19     ==> keys[i].Mem()
20 }

```

Figure 3.9: We keep the keys' memory permissions separate from set and iterator methods to facilitate reasoning in the presence of arbitrary interleavings. They are wrapped in a predicate `KeySeq(s.Elms())`, where `s` is a set of type `jwk.Set`.

`EmblemValidator` and once as an `EndorsementValidator`, and passing a corresponding instance as a validation option, in this case to the `jwt.Validate` function.

We extend the `Validator` interface by a predicate `Constraints(jwt.Token)`. A successful execution of `jwt.Validate` returns an instance of the predicate for the validated token (Fig. 3.10). Unfolding the predicate provides us with the necessary constraints to verify the safety of the code snippet in Fig. 3.11.

```

1 pred (EmblemValidator) Constraints(t jwt.Token) {
2     ...
3     t.Contains("ver") &&
4     t.Values()["ver"] == string(consts.V1)a &&
5     t.Contains("ass")
6 }

```

Figure 3.10: The implementation of the `Constraints` predicate for the `EmblemValidator`. `jwt.Validate` returns an instance of this predicate as a postcondition if the validation of an emblem succeeds.

```

1 if err := jwt.Validate(
2     emblem.Token, jwt.WithValidator(EmblemValidator)
3 ); err != nil {
4     return ResultInvalid()
5 }
6 ass, _ := emblem.Token.Get("ass")
7 protected = ass.([]*AI)

```

Figure 3.11: After validating a token, we provably know that the custom key "ass" is contained in the token's payload and that it is of type `AI`. Therefore, the call `Get("ass")` is guaranteed to succeed and return a value of type `AI`.

3.6 Summary

To summarize, we proved memory safety for the verification component of the ADEM codebase. That is, we showed memory safety (e.g. that there are no null-pointer dereferences), crash safety (e.g. no division by zero), and data race freedom of the implementation.

Besides the soundness of the program verifier, our proofs rely on the correctness of the dependencies' specifications as we do not verify the dependencies. We mitigate this assumption by inspecting the dependencies' implementation.

In addition, we do not guarantee that the implementation terminates as we proved partial correctness. That is, if the program terminates, it will satisfy the post-condition. The reason we do not show termination is, among other reasons, a consequence of the frequent use of channels: Gobra currently does not support verifying that a channel send or receive operation terminates.

Chapter 4

Functional Verification

In this chapter, we build on the memory safety verification presented in chapter 3 to verify functional properties about ADEM’s verification component. We begin by briefly explaining what functional verification is, and how it relates to the verification of security properties in protocols and ADEM in particular. Then, we provide more details on Tamigloo [4] and describe in depth how we applied it to ADEM. We then proceed to highlight some of the main challenges we encountered, and how we solved them. Finally, we summarize what we proved, and under which assumptions.

Functional verification, complementary to memory safety verification, reasons about a program’s properties, which are typically expressed as a user-provided specification. That is, it is the verification of the program’s intended functionality. In the context of security protocols, this means verifying that an implementation is faithful to the underlying protocol and provides the desired security properties.

Arquint et al. [14] present a trace-based approach by modeling a global trace of the protocol execution. Security properties are expressed as invariants on the global trace. This methodology allows the verification of security properties directly at the level of the implementation. However, we deemed this approach impractical in our case, as an abstract model and its desired security properties already exist.

Alternatively, Arquint et al. [4] present a methodology to bridge the gap between an abstract security model and its implementation. The methodology is commonly dubbed *Tamigloo*. For convenience, we use that name here.

Tamigloo extracts a specification of I/O operations from a given Tamarin model. Proving that an implementation satisfies this specification guarantees that the implementation performs only the I/O operations specified by the Tamarin model. An implementation that satisfies the I/O specification is a refinement of the Tamarin model in terms of trace inclusion and, therefore, inherits any trace property shown to hold for the model. [4] In particular, it inherits security properties such as authenticity and accountability.

The latter approach comes with several advantages over the former. First, it requires the expression of strong enough invariants that it is possible to verify security properties. Finding such invariants is non-trivial. Tools like Tamarin do not require such invariants, as they explore all possible traces. Second, the ADEM protocol and its security properties have been modeled in Tamarin. Finally, Tamigloo gives us clear proof obligations under which the properties proven in Tamarin carry over to the implementation.

Therefore, we adopt the methodology presented in [4] to prove that the ADEM implementation is a refinement of the Tamarin model.

4.1 Tamigloo

Here, we present the required background on I/O specifications as presented by Penninckx et al. [15], the transformation steps performed by Tamigloo to obtain an I/O specification, and the verification of the adherence to an I/O specification.

4.1.1 I/O specifications

Penninckx et al. [15] present the idea of an I/O specification. Each I/O operation is associated with an I/O permission which it consumes as its precondition. Necessary I/O permissions are granted by an I/O specification which is inhaled as the precondition of a program's main function.

More concretely, an I/O operation io requires an abstract predicate $io(p_1, \bar{v}, \bar{w}, p_2)$, expressing the permission to execute io with outputs \bar{v} and inputs \bar{w} . The operation io is said to move a token from a place p_1 to another place p_2 , encoded as the consumption of the abstract predicate $token(p_1)$ and production of $token(p_2)$. The movement of the token induces an ordering of I/O permissions and therefore of permitted I/O operations.

In addition to I/O permissions, it is possible to include permissions for internal operations, that is, operations that do not have an observable I/O effect but change the internal state of a protocol participant. This can be used to enforce particular internal operations between I/O operations.

An I/O specification is encoded as the separating conjunction of I/O and internal permissions with the same source place, thereby allowing for a non-deterministic choice between permissions. Each permission co-recursively produces an updated version of the I/O specification, thereby allowing non-terminating sequences of I/O operations.

4.1.2 Model constraints

Tamigloo generates a set of I/O specifications, one for each protocol role, from a Tamarin protocol model by successively applying multiple transformation steps.

First, the model's multiset rewriting system (MRS) is decomposed into several component models, one for each role and one for the environment, thereby moving from a global view of the protocol to a local view of each role. Next, each component model is transformed into an I/O specification, expressing the permitted I/O and internal operations for this particular component, and encoded according to the syntax of a particular program verifier.

To enable these transformation steps, Tamigloo makes multiple formatting assumptions on the Tamarin model. We highlight those relevant to our purposes here and refer to the original publication for details. [4]

First, for an n -role protocol, the model's fact signature must consist of mutually disjoint sets of action facts, environment facts, and each role i 's state facts.

$$\Sigma_{facts} = \Sigma_{act} \uplus \Sigma_{env} \uplus \left(\biguplus_{1 \leq i \leq n} \Sigma_{state}^i \right)$$

where Σ_{env} contains two disjoint subsets Σ_{in} and Σ_{out} of input and output fact symbols, and Σ_{in} contains an initialization fact symbol $Setup_i$ for each role i .

Next, the MRS R must consist of pairwise disjoint rule sets of environment and protocol rules:

$$R = R_{env} \uplus \left(\biguplus_{1 \leq i \leq n} R_i \right)$$

For all rules $l \xrightarrow{a} r \in R_{env}$, agents' internal states may not be directly used, that is, $facts(l \cup r) \subseteq \Sigma_{env}$. Furthermore, protocol rules must use input and output facts to communicate with the environment. That is, for all $l \xrightarrow{a} r \in R_i$, $facts(l) \subseteq \Sigma_{state}^i \cup \Sigma_{in}$ and $facts(r) \subseteq \Sigma_{state}^i \cup \Sigma_{out}$. In addition, at least one state fact must appear in r , and the first $k_i \geq 1$ arguments of all state facts must be the same, representing the parameters of the run of the protocol role. In our case, the latter restriction is limited to the inclusion of a fresh thread identifier rid as the first argument of all state facts.

A Tamarin model adhering to these constraints is transformed and decomposed to obtain MRS R_{env}^e for the environment and R_{role}^i for each protocol role i . We refer the reader to the original publication for the details of the transformation steps.

For each protocol role i , an I/O specification ψ_i is extracted from R_{role}^i :

$$\psi_i(rid) = \exists p.token(p) \star P_i(p, rid, [])$$

