ETH zürich

Secure Deletion of Sensitive Data in Protocol Implementations

Master's Thesis

Hugo Queinnec

September 24, 2023

Advisors: Prof. Dr. Peter Müller, Linard Arquint

Department of Computer Science, ETH Zürich

Abstract

Security protocols are crucial for protecting sensitive information and communications in today's digital age. Even a minor flaw in how these protocols are implemented can lead to serious consequences. Hence, proving the implementations secure is attractive as we prove the absence of such flaws.

Arquint et al. [1] propose a generic and modular methodology to verify the security of protocol implementations. We extend their methodology to reason about ephemeral and time-sensitive data, which must be deleted within certain time frames. This enhancement allows us to verify strong security properties, such as forward secrecy and post-compromise security, for protocols that frequently renew their keys, such as Signal. Our contributions encompass a conceptual expansion of their methodology and an extension of their Go library, which simplifies the verification of protocol implementations in Go. A case study, featuring a Signal-like protocol implementation, showcases expressiveness and practical applicability of our methodology extension.

Acknowledgements

I would like to sincerely thank Linard Arquint who supervised me throughout this thesis. He made himself available every week, whether in his office in Zurich or from the USA, to discuss for several hours the many questions I had. Without him, this project would not have progressed as far as it has. I would also like to thank him for his very attentive proofreading of this thesis, and for the hundreds of comments that helped improve it.

I would also like to thank Prof. Dr. Peter Müller and all the members of the Programming Methodology Group at ETH Zurich, for giving me the opportunity to complete this thesis with them, and for the questions and feedback they gave me during my oral presentations.

Finally, I would like to thank my family and friends for their support throughout my studies, especially during this thesis.

Contents

Contents **iii**

1 Introduction **1**
1.1 Outline . . . 3

2 Background **5**
2.1 The Gobra verifier . . . 5
2.1.1 Verifying a Go program . . . 5
2.1.2 Permissions . . . 6
2.2 Verification of protocol implementations . . . 7
2.2.1 Symbolic protocol analysis . . . 8
2.2.2 Security properties . . . 8
2.2.3 DY* . . . 9
2.2.4 Modular verification of existing implementations . . . 11
2.3 Cryptography . . . 12
2.3.1 AEAD encryption . . . 13
2.3.2 Diffie-Hellman key exchange . . . 13

3 Design **15**
3.1 Overview . . . 15
3.1.1 Goal . . . 16
3.1.2 Versions . . . 18
3.1.3 Enforcing deletion of old data . . . 19
3.2 Counting permissions . . . 21
3.2.1 Introduction . . . 21
3.2.2 Implementation in Gobra . . . 22
3.3 Extension of the library . . . 26
3.3.1 Storing the current session version . . . 26
3.3.2 Creation of versioned values . . . 27
3.3.3 Secure deletion of versioned values . . . 29

3.3.4 Creating values with multiple versions for ratcheting . 30
3.3.5 Encryption and decryption of versioned values 32
3.3.6 Increasing the version of a session 35
3.4 Alternative design using obligations 37

4 Case study **39**
4.1 Choosing a protocol to verify 39
4.1.1 Ratcheting protocols 39
4.1.2 Initial key agreement and chosen protocol 40
4.2 Implementation and verification 44
4.2.1 Ratcheting using a key derivation function 45
4.2.2 Usages and message invariants 47
4.2.3 First communication round 48
4.2.4 Using the deletion mechanism 49
4.2.5 Corruption 51
4.3 Remaining work 52
4.4 Results 54

5 Conclusion **57**
5.1 Future work 58

Bibliography **59**

Chapter 1

Introduction

Security protocols are omnipresent in our daily lives, they are the foundation for many applications ranging from online banking to text messaging. These protocols employ cryptography to achieve fundamental security properties such as authentication and confidentiality. Online banking operations serve as a prime example of how heavily we rely on security protocols. A breach in the security of these protocols could lead to substantial financial losses, which is why it is essential that these protocols and their implementations are both correct and secure. For example, proving a property such as *injective agreement* protects against replay attacks, meaning that the protocol participants reject messages from an attacker attempting to resend messages that participants may have sent in the past. In our online banking example, such a message could contain the request to perform a certain transaction, and it is thus crucial that this request cannot be replayed by an attacker.

Moreover, security protocols are not limited to the banking sector and are now crucial to a wider audience, especially for secure messaging. Secure messaging is a cornerstone of modern communication, with individuals, businesses, and governments relying on it to protect sensitive information. Robust security protocols are paramount in safeguarding the confidentiality and integrity of messages. For instance, a security breach could lead to the exposure of personal conversations, critical business data, or confidential government communications, potentially resulting in severe consequences. Therefore, several popular messaging applications, such as Signal, WhatsApp, and Facebook Messenger, rely on the Signal protocol [9] to offer secure and end-to-end encrypted communication to billions of users.

Guaranteeing security properties for protocols is a challenging task, especially in the presence of a strong attacker. In order to ensure these properties for all possible protocol executions, verification must consider an arbitrary number of participants and protocol sessions, as well as any possible ordering of protocol steps. Furthermore, protocol security properties are generally not

local to a particular participant but global, such as mutual authentication between two parties. This presents an additional challenge for verification as it requires a global view of the protocol execution. Moreover, it is also particularly difficult to reason about implementation-level specificities, such as mutable state and concurrency, while reasoning about security properties.

In this regard, two aspects of the protocol must be considered: the high-level protocol itself (that we will call *model*), and its implementation. Proving security properties for a protocol model is an active field of research and several promising automatic verifiers have been proposed (like Tamarin [10]). However, a verified protocol model does not imply that an implementation is free of security vulnerabilities and achieves the same security properties as its model. This is why we focus on verifying security properties for protocol implementations.

While Arquint and al. [1] propose a promising methodology for verifying security properties for protocol implementations, they do not reason about sensitive data that must be deleted in a timely manner. However, such reasoning is necessary for proving security properties like forward secrecy and post-compromise security, for protocols that periodically renew keys, like Signal. Forward secrecy informally guarantees that past keys remain secret even when newer keys or long-term keys become known to the attacker. It is crucial for secure communication, as it ensures that an attacker cannot decrypt past messages even if they manage to compromise a participant later on. Post-compromise security informally means that a participant can communicate securely with a peer, even when certain secrets from the past have leaked. This is sometimes referred to as *healing*, which allows communication to resume securely at some point after a compromise. Protocols like Signal achieve both of these properties by using a key rotation scheme within a protocol session, where a communication key is used to generate its successor and is then deleted.

The work of Arquint et al. comes with a Reusable Verification Library that implements protocol-independent components of the methodology to reduce the verification effort per protocol. Because their methodology is generic, the Reusable Verification Library can be implemented in a wide variety of programming languages. For the scope of this thesis, we will limit ourselves to protocol implementations written in Go, for which the Reusable Verification Library has been implemented.

In this thesis, we extend the methodology for the modular verification of security protocol implementations by Arquint et al. to make it capable of reasoning about sensitive and ephemeral data that must exist only for a limited time before being deleted. We present a generic extension of the methodology that allows the verification of any type of protocol implementation using ephemeral data, like Signal, in order to satisfy strong security properties

such as forward secrecy and post-compromise security. In addition to the methodology, we extend the Go Reusable Verification Library accordingly. Finally, we evaluate our methodology and our extension of the Go Reusable Verification Library by partly verifying a protocol implementation employing a key rotation scheme to achieve forward secrecy and post-compromise security.

1.1 Outline

The remainder of this thesis proceeds as follows.

In Chapter 2, we introduce the necessary background knowledge for this thesis, including an introduction to the verification of Go programs, a review of existing works aiming at verifying protocol implementations, and explanations of basic cryptography concepts.

In Chapter 3, we first present the conceptual idea behind our methodology to verify protocol implementations relying on ephemeral data. We then explain how we extended the Go Reusable Verification Library to support our methodology.

In Chapter 4, we present a case study of our methodology on a Signal-like protocol implementation, and explain how we can use the library to verify strong security properties for this implementation.

Finally, in Chapter 5, we conclude and discuss future work.

Chapter 2

Background

In this chapter, we introduce the reader to the concepts that are necessary to understand the rest of this thesis. We start by introducing how code verification works with a particular focus on the Gobra [16] verifier. Then, we present existing work on the verification of protocol implementations, which will form the foundation of our work. Finally, we explain two important cryptographic concepts that will be used in the protocols that we aim to verify.

2.1 The Gobra verifier

Gobra [16] is a verifier for programs written in the Go programming language. In practice, a user writes a specification of the program, and Gobra verifies that every execution of the program satisfies this specification. Gobra is based on the Viper [11] verification infrastructure. This means that the Gobra specification and Go implementation are translated to the Viper intermediate language, which is then verified using one of the Viper backends.

In this section, we first provide the reader with a brief introduction to the modular verification of Go programs using Gobra. We then introduce fractional permissions, which are used to reason about heap memory accesses and in particular allow concurrent read operations.

2.1.1 Verifying a Go program

The first step in the verification of a Go program is to write a specification of the program, describing its intended behavior. To do so, the user writes verification *annotations* for each function of their Go implementation. These annotations do not interfere with the execution of the program and are only used for verification purposes.

```
1  requires a >= 0 && b >= 0
2  ensures  product == a * b
3  func multiply(a, b int) (product int) {
4      product = a * b
5      assert product == 0 ==> (a == 0 || b == 0)
6  }
```

Figure 2.1: This function returns the product of two positive integers. On line 1, "`requires`" specifies the precondition. Here, we specify for this particular example that `multiply` may only be called with positive arguments. On line 2, "`ensures`" specifies the postcondition. Here, the postcondition states that the product is equal to the product of the two arguments. On line 5, we use an "`assert`" statement to prove that one of the parameters must be zero if the product is zero. In this case, the assertion is not necessary for the verification of the function and is only given as an example.

The main types of annotations are *preconditions*, *postconditions*, *lemmas* and *loop invariants*. Preconditions and postconditions are used to specify the behavior of functions. Preconditions specify the requirements that must be met before a function is called, and postconditions specify the guarantees that the function provides when it returns. The verification of the function is successful when Gobra can prove that the postcondition holds whenever the precondition holds. Lemmas are ghost functions, meaning that they are only used for specification purposes and are not part of the implementation. They are used to express and prove auxiliary properties of the program's specification and can be called in other verification annotations to help Gobra prove more complex properties. Loop invariants are assertions that must hold at the beginning and the end of each iteration of a loop, and are used to specify the behavior of loops. An example of a function with some verification annotations is shown in Figure 2.1.

Preconditions and postconditions enable a modular verification of the program, i.e. we verify functions in isolation and only consider the specification of the functions that they call, not their bodies. Doing so, we verify only once the correctness of a function. This greatly improves performance, as it decomposes the verification of a program into the verification of each component.