where rid is the run identifier, denoting the identifier of a particular role's instance, p is some starting place, and the model state is initialized to the empty multiset.

For each rewrite rule $R \in R_{role}^i$, an application of R to the model state S is encoded as a formula φ_R . It provides the permission $R(p, \dots, p')$, which a program must hold to execute the internal or I/O operation R . P_i is defined as the separating conjunction over all φ_R for $R \in R_{role}^i$:

$$P_i(p, rid, S) = \star_{R \in R_{role}^i} \varphi_R(p, rid, S)$$

φ_R co-recursively provides $P_i(p', rid, S')$ with target place p' and updated state S' .

An I/O specification ψ_i can be encoded into any program verifier that supports separation logic and abstract predicates. We provide a sample of an I/O specification for Gobra in App. A.1.

4.1.3 Term-level representation

Tamarin (Sec. 2.4) and an I/O specification ψ_i operate on message terms $m \in \mathcal{M}$, which are ground instances of terms, while the code operates on concrete messages from bytestring algebras \mathcal{B} . Tamigloo bridges this gap through the use of a surjective homomorphism $\gamma : \mathcal{M} \rightarrow \mathcal{B}$. Importantly, note that γ is *not* injective. This makes reasoning that a particular term $m \in \mathcal{M}$ matches a pattern $t \in \mathcal{T}$ more challenging.

For example, through successive internal parsing operations, we may find that a concrete value $\gamma(m)$ equals a concrete instance of the pattern $\gamma(t\sigma)$. As γ is not injective, this does not imply that $m = t\sigma$. Therefore, we cannot immediately conclude that a rewriting rule operating on t is applicable.

The *pattern requirement* for a pattern $t \in \mathcal{T}$ expresses that instances of top-level protocol messages we receive and parse have unique term representations, that is, no two terms concretize to the same byte-level representations:

$$\gamma(t\sigma) = \gamma(m) \Rightarrow \exists \sigma'. m =_E t\sigma'$$

Arquint et al. [4] show that image disjointness and pattern injectivity for a pattern t imply its pattern requirement.

4.2 Tamarin model

The ADEM paper [1] formalizes the ADEM protocol in the Tamarin protocol verifier. We provide a brief overview here of the original model.

The author models the protocol participants as certificate authorities (CAs), CT logs, authorities, and PPs. The protocol consists of two phases. The aforementioned

protocol participants run the first phase of the protocol: domain and subdomain names are non-deterministically assigned to parties with corresponding Transport Layer Security (TLS) keys and certificates, and ADEM parties request any necessary Web PKI certificates to commit to their root keys. In a second phase, a verifier, corresponding to the component we analyze in this work, receives an emblem and a set of endorsements, and verifies them as described in Sec. 2.1. The endorsements are received in a non-deterministic loop, corresponding to an arbitrary number of iterations, thereby letting Tamarin exhaustively inspect all possible sets of endorsements.

The model includes lemmas to prove the executability of the model, and authentication and accountability security properties as defined in the ADEM paper. To enable the automation of the proofs of the specified properties, the model employs several abstractions. Importantly, the model neither includes internal endorsements — instead assuming that the organizational root endorsement directly endorses the emblem — nor endorsement constraints.

4.2.1 Changes to the model

To prove that the verifier implementation is a refinement of the Tamarin model using Tamigloo, we modified the Tamarin model in multiple ways. We report next on the particular modifications and the reasons why these modifications are necessary.

Output operations

To be able to apply Tamigloo in a meaningful way, we require that the output operations performed by the implementation, that is, the output of achieved security levels to the console, are reflected in the Tamarin model. To that end, we modify the model such that its rewriting rules produce `Out` facts for all achieved security levels (Fig. 4.1).

Syntax constraints

We ensured that the model is compatible with the syntactic constraints presented in Sec. 4.1.2. This involved the introduction of state facts and corresponding setup rules. As we limit our analysis to the verifier component of the implementation, we did not introduce roles for parties involved in the first phase of the protocol, that is, CAs, CT logs, authorities, and PPs. Instead, we consider the corresponding rewriting rules to be part of the environment.

Concurrent token verification

Recall from Sec. 4.1.1 that the I/O permissions' place arguments induce an ordering of permitted I/O operations. While the original paper by Penninckx et al. also introduces split and join operators to enable concurrent operations, it is not

clear how one would extract an I/O specification with concurrency from a Tamarin model. Thus, Tamigloo emits a sequential I/O specification.

As described in Sec. 3.2, the implementation performs the tokens' signature verification in parallel. Importantly, each thread may need to perform multiple network requests to verify root key commitments. In Tamarin, these network requests are modeled as multiple `In` facts, consumed by a single rewriting rule.

A sequential I/O specification can be used for a concurrent program if it is shared among the threads, for example, via a concurrency primitive, or if we assume that I/O operations happen atomically via a shared region. Both require however that an invariant can be specified about the I/O specification and its corresponding model state. As previously described, a thread may perform multiple I/O operations before re-establishing a model state. In other words, before being able to reestablish any invariant on the model state, a thread must go through multiple transitions. Therefore, to maintain an invariant on the model state, the threads' sequences of network operations must conceptually be modeled as critical sections.

To model this requirement, a synchronization primitive must be introduced. There are two options. First, one could introduce a lock, thereby modifying the code's runtime behavior. Alternatively, one could introduce a *ghost* lock. However, this requires that we prove that its erasure does not influence the code's execution, that is, the ghost lock's critical section is performed atomically. In this case, we could conclude that the ghost lock does not remove possible thread interleavings that could occur at runtime.

Regarding the latter option, it is not trivial to show that the introduction of such a primitive would preserve the program's behavior, as the threads perform network I/O in parallel.

Instead, we modify the Tamarin model to reflect this parallelism directly in the model. We introduced two protocol roles `TokenVerifier` and `Verifier`, which model the asynchronous token signature verification and security level validation, respectively. The `Verifier` role sends a token `t` to the `TokenVerifier` role by producing a `PermitTokenVerification(t)` fact. We associate multiple rules with the `TokenVerifier` role, one for each valid token pattern, explained in more detail in Sec. 4.5. A rule consumes a `PermitTokenVerification(t)` and produces a `ValidTokenOut(t)` fact. The validity of the token's signature, and its public key commitment in the case of a root endorsement, are enforced via action labels and corresponding restrictions.

The communication between the two roles is modeled as a secure channel providing authenticity and confidentiality. That is, we assume that the memory in which the byte slice passed as the first argument to `vfyToken` and the values sent to the `results` channel are stored provides confidentiality and authenticity.

Unbounded loops

While the previously described model accepts one valid emblem, one valid organizational root endorsement, and an arbitrary number of valid authority endorsements as inputs, the implementation's `VerifyTokens` function is called with an arbitrary number of byte slices, each representing an input token, as described in Sec. 2.2. Passing an input byte slice `t` as an argument to the `vfyToken` function requires a corresponding I/O permission `PermitTokenVerification(t)`, as this corresponds to communication between the two protocol roles `TokenVerifier` and `Verifier`. Such a permission can only be obtained by first applying a corresponding rewriting rule. That is, to ensure the implementation adheres to the model, we included the reception of an arbitrary number of byte slices in the MRS described by `Verifier` (Sec. 4.1) as an unbounded, non-deterministic loop.

Similarly, the aforementioned abstraction of internal endorsements needed to be dropped to relate the model to the implementation. Consider the rule in Fig. 4.1, which produces an output fact for each of the security levels `SIGNED` and `ORGANIZATIONAL`. Note that the `ValidTokenIn(root)` and `ValidTokenIn(emblem)` facts are symbolically related by `intKey` and `oi`, representing the fact that `root` is an organizational endorsement of the emblem. We cannot relate the term-level representations of the root and emblem tokens in the implementation's specification, as the implementation must account for an internal endorsement chain of arbitrary length. That is, the root and emblem tokens' term-level representations are related by a chain of `ValidTokenIn` facts of arbitrary length. To address this issue, we included this internal endorsement chain in the model as an unbounded, non-deterministic loop.