2.1.2 Permissions

Gobra supports verifying programs that manipulate heap memory. Gobra reasons about potential side-effects or their absence using *separation logic*, an extension of Hoare logic. In particular, a permission is associated with every heap location and a heap access induces a proof obligation that the current function has the necessary permission. A permission is created when allocating a heap location. Permissions allow us to deduce whether a callee might modify a heap location or not, because the callee may only modify a heap location if it has permission to do so. To specify which heap

```
1 requires acc(a, 1/2) && acc(b, 1/2)
2 requires *a >= 0 && *b >= 0
3 ensures  acc(a, 1/2) && acc(b, 1/2)
4 ensures  product == *a * *b
5 func multiply(a, b *int) (product int) {
6     product = *a * *b
7     assert product == 0 ==> (*a == 0 || *b == 0)
8 }
```

Figure 2.2: This function returns the product of two positive integers. This is the same example as in Figure 2.1, but this time reading `a` and `b` from heap locations. `acc(a, 1/2)` specifies permission of 1/2 to the heap location `a`. Notice that we require read permissions (line 1), which we can return at the end of the function (line 3). Removing line 1 would result in a verification error.

locations are accessible to a function and with which permissions, we use an accessibility predicate, denoted `acc`.

By default, a function has no permission to access heap locations. To give a function permission to access some location, we specify the corresponding permission in the preconditions. A permission of 1 to a heap location gives full permission, allowing the function to read and write to the location, and a permission of 0 gives no permission. Gobra uses fractional permissions, which defines a permission as a rational number between 0 and 1. Any strictly positive permission smaller than one only gives read permission to the heap location. There can only be a total permission amount of 1 for each heap location, which enforces that there can be either only a single writer at a time or multiple readers. To illustrate how we can use permissions in Gobra, we show in Figure 2.2 the same example as in Figure 2.1, but this time `a` and `b` are pointers to heap locations.

Another construct supported by Gobra that will be useful in this thesis is the *predicate*. A predicate is a parametrized assertion to which we give a name. It is defined with the keyword `pred`. In particular, a predicate can be used to abstract over individual permissions. For example, we could create a predicate to express permission to access an array of bytes instead of specifying the permission to each byte individually. A predicate that does not contain an assertion is called an *abstract* predicate. We will see later how they can be useful.

2.2 Verification of protocol implementations

Verifying a protocol implementation requires more work than verifying the correctness of a simple Go program. Indeed, a protocol describes a distributed system where each participant is implemented in a separate program. We can verify each participant individually, but we would like to express and prove protocol-level global properties that require the collaboration of all participant implementations.

In this section, we introduce the notion of symbolic protocol analysis to present how protocols are usually modeled when verifying their security properties. Next, we discuss the security properties that we aim to verify. We then present two existing approaches aiming at verifying protocol *implementations*. These approaches are the main basis for our work.

2.2.1 Symbolic protocol analysis

In symbolic protocol analysis, we analyze a protocol at a higher level of abstraction to verify its behavior in all possible executions. We use the symbolic model of cryptography, where we assume that the cryptographic primitives are secure, i.e. the plaintext can only be obtained from a ciphertext if decryption is performed with the correct decryption key. Additionally, we assume that all operations are performed on symbolic *terms* instead of bytes.

Furthermore, we consider a Dolev-Yao [7] attacker present in the network. This attacker can perform arbitrary operations on this term level, has full control over the network (including reading and sending any message), and can corrupt any participant (which means that the attacker learns all the terms contained in a participant's state).

A common way for reasoning about protocols in this setting is to consider all possible *traces* of a protocol. A trace records the sequence of protocol events in a particular run of the protocol. Security properties can be expressed as a logical expression about a trace, and then verified by checking whether it holds for all possible traces of the protocol. Automated provers like Tamarin [10] and ProVerif [3] can verify trace-based security properties by analyzing all possible traces.

In this work, we aim to verify security properties for protocol *implementations*. While verifying protocol *models* is useful to prove the security of the protocol design, it does not guarantee that an implementation of this protocol is free of security vulnerabilities induced by common programming errors or the incorrect implementation of the protocol design. When considering the verification of protocol implementations, traces can also be used. By expressing an invariant over the trace and proving that this invariant is maintained by all executions, we obtain properties that hold for all possible traces. We call this invariant the *trace invariant*. Then, we verify security properties by showing that these security properties are implied by the trace invariant. We detail the security properties that we aim to verify in the next paragraph.

2.2.2 Security properties

We have mentioned that using a trace and maintaining a trace invariant allows us to prove some security properties. In particular, we want to prove

forward secrecy and post-compromise security for certain protocols. In this subsection, we explain what these properties are and why they are important.

Forward secrecy is a security property expressing how past communications are protected from a future compromise. A protocol is forward-secure if compromising the current state of a participant does not reveal past communications, meaning that the attacker cannot decrypt previous messages. However, forward secrecy does not protect future communication. The attacker could use the compromised knowledge to impersonate a legitimate participant and read all future communications. This is why we are interested in the next property, post-compromise security.

Post-compromise security is a security property expressing how future communications are protected from a past compromise. First, notice that post-compromise security is not achievable after the *unrestricted* compromise of a participant. After such a compromise, other participants have no guarantee whether they are communicating with the compromised participant or the attacker because the attacker has access to the same secrets as the compromised participant and can thus impersonate this participant. We instead consider the formalized notion of post-compromise *via state* [5]: a participant can communicate securely with a peer even when a past state of the participant is revealed to the attacker, as long as the participant's current state is not revealed to the attacker and some secret data remains available exclusively to the participants. If the participant's current state is revealed to the attacker, post-compromise security via state trivially holds but we get no guarantee of the security of future communications. To be able to prove this property, we must be able to prove that a certain state in the past, to which an attacker has gained access, does not contain the secret data on which future communication depends.

In the following, we will introduce two existing approaches in this verification setting aiming at verifying strong security properties for protocol *implementations*. These approaches are the main basis for our work.

2.2.3 DY*

DY* [2] is a framework for proving security properties about protocol implementations written in the F* programming language. Using a particular code structure and a particular way of storing program state, DY* is able to account for some implementation-level specificities.

While DY* uses a trace to record protocol events, it is not *ghost*, i.e. the trace is not only used for verification purposes. In particular, each participant's state is stored in a serialized way on the trace. When a protocol participant is taking a protocol step, it first reads its serialized state from the trace, deserializes it, performs the step, and then saves its new serialized state to

the trace. This means that only protocols with this particular way of storing program state can be verified with DY*. This also affects the runtime behavior of the protocol, as the trace must exist and be used to store and retrieve the program state.

This trace is append-only and existing entries cannot be modified or deleted. In addition to state changes, the trace records all operations that are important for proving security properties (nonce generation, sent messages, and corruption). Ultimately, this global trace contains each participant's current and all past states, and provides a global view over the entire distributed system. DY* achieves modular reasoning by specifying a trace invariant and verifying that each function modifies the global trace only in ways that maintain the trace invariant. Global security properties can then be proved from this trace invariant.

On the global trace, each state is annotated with a *session state identifier* to indicate to which participant and protocol session it belongs. This models the fact that a participant may be involved in multiple independent protocol sessions simultaneously. The session state identifier includes a *version* that distinguishes between different phases within a protocol session (further explanations about versions will be given in section 3.1.2). Each participant's state comprises several values that together constitute the program state of the participant. By value, we mean here and in the rest of this thesis a piece of data that is stored on the heap. Each of these values is assigned a *secrecy label*, indicating which participant is allowed to read a value and at what times. These secrecy labels enable modular reasoning about secrecy because a secrecy label provides an over-approximation of which participant states might contain a particular value. A value can be made accessible to a participant p, to only some specific session s of a participant, or even to some specific version v of a session. A secrecy label is most often defined as a list of everyone who can access the value[1]. For example, a value with a secrecy label $[(\text{Alice}), (\text{Bob}, 3), (\text{Charlie}, 2, 7)]$ can be accessed by Alice at any time, by Bob in session 3, and by Charlie when his second session is in version 7. We will keep this notation of secrecy labels throughout this thesis. Additionally, the secrecy label of a value can also be *Public*, which means that the value is accessible to everyone, including the attacker.

The version label attributed to each session is used to introduce a notion of temporality. Initially, all session versions are 0 and are incremented at some times to represent new protocol phases. A value with a secrecy label $[(p, s, v)]$ can only occur in the specific phase of the protocol when the session s of participant p has version v. DY* enforces this restriction with a suitable invariant over states, which they enforce whenever state is stored on the trace.

[1] A secrecy label can also be defined by the union or the intersection of participant, session and version identifiers, which we omit for brevity.

Thus, only data from the current version can be stored, ensuring that neither outdated keys nor future keys are present in memory. Hence, these versions are used to define the period of existence of values. Building on this, they are able to prove forward secrecy and post-compromise security for protocols like Signal.

A hierarchy between secrecy labels is necessary to express some properties. For this, DY* introduces a `CanFlow(l1,l2)` relation specifying that a less restrictive secrecy label `l1` can flow to a more restrictive secrecy label `l2`. For example, the label $[(\text{Alice}), (\text{Bob})]$ can flow to the more specific $[(\text{Alice})]$. And the label $[(\text{Alice})]$ can flow to a single session $[(\text{Alice}, 3)]$. In particular, DY*'s state invariant expressing that a value should be readable in the current context can be specified using the `CanFlow` relation: the value's label should flow to the current participant, session and version.

Protocol implementations written in F* and verified with DY* are executable but require a special runtime environment that provides access to the trace such that a participant's state can be stored and retrieved. Additionally, the DY* framework is composed of a library, containing protocol-independent parts that only need to be verified once, and can then be reused across different protocols. This significantly reduces the overall verification time.

2.2.4 Modular verification of existing implementations

The DY* framework can only verify protocol implementations written in the functional F* language that adhere to certain assumptions about their structure and program state storage. However, these assumptions do not hold in general for existing implementations. Arquint et al. [1] present a methodology for verifying existing heap-manipulating protocol implementations. This methodology is agnostic to the programming language and relies only on standard features present in most separation logic-based verifiers. In the following, we will focus on Go implementations and the Gobra [16] verifier.

This methodology is inspired by DY* [2] and also uses a global trace to provide a global view of the entire system. To address the issue of arbitrary code structure in existing implementations, the trace is treated as a concurrent data structure. This treatment ensures that all possible interleaving of events are considered, without relying on any code structure assumptions. As existing implementations manage the program state in their unique ways, we cannot assume that they store the state on the trace or rewrite implementations to do so. Instead, *local invariants* are used to relate the program state stored at a participant to events on the trace. Because the global trace is a shared structure, each participant maintains its own local *snapshot*, which is a local copy of the global trace. Since the global trace can only grow, a snapshot is a prefix of the global trace. The global trace only records a sequence of

events corresponding to high-level operations (similar to DY*, except for the state storage entry) that must maintain a trace invariant, which is used to prove global security properties. Unlike DY*, the global trace is a *ghost* data structure for verification-only purposes, which will be erased before compilation. As such, the global trace has no impact on the runtime behavior or performance.