```

1 rule OrganizationalEmblem:
2   let emblem = <intKey,      ai, oi>
3     root    = <orgKey, intKey, oi> in
4     [ ValidTokenIn(root), ValidTokenIn(emblem) ]
5     -->
6     [ Out(<'SIGNED', ai>), Out(<'ORGANIZATIONAL', ai, oi>) ]

```

Figure 4.1: The rule `OrganizationalEmblem` consumes two `ValidTokenIn` facts: one for an emblem, signed with an internal organizational key `intKey`, and one for a corresponding organizational root endorsement of the `intKey`. We display a simplified version of the rule here. For example, we omit any action labels or state facts.

4.3 I/O permissions

Typically, I/O operations in Tamigloo are understood to be *network* operations. However, in our case, the implementation does not perform network requests beyond those needed to verify root key commitments. It receives its input directly as its command line argument and outputs the verification results as its return value. To enable the application of Tamigloo to our case, we assume that parts

of the program's command line input, that is, the list of byte-encoded tokens, are provided by the adversary and thus correspond to inputs from the untrusted network. We reason that this is a valid assumption, as a user of the verifier implementation typically obtains its set of input tokens via the network.

Together with the communication between the `TokenVerifier` and `Verifier` roles, we must therefore provide specifications for command line send and receive operations, and for the I/O operations between the `VerifyTokens` and the `vfyToken` function.

Recall that `vfyToken` goroutines are started in a loop, iterating over the input byte slices. Consider the rewriting rule in Fig. 4.2. To encode that a byte slice supplied as an argument to `vfyToken` must first be received from the command line, we require a corresponding `PermitTokenVerification` permission. We apply the rule `ReceiveToken` in the loop body, thereby obtaining the required output facts. Note that without a `PermitTokenVerification` fact, none of the rewriting rules of the `TokenVerifier` are applicable.

```

1 rule ReceiveToken:
2   [ St_Verifier_1(rid), In(t) ]
3   -->
4   [ St_Verifier_1(rid), PermitTokenVerification(t) ]

```

Figure 4.2: The `ReceiveToken` rule describes the reception of a byte-encoded token from the environment. The output fact `PermitTokenVerificationOut(t)` is consumed as an input fact by the `TokenVerifier` rule. The rule produces an identical state fact, allowing for the reception of an arbitrary number of tokens.

To simulate the writing of a result to the command line, we provide trusted specifications of output functions for security levels `SIGNED`, `ORGANIZATIONAL`, and `ENDORSED`. Consider the specification in Fig. 4.3. The function requires an I/O permission and verifies that the arguments `protected` and `iss` correspond to permitted outputs `ai` and `oi`. If the call is successful, it moves the token from `p` to `p0`. All three provided specifications follow the same pattern.

The `VerifyTokens` function returns the achieved security levels as a slice and therefore does not impose an ordering on any executed I/O operations. As an I/O permission as required by the precondition of `OrgOut` is already bound to a place `p`, `VerifyTokens` cannot produce them directly. Instead, it returns the I/O specification `P_Verifier(p, rid, S)` such that `S` contains an output *fact* for each achieved security level. The I/O specification provides nested predicates to exchange the output fact for a corresponding I/O permission, bound to a place `p`.

Finally, sending or receiving a value to or from the `results` channel similarly requires a corresponding I/O permission. To send a value, an `e_ValidTokenOut` permission must be held. We encode this requirement by wrapping the channel send operation in a trusted function, consuming the permission in its precondition

```

1 requires token(p) &&
2   e_OutFact(p, rid, (ORGANIZATIONAL, ai, oi)) &&
3   gamma(ai) == AbsAI(protected) && gamma(oi) == stringB(iss)
4 ensures err == nil ==> token(p0) &&
5   p0 == get_e_OutFact_placeDst(p, rid, (ORGANIZATIONAL, ai, oi))
6 func OrgOut(
7   protected []*AI, iss string, p Place, rid, ai, oi Term
8 ) (err error, p0 Place)

```

Figure 4.3: Trusted specification of output function `OrgOut`. `AbsAI` and `stringB` abstract a slice of AI structs and a string to an element of \mathcal{B} , respectively. The function consumes an I/O permission and moves the token from `p` to `p0`.

(Fig. 4.4). We specify the channel receive operation in a complementary fashion, consuming a `ValidTokenIn` permission in its precondition.

```

1 requires result.err == nil ==> e_ValidTokenOut(rid, p, t)
2 ...
3 func resultsSend(
4   results chan *TokenVerificationResult,
5   result *TokenVerificationResult,
6   ...,
7   ghost rid Term, p Place, t Term
8 ) {
9   results <- result
10 }

```

Figure 4.4: Specification of the channel send operation. Sending a (successful) verification result from a `vfyToken` thread, implementing the `TokenVerifier` role, to the `VerifyTokens` thread, implementing the `Verifier` role, requires an `e_ValidTokenOut` permission.

4.4 Internal permissions

In this section, we present our work on the acquisition of permission for performing *internal* operations in the ADEM implementation.

4.4.1 Arbitrary numbers of input facts

The function `VerifyTokens` receives `n` tokens, initially encoded as byte slices, as its input. As described in Sec. 2.2, these `n` tokens are handled in multiple loops. Importantly, after verifying the token signatures and validating the emblem resp. endorsement constraints, the emblem's security levels are determined across two separate loops: the former to find the organizational root endorsement if it exists, and the latter to find any authority endorsements.

If an organizational root endorsement with a valid root key commitment is found, the implementation outputs security levels `SIGNED` and `ORGANIZATIONAL`. From Sec. 4.3 we know that corresponding I/O permissions must be obtained. They can be obtained from the `IsOrganizationalEmblem`, `CollectInternalEndorsements` and `CollectAuthorityEndorsements` transitions in the I/O specification. Similarly,

in order to output security level `ENDORSED`, corresponding I/O permissions must be obtained by applying the `IsEndorsedEmblem` rule. Each of these rules requires a `ValidTokenIn(t)` fact in order to be applicable, where `t` symbolically represents an ADEM token, subject to specific pattern constraints.

The abstract term representation of each token along with, as discussed in Sec. 4.3, a corresponding `ValidTokenIn` fact is produced by the receive operation on the `results` channel (Fig. 4.5).

```

1  unfold P_Verifier(p, rid, S)
2  unfold phiRF_Verifier_ValidTokenIn(p, rid, S)
3  assert e_ValidTokenIn(p, rid)
4  res, p, t := resultsRcv(results, p, rid)
5  S = S union mset[Fact] { ValidTokenIn_Verifier(rid, t) }
6  if result.err == nil {
7      assert Abs(res.token) == gamma(t)
8      tokens = append(tokens, result.token)
9      terms = terms ++ seq[Term] { t }
10     ...
11 }

```

Figure 4.5: A simplified version of the receive operation on the `results` channel in `Verify-Tokens`. The operation is preceded by `unfold` statements to obtain the input fact required to perform the receive operation. An abstract term is received in addition to a result, which is specified to be an abstraction of the enclosed ADEM token if the signature verification was successful. As a result of the `resultsRcv` operation, the `ValidTokenIn_Verifier` fact is added to the state multiset.

The terms are collected in a sequence `terms` such that `terms[i]` is the term-level representation of the `i`-th received token. To apply the aforementioned rewriting rules, we need to maintain an invariant on `s` such that a required I/O permission `ValidTokenIn` is available. At this point, we must note two things. First, recall from Sec. 4.1.3 that the elements of `terms` are not necessarily unique: the mapping from concrete tokens to abstract terms is not injective. [4] Therefore, an invariant such as `ValidTokenIn_Verifier(rid, t)` in `S` will not suffice: we need to constrain the facts' multiplicities in `s`. Second, a token is never removed from `tokens`, even after the corresponding input fact has been consumed. Therefore, an invariant on `s` must also specify which input facts have been used and, more importantly, which ones are available. Based on these observations, we write the invariant as shown in Fig. 4.6. By constraining the elements in `used`, the invariant can thus be used to reason about the presence of particular I/O permissions.

Unfortunately, applying this invariant to the codebase proved to be challenging: due to limitations in the current axiomatization of sequences and multisets in `Gobra`, in particular, in reasoning about their multiplicities, the verifier fails to deduce that the invariant is maintained if an input fact is consumed and the variable `used` is updated accordingly. (Fig. 4.7).

We were unable to circumvent this issue within a reasonable timeframe. Therefore, we assume `ValidTokenIn_Verifier(rid, t)` in `s` wherever needed. That is, we

```

1 pure func InFacts(
2     rid Term,
3     terms, used seq[Term],
4     S mset[Fact]
5 ) bool {
6     return forall i int :: 0 <= i && i < len(terms) ==> (
7         terms[i] # terms - terms[i] # used
8             <= ValidTokenIn_Verifier(rid, terms[i]) # S
9     )
10 }

```

Figure 4.6: The invariant on S , propagated throughout the program to specify the multiplicity of available I/O permissions.

```

1 requires InFacts(rid, tokens, used, S)
2 requires t in tokens && !(t in used)
3 func InFactsExample(rid, t Term, S mset[Fact], tokens, used seq[Term]) {
4     assert ValidTokenIn_Verifier(rid, t) in S
5     S = S setminus mset[Fact] {ValidTokenIn_Verifier(rid, t)}
6     used = used ++ seq[Term] { t }
7
8     // fails
9     assert InFacts(rid, tokens, used, S)

```

Figure 4.7: A minimal example presenting the issue with trivial usage of the `InFacts` invariant. Due to limitations in the Gobra program verifier, the final assertion cannot be proved.

assume that a rule never consumes the same token t , more specifically, the same `ValidTokenIn_Verifier(rid, t)` fact, twice. To see the soundness of this assumption, we make three observations.

First, each token received from the `results` channel uniquely corresponds to a byte-encoded token in the input. Therefore, each received token corresponds to a distinct `ADEMTOKEN` struct. Second, in any loop in which rewriting rules are applied, the list of tokens is simply iterated for increasing indices. Therefore, no token is consumed twice within a single loop. Third, in order for `IsOrganizationalEmblem`, `CollectInternalEndorsements` or `CollectAuthorityEndorsements` to be applicable, a token's issuer, and subject if applicable, must be equal to the emblem's issuer. In order for `IsEndorsedEmblem` to be applicable, however, the token's issuer must be distinct from the emblem's issuer. Therefore, a token can't be consumed twice across multiple loops, either.

In summary, a token is consumed at most once. Since, a token is only ever received from the `results` channel along with a corresponding input fact, that is, for any consumed token a corresponding input fact must be present.

4.4.2 Organizational endorsement chain

As described in Sec. 4.2.1, to obtain the I/O permissions for the security levels `SIGNED` and `ORGANIZATIONAL`, we need to model the chain of internal endorsements, leading from a root endorsement with a valid public key commitment to the emblem.

Fig. 4.8 shows the rewriting rule describing the iteration over internal endorsements in a non-deterministic loop. Note that the state fact `St_Verifier_3` is continuously updated with a new endorsement key. The execution enters this loop by receiving a valid root endorsement, issued by the same organization as the emblem, and traverses the chain of internal endorsements until the key signing the emblem is encountered. A transition out of the loop produces I/O permissions for the security levels `SIGNED` and `ORGANIZATIONAL`.