Furthermore, separation logic's resources are used to prove injective agreement, which is to the best of our knowledge not possible with DY*. To do so, they use that separation logic's resources are not duplicable, so if an assertion includes two resources, they are distinct. This can be used to express the uniqueness of an event and prove that an implementation detects when an attacker replays messages, which is necessary to satisfy injective agreement. Because DY*'s invariant is expressed in first-order logic, it is unclear how they could reason about the uniqueness of particular events.

Similarly to DY*, this methodology comes with a library, called the Reusable Verification Library, which allows the reuse of protocol-independent parts (verified only once) across different protocol implementations. Because they are using Gobra, verification is based on separation logic, which allows us to reason about heap manipulations. Currently, any creation of an array of bytes (i.e. creation of nonce, keys, etc.) is controlled by a memory predicate `Mem`. The `Mem` predicate is used to abstract over the memory of a byte array and thus specifies permissions to every byte in the array. The predicate body is shown below, for illustration purposes only[2]:

```
1 pred Mem(s []byte) {
2     forall i int :: 0 <= i && i < len(s) ==> acc(&s[i])
3 }
```

Permission to this predicate is then required by all library functions performing operations on the associated byte array.

Each created value is also assigned a secrecy label, like in DY*, to specify allowed readers. However, this methodology does not support versions in secrecy labels. Without this fine-grained temporality, it is not possible to prove forward secrecy and post-compromise security for protocols like Signal.

2.3 Cryptography

In this section, we explain two important cryptographic concepts: AEAD encryption [14] and Diffie-Hellman key exchange [6]. These notions will play

[2]The `Mem` predicate is in fact abstract in the library, meaning that clients of the reusable verification library cannot get direct access to the individual bytes of the array and instead have to perform all operations via corresponding library calls.

a major role in the rest of this thesis. However, the discerning reader can skip this section if he or she is already familiar with these basics.

2.3.1 AEAD encryption

In the protocols that we aim to verify and that we will present later, authenticated encryption with associated data (AEAD) [14] plays an important role in the communication between participants. AEAD is a symmetric encryption scheme. It provides confidentiality, meaning the encrypted message cannot be read by an attacker who does not know the key. It also provides authenticity, meaning that an active attacker cannot modify the encrypted message or associated data without being detected and that the receiver can verify that the message was sent by the expected sender.

AEAD encryption takes as input a key, a plaintext, and optionally some associated data, and returns a ciphertext. While the message is sent encrypted and authenticated, the purpose of the associated data is to send in clear some data that is not confidential but that is authenticated. For example, we will in the following use the associated data to send public keys in an authenticated way. Then on the receiver side, AEAD decryption takes as input the same key, the ciphertext, the received associated data, and returns the plaintext if decryption succeeds.

2.3.2 Diffie-Hellman key exchange

A Diffie-Hellman key exchange [6] is a public-key protocol that allows two participants to agree on a shared secret key over an insecure channel. Each participant p has a secret key s_p and a public key g^{s_p}, where g is a primitive root modulo a prime number[3]. In a Diffie-Hellman key exchange, participants Alice and Bob first exchange their public keys $g^{s_{Alice}}$ and $g^{s_{Bob}}$ over the insecure channel. Then, Alice computes a shared secret key $(g^{s_{Bob}})^{s_{Alice}} = g^{s_{Alice} \cdot s_{Bob}}$ by combining her secret key and Bob's public key, and Bob computes the same shared secret key $(g^{s_{Alice}})^{s_{Bob}} = g^{s_{Alice} \cdot s_{Bob}}$ by combining his secret key and Alice's public key.

This key exchange will be extensively used in the protocols that we aim to verify and that we will present later.

[3] In the following, we will not mention the modulo prime number for brevity.

Chapter 3

Design

In this chapter, we explain how we extended the work of Arquint et al. [1], presented in section 2.2.4, to express time-sensitive values and to enforce their deletion at the right time. Practically, we extended their Reusable Verification Library in order to provide someone verifying a protocol implementation with the possibility of proving security properties like forward secrecy and post-compromise security. In the following sections, we will refer to this person as *the developer*, who uses this library to verify a protocol implementation.

We start by giving an overview of our goal and a high-level idea of our methodology to achieve it. Then, we introduce counting permissions, as the core tool on which our deletion mechanism is based. We continue by explaining how we extended the library to implement our deletion mechanism. Finally, we give a brief look into an alternative approach that would have been simpler, but that is not currently supported by Gobra [16].

3.1 Overview

This section gives an overview to better understand our goal and the high-level idea of our methodology to achieve it. With the aim of verifying properties like forward secrecy and post-compromise security for a protocol implementation, we need to model the fact that some ephemeral values should only exist for a limited amount of time. Additionally, we need to enforce that these values are deleted before the end of their time frame.