```

1 rule CollectInternalEndorsements:
2   let endorsement = <key, oi, <'end', oi, newKey>, sig>
3   in
4   [ St_Verifier_3(rid, oi, rootKey, key)
5     , ValidTokenIn(endorsement) ]
6   -->
7   [ St_Verifier_3(rid, oi, rootKey, newKey) ]

```

Figure 4.8: The rule `CollectInternalEndorsements` describes the non-deterministic loop modeling an arbitrary-length chain of internal endorsements. The term `rootKey` is required to receive authority endorsements in later states of the `Verifier` role.

To apply this rewriting rule for an endorsement at the implementation level, two things are required: the term-level representation τ of the endorsement along with a corresponding `ValidTokenIn` fact, and that the signing key `key` in τ matches the corresponding term in the state fact. The latter implies that we require that the endorsement is endorsed by the preceding endorsement, while the former requires us to explicitly keep track of the endorsement in order to be able to retrieve τ from `terms`, the list of term-level representations of received ADEM tokens. Therefore, we must explicitly model the entire chain of internal endorsements, and loop over its members starting from the root endorsement.

We model the chain of internal endorsements as a sequence of values of type `*ADEMToken`. We maintain the invariant that every element at index i of the sequence endorses the element at index $i + 1$, and require that the first element of the chain to be the organizational root token and the final element is the emblem.

Unfortunately, maintaining the invariant proved challenging. We were unable to formulate a specification such that the program verifier terminated within a reasonable timeframe.

Therefore, we limited our analysis to the case of an internal endorsement chain of length 1. That is, the emblem's verification key is directly endorsed by the organizational root endorsement. This assumption corresponds to marking the portion of code responsible for identifying the root endorsement as trusted. A manual inspection of the code provides us with confidence that this trust assumption holds up.

4.4.3 Library interfaces

As described in Sec. 4.2.1, the `TokenVerifier` role is implemented by the `vfyToken` function. The function parses a byte slice as a JWS structure and verifies its signature if one is included. If the signature verification was successful, the parsed token is stored in an `ADEMTOKEN` struct, which is then sent as a pointer to the `results` channel.

To obtain the necessary I/O permission for sending on this channel, one of the rewriting rules of the `TokenVerifier` role in the Tamarin model must be applied. Which rule to apply depends on the term-level representation of the underlying ADEM token, which is resolved by parsing the token. Independent of the particular rule in question, a `PermitTokenVerificationIn` permission is consumed, and a `ValidTokenOut` permission is produced. Note that the latter is the permission required to send a value to `results`. In addition, most rules require that the token's signature is valid through corresponding action labels.

Normally, the constraints required to apply a particular rule would be collected through the successive application of library functions. For example, one would call a function to parse the aforementioned byte slice as a JWS structure, or another function to verify a signature. However, most of this logic is implemented directly in the `jwt.Parse` function provided by an external library. [13] Therefore, we encode the state transitions directly in the pre- and postconditions of our manually written specification of the library function. Note that this means we assume the function works as documented.

4.5 Pattern requirements

As introduced in Sec. 4.1.3, we employ the surjective homomorphism $\gamma : \mathcal{M} \rightarrow \mathcal{B}$ to relate terms to concrete bytestrings. It is encoded as an abstract function `gamma` with the required totality and homomorphism constraints included as axioms. To abstract in-memory objects, such as the `ADEMTOKEN` structs on which our program operates, to the bytestring algebra \mathcal{B} , we employ a function `Abs`.

As described in Sec. 4.4.2, we maintain a slice of concrete ADEM tokens `tokens` along with a corresponding sequence of abstract terms `terms`, such that `Abs(tokens[i]) == gamma(term[i])` for all valid indices `i`.

The ADEM protocol model introduces six non-linear patterns for *valid* tokens, induced by corresponding rewriting rules:

- (1) *unsigned emblems*, which are valid emblems that are unsigned,
- (2) *anonymous emblems*, which are signed emblems that do not specify an organization identifier,
- (3) *signed emblems*, which are signed emblems that *do* specify an organization identifier,

- (4) *organizational endorsements*, which are endorsements that are issued by the same organization as the key they endorse and do *not* specify a public key commitment,
- (5) *root endorsements*, which are endorsements that are issued by the same organization as the key they endorse, and *do* specify a public key commitment, and
- (6) *authority endorsements*, which are endorsements that endorse an organization that is distinct from their issuer, and *do* specify a public key commitment.

We define `Abs` such that it maps a token to a corresponding pattern if one of the above cases applies, and is left unspecified otherwise. Note that the listed cases are mutually disjoint.

Due to prohibitive increases in verification time, we do not implement `Abs` as a single pure function. Instead, we define multiple abstract functions, one for each listed pattern. The functions' preconditions check that a token `token` is constrained correctly and provide corresponding patterns for `Abs(token)` in their postconditions. Consider the function `RootPattern` in Fig. 4.9 for an example.

```

1  ghost
2  requires t.Headers.ContentType() == string(EndorsementCty) &&
3         t.Token.Contains("key") &&
4         t.Token.Issuer() != "" &&
5         t.Token.Contains("log") &&
6         t.Token.Subject() == t.Token.Issuer()
7  ensures Abs(t) == tuple4B(
8         stringB(t.VerificationKey.KeyID()),
9         stringB(t.Token.Issuer()),
10        tuple3B(
11            rootEndB(),
12            stringB(t.Token.Subject()),
13            stringB(t.Token.PureKeyID())
14        ),
15        stringB("sig")
16    )
17  pure func RootPattern(t *ADEMToken)

```

Figure 4.9: The function `RootPattern` provides a concrete bytestring representation of an organizational root endorsement `t`, assuming it matches the listed constraints. The functions `tuple4B` and `tuple3B` return bytestring representations of tuples of length 4 and 3, respectively. The function `stringB` returns the bytestring representation of the given string. Note while the token's signature is part of its term pattern, it is not stored in the `ADEMToken` structs, and therefore we do not have access to their values, here. However, as the signature can be immediately deduced from a verification key and the token's body, we can safely omit it.

At this point, an application of a pattern requirement is necessary to constrain a token's term-level representation to match the correct term in the I/O specification. To that end, we implemented the corresponding pattern requirements. Consider Fig. 4.10 for an example.

Recall that ADEM tokens are represented as JavaScript object notation (JSON) structures [16], encoded using UTF-8 [17]. Therefore, a token can unambiguously