We start by explaining the goal of our methodology by introducing a relevant protocol, the Diffie-Hellman Ratchet [13], and the security properties we want to prove about it. Then, we explain how we introduce a notion of temporality in the methodology, by adding *versions*, which define fine-grained time frames during which certain values are present in memory. Finally, because

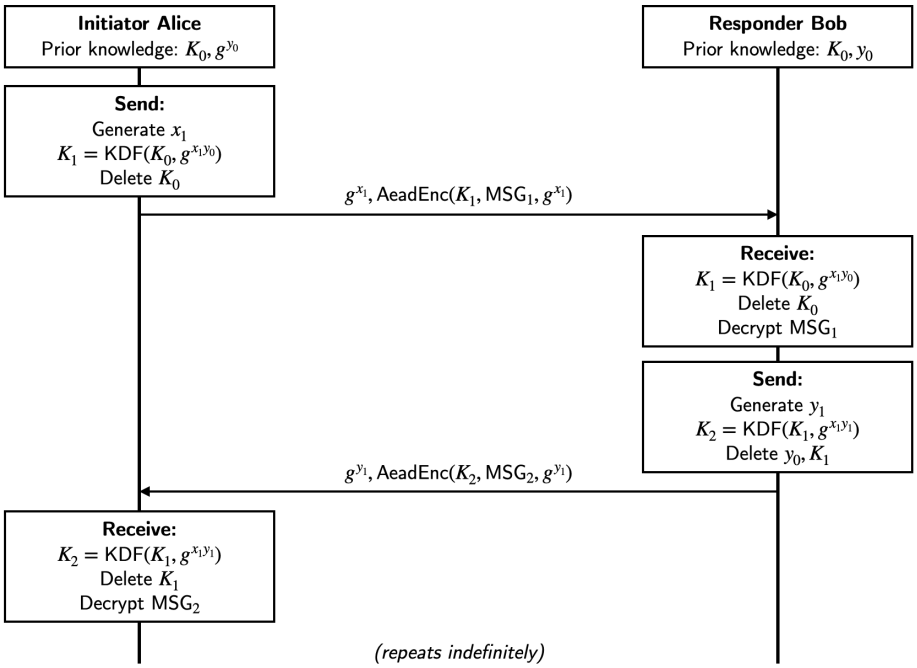

Figure 3.1: The Diffie-Hellman (DH) Ratchet [13] is a continuous key agreement protocol between two participants, Alice and Bob, which repeatedly performs Diffie-Hellman key exchanges to encrypt each message with a new key. Here, `AeadEnc(K, MSG, ad)` refers to the AEAD encryption of the message `MSG` with the key `K` and the associated data `ad`. `KDF(a, b)` refers to a key derivation function that derives a new key from two inputs `a` and `b`. This figure is inspired by a similar figure from the DY* paper [2].

versions specify that a value should only be able to exist in some time frame, we reason about how to enforce that it is deleted before this time frame ends.

3.1.1 Goal

We aim to verify protocol implementations that frequently renew their communication keys to provide strong security properties. A notable example is the Signal Double Ratchet protocol. We consider a slightly simpler protocol, on which Signal is based, involving a single ratchet; the Diffie-Hellman Ratchet.

Diffie-Hellman Ratchet

The Diffie-Hellman (DH) Ratchet [13] is a continuous key agreement protocol, which repeatedly performs Diffie-Hellman key exchanges to encrypt each message with a new key. It is presented in Figure 3.1.

The protocol starts with an established communication key K_0, mutually shared by both participants. The initiator uses the responder's public key and a newly generated secret key to compute a shared Diffie-Hellman secret. This new shared secret and K_0 are input to a key derivation function (KDF) to derive a new communication key K_1, which is used to encrypt a payload, i.e. $\texttt{MSG}_1$. This step of generating a new key from the previous one is called a *ratcheting step*. Behind this name, there is the idea of a ratchet, which one could turn in one direction to generate new keys, but that cannot be turned back, meaning that previous keys cannot be derived from the current one. Then, K_1 is used to encrypt the message sent to the responder, accompanied by the initiator's new DH public key g^{x_1}. The responder can compute the same Diffie-Hellman shared secret and take the same ratcheting step to obtain K_1, then use K_1 to decrypt the message. This iterative process is repeated for each message, i.e., every payload is encrypted with a different key obtained by ratcheting the previous one.

After this ratcheting process has been performed and has given us a communication key K_n, the previous key K_{n-1} is no longer needed and is deleted. Deleting the previous key is an essential step for a protocol implementation to achieve forward secrecy, which we explain in the following.

Security properties

The DH Ratchet protocol is designed to provide strong security properties, such as forward secrecy and post-compromise security. We define these properties below and explain how the DH Ratchet protocol satisfies them.

Forward secrecy. The DH Ratchet protocol is designed to be secure against an attacker recording all previous encrypted messages and obtaining a shared secret or a communication key at some point. If the attacker compromises a participant, for example, Alice, they may be able to decrypt some messages using the keys and secrets stored in Alice's memory. If Alice keeps storing all previous secrets and session keys, then the attacker would be able to decrypt all previous messages that they previously observed on the network. This is why it is crucial for Alice to delete previous secrets and communication keys as soon as she has derived the new ones. Indeed, if previous keys are correctly deleted from Alice's memory, then the attacker may only be able to decrypt the last message and not all previous ones.

Therefore, the DH Ratchet protocol satisfies forward secrecy. This property is enabled by two main factors: cryptographically preventing past communication keys from being derived from long-term secrets and current communication keys, and securely deleting previous keys.