```

1  ghost
2  requires token(p) && P_Verifier(p, rid, S) && St_Verifier_2(rid) in S
3  requires gamma(t) == gamma(tuple4(
4      someRootKey,
5      someOi,
6      tuple3(
7          pubTerm(const_root_end_pub()),
8          someOi,
9          someEndorsedKey
10     ),
11     someSig))
12 ensures token(p) && P_Verifier(p, rid, S) && St_Verifier_2(rid) in S
13 ensures t == tuple4(
14     rootkey,
15     oi,
16     tuple3(pubTerm(const_root_end_pub()), oi, endorsedkey),
17     sig)
18 func RootPaR(
19     t, rid, someRootKey, someOi, someEndorsedKey, someSig Term,
20     s mset[Fact],
21     p Place
22 ) (rootkey, oi, endorsedkey, sig Term)

```

Figure 4.10: The pattern requirement for the organizational root endorsement pattern encodes the implication that if a term t and ground instance of the organizational root endorsement pattern coincide under γ (l.3), then t must also match some ground instance of the pattern (l.13).

be parsed. As the previously listed patterns are mutually exclusive, we conclude that the mapping from a serialized ADEM token to a term representation is unambiguous. In combination with assumption 2 for cryptographic operations from the Tamigloo paper, we argue that ADEM satisfies the pattern requirement.

Chapter 5

Discussion

In this section, we evaluate the effectiveness and efficiency of our analysis. We begin by presenting verification time measurements and the overhead in terms of lines of code introduced by the formal specification. Then, we depict our experience using Gobra and Tamigloo, including a summary of found bugs, and highlight some of the issues discussed in chapters 3 and 4. Finally, we discuss the result relating to the security properties provided by the verified implementation.

5.1 Verification overhead

For each package, and for the entire codebase in total, table 5.1 lists the lines of code (LOC), lines of specification (LOS), and functional verification and memory verification times in seconds. The functional verification time was measured for the entire specification as described in Ch. 4, while the memory verification time was measured for the specification as described in Ch. 3, obtained by removing any assertions, pre-, and postconditions that produced proof obligations unrelated to memory safety verification. The ADEM code implementation consists of 1326 lines of verified Go code. The verification amounted to 4329 lines of specification, approximately 2000 of which are attributable to library stubs and the I/O specification generated by Tamigloo.

We note a significant increase in verification time for the `vfy` package. While this increase is roughly proportional to the LOC in the memory verification case, we see that the difference becomes much more pronounced in the functional verification case. As the majority of Tamigloo-related annotations are contained in the `vfy` package, this increase is likely attributable to the usage of complex data structures such as multisets and sequences. For example, we observed that the introduction of proof obligations concerning the multiplicities of particular facts in the fact multiset `s` used by the I/O specification led to a significant increase in verification time. In comparison, the case studies provided by the Tamigloo paper [4] constrained `s` by specifying its equality to a multiset literal.

Package[s]	LOC	LOS	Func. verification [s]	Mem. verification [s]
const	42	15	9.8	12.3
util	69	57	15.4	17.6
ident	158	59	35.5	36.3
roots	229	90	39.8	38.5
tokens	272	265	27.8	29.2
vyf	545	1503	226.2	89.7
total	1326	4349	358.7	171.7

Table 5.1: LOC and LOS (incl. ghost code), and the average functional and memory verification times, separated by package. The times were measured by computing the arithmetic mean over 10 runs on a Lenovo X1 Carbon with an Intel i7-1270P. Note that the total LOS include trusted library function specifications and the I/O specification files generated by Tamigloo.

5.2 Found bugs

In this section, we highlight several implementation bugs that were identified as a result of attempting to find a suitable specification for the code. Memory-related bugs were often identified as a result of Gobra failing to verify a particular section of code, while functionality-related bugs were identified as a result of attempting to come up with suitable invariants: back-propagating a necessary invariant through the implementation’s specification made it clear that it couldn’t be established in the first place.

Improper input sanitization

The implementation suffered from two bugs relating to improper input sanitization.

First, a missing nil-check for the input set of trusted keys could lead to a runtime error due to a nil-pointer access. Fixing the bug was trivial: if the set is `nil`, initialize it to be the empty set. The bug was identified directly by Gobra.

Second, the implementation suffered from a more subtle bug related to improper input sanitization. Namely, if the input list of byte slices was empty, this could lead to non-termination. More specifically, the `VerifyTokens` function will block indefinitely in an attempt to receive a signature verification from the `results` channel.

Note that Gobra does support verifying that channel send or receive operations terminate. Therefore, this bug needed to be identified manually. Curiously, this did occur as a by-product of formal verification using Gobra, in that coming up with precise loop invariants made the bug a lot easier to see.

Incomplete public key commitment verification

In formalizing the required constraints for tokens with a valid public key commitment (Sec. 4.5), it became clear that the validation process was incomplete.

Consider the code in Fig. 5.1. It validates the commitment of a public key to the CT logs. If the validation is successful, the public key can be used to validate a token's signature. Two issues arise from this implementation.

First, note that if `log` maps to an empty array the validation is successful. Thus, a party may provide a root verification key without ever publicly committing to it. Second, no knowledge regarding the validation results is propagated other than the presence of a signature verification key. Therefore, when determining the achieved security levels of an emblem, the implementation can only check for the presence of a `log` field in an endorsement to determine whether or not it provides a root public key. This leads to multiple issues. For example, an endorsement with `"log": []` may lift an emblem's security level to `ORGANIZATIONAL`, where it would otherwise only be `SIGNED` or even `INVALID`.

We circumvent this issue with a simple solution: if `err` is non-nil, the `FetchKeys` method returns an error. As a consequence, the signature verification will fail and the token will be dropped.

```
1  if logs, ok := t.Get("log"); ok {
2      headerKey := sig.ProtectedHeaders().JWK()
3      for _, r := range VerifyBindingCerts(
4          t.Issuer(), headerKey, logs.([]*LogConfig)
5      ) {
6          if !r.Ok {
7              err = ErrRootKeyUnbound
8              break
9          }
10     }
11
12     if err == nil {
13         km.put(headerKey)
14     }
15 }
```

Figure 5.1: An incomplete implementation of the public key commitment validation process. The "log" key provides an array of JSON objects, each corresponding to a CT log commitment. The entries are validated by the `VerifyBindingCerts` function. If the array does not contain any invalid entries, the verification key can be used to verify the signatures of ADEM tokens where possible.

Incorrect security level verification

Another interesting issue arose during the verification of the security level verification procedures in `verifySignedOrganizational` and `verifyEndorsed`.

Namely, the case of an undefined `iss` claim was handled improperly. Specifically, if an emblem's `iss` claim, that is, its organization identifier (OI), was undefined,

it could achieve security levels `SIGNED` and `ENDORSED`, but not `ORGANIZATIONAL`. This situation could occur if the top-level endorsement with `iss` undefined was endorsed by a valid authority endorsement. We amended the situation simply by explicitly checking that security level `ORGANIZATIONAL` was achieved before calling `verifyEndorsed`.

This issue was identified during the application of the Tamigloo [4] methodology. It became obvious that the state multiset would not necessarily contain the necessary state fact to apply the rewriting rules required to obtain I/O permission to output `ENDORSED` security level.

5.3 ADEM security proof

In Sec. 4.2.1 we described the design decisions that were made to ensure the implementation adheres to the model and the model fulfills Tamigloo's syntax constraints. In addition to the definition of two protocol roles `Verifier` and `Token-Verifier`, we introduced two additional non-deterministic loops, one to model the arbitrary number of input tokens, and one to model the arbitrary-length chain of internal endorsements.

In Tamarin, when reasoning about non-deterministic loops such as the ones described above, we often make use of facts with injective instances: if a MRS \mathcal{R} has no reachable state with more than one instance of a fact symbol \mathcal{F} with the same term as a first argument, \mathcal{F} is said to have injective instances. As the problem of computing the set of all such \mathcal{F} in \mathcal{R} is undecidable in general, Tamarin computes an under-approximation to this set. To that end, it employs a simple heuristic. The heuristic requires that for each occurrence of \mathcal{F} in a rule, there is no other occurrence with the same first term. Additionally, it requires that either there is `Fr` fact of the first term as a premise, or there is exactly one fact tag with the same first term in a premise. [18]

Here, a clash occurs with the syntax constraints imposed by Tamigloo. Recall from Sec. 4.1.2 that for all $l \xrightarrow{a} r \in R_i$, the first term *rid* of all state facts must be the same and of type *fresh*, representing the thread identifier of a run of a protocol role. Usually, a protocol transitions through multiple states. That is, there are *multiple* state facts such that the first term is the same. Therefore, the described heuristic may fail to identify injective instances of state facts. Note that this is a result of the under-approximation: for any occurrence of a state fact on the right-hand side of a rule in R_i , either exactly one state fact is consumed, or $Fr(rid)$ occurs in the premise. Therefore, injective instances of the state facts *should* exist.

As a result, we cannot make use of facts with injective instances to reason about the three aforementioned, non-deterministic loops. Proving the desired security properties without the use of injective instances of facts is non-trivial and out-of-scope for this thesis.

5.4 Summary

The introduction of Tamigloo enabled us to show the security of the implementation without needing to explicitly model the security properties described in Sec. 2.1. This advantage is considerable, as finding sufficiently strong invariants to prove security properties at the implementation level can be challenging. [14]

However, Tamarin models often employ abstractions to support the automation of the proofs of security properties. Such abstractions create a gap between a model and its implementation. As demonstrated in Sec. 4.2, in the presence of such abstractions, relating an implementation to its model requires modifications, either to the model or to the implementation.

Furthermore, the syntax constraints imposed by Tamigloo conflict with the current heuristic employed by Tamarin to identify injective instances of facts. Proving properties for non-deterministic loops without being able to leverage the injectivity of fact instances can be non-trivial. Note that this shortcoming is attributable not *just* to the syntax constraints imposed by Tamigloo, but also to the limited support of reasoning about fact instance injectivity in Tamarin. For example, if the user could specify fact symbols with injective instances and explicitly construct a soundness proof, proving properties about non-deterministic loops would again become possible.

Finally, we note that the limited support for reasoning about element multiplicity in multisets and sequences in Gobra led to a significant increase in verification time, despite being subject to a considerable number of assumptions. Again, this shortcoming is attributable not to either Gobra or Tamigloo, but to the interplay between the two.

Chapter 6

Conclusion

In this thesis, we formally verified the memory safety of the verification component of the ADEM codebase. We showed memory safety (e.g. that there are no null-pointer dereferences), crash safety (e.g. no division by zero), and data race freedom of the implementation.

Next, we used the methodology presented in [4] to show that the implementation is a refinement of a preexisting Tamarin model. We modified the security model where necessary and, consequently, kept changes to the implementation minimal.

We qualitatively evaluated our work and compared the verification time overhead of memory and functional verification, respectively, observing that the functional verification with Tamigloo significantly increased the verification time overhead. We identified multiple bugs in the implementation and amended them accordingly.

Finally, we note that proving the desired security properties for the modified model is out-of-scope for this thesis. Therefore, we cannot conclude that the desired security properties, described in Sec. 2.1 and shown to hold for the preexisting model, hold for the implementation.

6.1 Future work

Showing the security properties

As previously mentioned, we cannot conclude that the desired security properties are provided by the implementation. In future work, it would be desirable to build on this work and show that the implementation does so. There are multiple possible approaches.

First, the Tamarin protocol prover could be extended to support the explicit specification of instances of injective facts. With this tool in hand, it could be possible to prove that the security properties hold.

Alternatively, one could show that the properties hold directly at the implementation level. For example, one might employ the approach outlined by Arquint et al [14] to prove security properties through the definition and verification of trace-based invariants.

Termination

It would be interesting to show that the implementation terminates in all cases. Recall the bug relating to improper input sanitization introduced in Sec. 5.2: due to missing input validation, the `VerifyTokens` function could stall indefinitely. To show termination, Gobra would need to be extended to allow reasoning about the termination of channel send or receive operations, as these operations may block depending on the number of active goroutines and the channel's buffer size.

Tamigloo syntax relaxation

Finally, it might be interesting to research the possibility of relaxing the syntax constraints imposed by Tamigloo. For example, by not requiring that a protocol role's initial parameters appear as the first k_i arguments of every state fact, it may be possible to utilize the heuristic described above to automatically identify injective instances of facts.

Appendix A

Code excerpts

A.1 Generated I/O specifications

The following presents an excerpt from the generated I/O specification for the Verifier role. The predicate `P_Verifier` encodes the separating conjunction $P_i(p, rid, S)$, as presented in Sec. 4.1.

The predicates `phiR_Verifier_*` encode role-specific rewriting rules, that is, those that do not produce any I/O permissions. Their bodies represent the intuition that if a state `s` contains all the facts on the left-hand side `l` of the rewriting rule, it is updated by removing any linear facts contained in `l` and producing any facts in `r`. Functions `M` and `U` implement the respective multiset operations. In addition, the right-hand side of the implication grants emphinternal permissions to perform any corresponding operations.

The predicates `phiRG_Verifier_*` and `phiRF_Verifier_*` grant I/O permissions required to execute any I/O operations. Note, in particular, that the listed predicate produces the necessary out facts for the `VerifyTokens` function.