Post-compromise security. Additionally, the DH Ratchet protocol is designed to be *self-healing*, meaning that it should allow communication to resume securely at some point after a compromise. This is the property of post-compromise security. Recall that post-compromise security is not achievable after the *unrestricted* compromise of a participant, but we instead consider that some secret data remains available exclusively to the participants after the compromise (post-compromise *via state* [5]). In the DH Ratchet protocol, we consider that the attacker compromised Alice's K_n communication key *after* she derived the new K_{n+1} key, and after she and her peer Bob have deleted their DH private key they used to compute K_{n+1}. At this point, the attacker cannot obtain K_{n+1} because they cannot have access to the associated DH shared secret.

Therefore, all future communication keys are safe from the attacker, so the DH Ratchet protocol satisfies post-compromise security via state. Post-compromise security requires that past states do not contain secret data on which future communication depends. This is why it requires fine-grained reasoning about the data occurring in a participant's state at a particular point in time, and secure deletion of old data.

Our methodology to prove these two strong security properties, forward secrecy and post-compromise security, requires a notion of temporality because we have to specify the lapses of time during which certain keys are possibly present in memory. Outside these lapses, keys must not be present in memory because they have either not been generated yet or have already been securely deleted.

3.1.2 Versions

To introduce a notion of temporality, we divide the time axis into successive time frames. We number these time frames with integers, starting at 0, and call them *versions*. Similarly to what is done in DY* [2], each participant's session is given a version field, which keeps track of the current version of the protocol. At this point, we can uniquely identify a time frame in a protocol session by the triplet (p, s, v), where p is the participant executing the protocol session s, and v is the version defining the time frame. We call this triplet a version identifier.

We then use these identifiers to specify the time frames during which certain terms are present in memory. Recall that each term, e.g. a key or a nonce, is assigned a secrecy label stating who can access it. While these labels could previously only be defined from participant or session identifiers, we now allow them to be defined from version identifiers (p, s, v) as well.

We distinguish two cases: *versioned* and *unversioned* terms. A term is said to be versioned if it is made to be accessible only in one or several specific

versions of the current protocol session. Conversely, a term is said to be unversioned if it is made to be accessible by the current participant or session, i.e. in all versions of the current protocol session. For example, considering the current session s of participant p, a term with secrecy label $[(p, s, 0), (p')]$, where p' is any other participant, is versioned because it is only readable in version 0 of the current session s. But a term with secrecy label $[(p)]$ or $[(p, s)]$ is unversioned because it is readable in all versions of the current session s. Additionally, we define a i-versioned term as a versioned term that is accessible in version i.

The existing methodology already enforces that unversioned terms can only be accessed by participants or sessions allowed by their secrecy label. A key property of our methodology extension is to ensure that versioned terms can only be accessed when the session is in a version allowed by their secrecy label. Intuitively, this means that an ephemeral key, defined as a versioned value, should only be accessible during a limited time frame defined by the versions allowed by its secrecy label. The crucial difference of this work compared to the existing methodology is that it allows non-monotonic secrecy labels, i.e. a term being readable now might not be readable in all future timepoints.

On a high level, enforcing the time-limited existence of versioned terms requires two kinds of checks. First, we enforce when creating a versioned term that it should be readable in the current version of the protocol. Second, we enforce that when increasing the version of a session, all versioned terms that are no longer accessible in the new version have been deleted. While the first check can be easily implemented similarly to the existing check for unversioned terms, the second check requires more work. Indeed, using Gobra, there is no trivial way to check a condition on all versioned terms that are stored in memory at a particular time point.

This second check is to ensure that ephemeral keys are deleted when they are no longer needed. Let us discuss this in more detail.

3.1.3 Enforcing deletion of old data

Before increasing the version of a session, we have to securely delete all versioned terms that are no longer accessible in the new version. By doing so, we more generally ensure that terms are only accessible in the versions allowed by their secrecy label.

We discuss next the solution used in DY*. However, this solution is not applicable in our setting, which does not restrict the way implementations store application state. Hence, we afterward present a more generic solution.

Existing approach in DY*

In DY* [2], a participant uses a storage API to store all of its knowledge on the trace. In particular, recall from section 2.2.3 that a participant stores the knowledge of ongoing sessions into an array, composed of each serialized session state, and each session is given a version number. Upon updating some session state, the participant can also update and increment the version number of the session. An invariant over the state makes sure that only data from the session's version can be stored, ensuring that old keys cannot remain in memory. Practically, this means that previous keys are effectively deleted from the state when the session's version is incremented.

This solution is however not applicable to us because, as explained previously, we do not restrict the way implementations store application state, and we use a ghost trace, removed at runtime, which cannot store program data. Because we do not store the program state on the trace, it makes it impossible to express the same invariant over the state as in DY* in our case. Furthermore, DY* enforces the invariant over state only at certain time points, namely when the state is stored on the trace, without taking into account the state in-between. Finally, it is not sure if DY* enforces that outdated keys are *securely* deleted, meaning explicitly zeroed out from memory. In the next part, we present our generic approach to enforce the deletion of old data.

Our approach

As we cannot simply iterate over the full session state to remove outdated keys like in DY*, we present a new approach to ensure that versioned values get safely deleted before their version expires. Note that our approach only *extends* the existing approach of Arquint's et al. [1], which means that our modified library can still be used to verify unversioned protocols in the same way as before.

The intuition of the methodology is to let the developer (that is verifying a protocol implementation) choose when to delete a versioned value. For this purpose, we provide a *secure* deletion function that deletes a value by zeroing out the memory and ensuring that this write operation is not optimized away by the compiler. The crucial aspect now is to verify that, before the session version is incremented, the secure deletion function has been called for all versioned values that will no longer be accessible in the new version.