```
1  pred P_Verifier(p Place, rid Term, ghost S mset[Fact]) {
2    phiR_Verifier_0(p, rid, S) && ...
3    phiRG_Verifier_14(p, rid, S) && ...
4    phiRF_Verifier_16(p, rid, S) && ...
5  }

6  pred phiR_Verifier_1(
7    p Place, rid Term, ghost s mset[Fact]
8  ) {
9    forall rid Term, t Term, lp mset[Fact], ap mset[Claim], rp mset[Fact] ::
10     { e_ReceiveToken(p, rid, t, lp, ap, rp) } (
11       (M(lp, s) &&
12         lp == mset[Fact] {
13           St_Verifier_1(rid),
14           InFact_Verifier(rid, t)} &&
15         ap == mset[Claim] { } &&
16         rp == mset[Fact] {
17           St_Verifier_1(rid),
18           PermitTokenVerificationOut_Verifier(rid, t)})
19     ==>
20     (e_ReceiveToken(p, rid, t, lp, ap, rp) &&
21       P_Verifier(
```

```

22             get_e_ReceiveToken_placeDst(p, rid, t, lp, ap, rp),
23             rid,
24             U(lp, rp, s)
25         )
26     )
27 )
28 }

29 pred phiRG_Verifier_15(p Place, rid Term, ghost s mset[Fact]) {
30     forall x Term ::
31         { e_OutFact(p, rid, x) }{ OutFact_Verifier(rid, x) } (
32             ((OutFact_Verifier(rid, x) # s) > 0) ==>
33             (e_OutFact(p, rid, x) &&
34             P_Verifier(
35                 get_e_OutFact_placeDst(p, rid, x),
36                 rid,
37                 s setminus mset[Fact] {
38                     OutFact_Verifier(rid, x)}))
39         )
40 }

41 ...

```

Bibliography

- [1] F. Linker and D. Basin, “ADEM: An authentic digital emblem,” in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, to appear.
- [2] S. Meier, B. Schmidt, C. Cremers, and D. Basin, “The tamarin prover for the symbolic analysis of security protocols,” in *Computer Aided Verification*, N. Sharygina and H. Veith, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 696–701.
- [3] “adem-wg/adem-proto: This repository contains libraries and command line utility support for an authentic digital emblem (adem),” <https://adem-wg.github.io/adem-spec/>, (accessed Sept. 27, 2023).
- [4] L. Arquint, F. A. Wolf, J. Lallemand, R. Sasse, C. Sprenger, S. N. Wiesner, D. Basin, and P. Müller, “Sound verification of security protocols: From design to interoperable implementations,” in *2023 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2023, pp. 1077–1093. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00061>
- [5] ICRC, *Digitalizing the Red Cross, Red Crescent and Red Crystal Emblems: Benefits, Risks, and Possible Solutions*. Geneva: ICRC, 2022.
- [6] M. B. Jones, J. Bradley, and N. Sakimura, “JSON Web Token (JWT),” RFC 7519, May 2015. [Online]. Available: <https://www.rfc-editor.org/info/rfc7519>
- [7] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller, “Gobra: Modular specification and verification of go programs,” in *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds. Cham: Springer International Publishing, 2021, pp. 367–379.

- [8] P. O’Hearn, J. Reynolds, and H. Yang, “Local reasoning about programs that alter data structures,” in *Computer Science Logic*, L. Fribourg, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–19.
- [9] P. Müller, M. Schwerhoff, and A. J. Summers, “Viper: A verification infrastructure for permission-based reasoning,” in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, B. Jobstmann and K. R. M. Leino, Eds., vol. 9583. Springer-Verlag, 2016, pp. 41–62.
- [10] “Z3prover/z3: The z3 theorem prover,” <https://github.com/Z3Prover/z3>, (accessed Sept. 28, 2023).
- [11] “CVE-2014-0160 - the (1) tls and (2) dtls implementations in openssl 1.0.1 before 1.0.1g do not properly handle heart - CVE-Search,” <https://cve.circl.lu/cve/CVE-2014-0160>, accessed: Mar. 14, 2024.
- [12] “CWE-2023 - cwe top 25 most dangerous software weaknesses,” https://cwe.mitre.org/top25/archive/2023/2023_kev_list.html, accessed: March 14, 2024.
- [13] “lestrat-go/jwx: Implementation of various jwx (javascript object signing and encryption/jose) technologies,” <https://github.com/lestrat-go/jwx>, (accessed March 27, 2024).
- [14] L. Arquint, M. Schwerhoff, V. Mehta, and P. Müller, “A generic methodology for the modular verification of security protocol implementations,” in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1377–1391. [Online]. Available: <https://doi.org/10.1145/3576915.3623105>
- [15] W. Penninckx, B. Jacobs, and F. Piessens, “Sound, modular and compositional verification of the input/output behavior of programs,” in *Programming Languages and Systems*, J. Vitek, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 158–182.
- [16] T. Bray, “The JavaScript Object Notation (JSON) Data Interchange Format,” RFC 8259, Dec. 2017. [Online]. Available: <https://www.rfc-editor.org/info/rfc8259>
- [17] F. Yergeau, “UTF-8, a transformation format of ISO 10646,” RFC 3629, Nov. 2003. [Online]. Available: <https://www.rfc-editor.org/info/rfc3629>
- [18] “Tamarin prover manual, by the tamarin team. copyright © 2016.” <https://tamarin-prover.com/manual/index.html>, (accessed April 8, 2024).



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. In consultation with the supervisor, one of the following three options must be selected:

- I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies¹.
- I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used and cited generative artificial intelligence technologies².
- I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used generative artificial intelligence technologies³. In consultation with the supervisor, I did not cite them.

Title of paper or thesis:

Verification² of the Authentic Digital Emblem

Authored by:

If the work was compiled in a group, the names of all authors are required.

Last name(s):

Meinen

First name(s):

Lasse

With my signature I confirm the following:


- I have adhered to the rules set out in the Citation Guide.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date

Zürich, 11.04.2024

Signature(s)



If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

¹ E.g. ChatGPT, DALL E 2, Google Bard
² E.g. ChatGPT, DALL E 2, Google Bard
³ E.g. ChatGPT, DALL E 2, Google Bard