Intuitively, we could solve this problem with a counter. Starting with version i, we use the counter $c_i := 0$. Each time we create a versioned value readable in version 0, we increment the counter $c_i = c_i + 1$. And each time we delete a versioned value readable in version 0, we decrement the counter $c_i = c_i - 1$. When we increment the version of the session, we check that $c_i = 0$, meaning that all versioned values readable in version i have been deleted.

However, this approach cannot be trivially implemented in our case because we cannot just use a counter *variable* that the developer could access at will: they would just have to manually set it to 0 before incrementing the version of the session, and the counter check would be meaningless. We would have to ensure that the developer cannot directly manipulate the counter, which may be difficult to achieve. Additionally, we want to support concurrency which would require us at least to use an atomic counter or to protect the counter by a lock. Instead, to solve these challenges without the limitations of a counter variable, we will create a mechanism based on *counting permissions* [15]. Those will be explained in the next section, before using them to design our deletion mechanism.

Note that with this approach, one could think that the developer could just never increment the version of the session, and thus would never have to delete any versioned value. While this is true, if the developer wants to verify meaningful properties like forward secrecy, they will have to increment the version at some point to express the property in terms of the version.

3.2 Counting permissions

Because keeping track of created and deleted values using a counter variable could be easily bypassed by the developer, we base our work on counting permissions [15]. While counting permissions follow a similar idea as a counter, they overcome the limitations mentioned earlier and are suitable for our use case, i.e. they support arbitrary representations of the application state and can be adapted to support concurrency.

We start by introducing the notion of counting permissions. Because this permission model is not available in the current version of Gobra, we then explain how we can obtain similar functionality using the existing permission system.

3.2.1 Introduction

Counting permissions are a permission model where permission shares are valued in $\mathbb{Z} \cup \{u\}$, where u represents the identity element of the share addition operation. A share of value 0 means full permission, a share of value u means no permission, and any other value means partial permission. A non-negative share $n \geq 0$ is called a token *factory* and can be split, for any integer $k > 0$, into another factory $n + k$ and an equivalent amount of negative token *bundles* of value $-k < 0$. A token factory and some amount of token bundles can be summed, and the result is always non-negative (because the subtracted bundle value had to be added to the factory before).

In particular terms, counting permissions can be used as a counter: a full permission (0) can be split into a factory 1 and a bundle -1 representing incrementing the counter by one. A second increment would create a factory 2 and a second bundle -1. Invertly, decrementing the counter could be done by summing the factory 2 with a bundle -1, resulting in a factory 1.

We therefore annotate the library functions such that they keep track of the number of versioned values, i.e. increment the counter when creating a new versioned value and decrement the counter when safely deleting such a value. Unversioned values do not need to be tracked as they do not have to be safely deleted because they can be present in memory for the entire (remaining) execution of the protocol.

To implement a deletion mechanism using the counting permissions model, we use an abstract predicate that we call `guard(i int)`, which takes a version number as an argument, and on which we initially have full permission 0. The library functions that create a versioned term readable in version i require access to the token factory $n \geq 0$ of `guard(i)`, and return an incremented share $n + 1$, *without* returning the token bundle -1. Invertly, the secure deletion function returns a token bundle -1 upon deletion of a versioned term readable in version i. Therefore, if the developer deletes all i-versioned terms, they will obtain as many token bundles -1 as the value of the token factory n. Summing them together will result in a full permission $n = 0$. This means that we know that no i-versioned values exist in memory when we have full permission on `guard(i)`.

The major difference with a simple counter variable is that the developer *cannot* create a -1 token bundle the way they could just decrement the counter variable. They *have to* delete all i-versioned terms and cannot circumvent the deletion mechanism.

To support concurrency using counting permissions, we could protect the token factory with a (ghost) lock. Then, splitting the factory to obtain a bundle, or summing the factory with a bundle, would require acquiring the lock. This lock would however complicate the entire reasoning as the lock and each thread would have to be aware of the amount of created versioned values. Thus, in the next section, we present an alternative to counting permissions that, in addition to supporting concurrency without a lock, is supported by Gobra.

3.2.2 Implementation in Gobra

Gobra [16] currently does not support counting permissions but only fractional permissions, which were explained in section 2.1.2. Recall that in this model, permission shares are valued in $[0, 1] \cap \mathbb{Q}$, a share of value $0/1$ means

no permission, a share of value 1/1 means full permission, and any other value means partial permission.

Because fully encoding counting permissions using fractional permissions is not trivial, we instead restrict ourselves to a solution based on fractional permissions that tracks the number of versioned values with the help of our annotated library functions.

Idea

We introduce below two abstract predicates that we use to implement our deletion mechanism.

```
1 pred guard(v uint32)
2 pred receipt(key []byte, v uint32)
```

The `guard` predicate can be seen as the token factory and is initially given with full permission (1/1) to the developer. The `receipt` predicate can be seen as a token bundle. The idea is for all library functions that create a versioned value `key` readable in version `v` to require partial permission to `guard(v)`, and to return the same amount of permission of `receipt(key, v)`. Then, upon deletion of `key`, the secure deletion function requires some amount of permission of `receipt(key, v)` and returns the same amount of permission of `guard(v)`. We illustrate these explanations with the Gobra specification of two basic `Create` and `Delete` functions given below. Note that in Gobra, a function argument annotated with `ghost` is an argument for verification purposes only, i.e. it is not part of the program state and is removed at runtime.

```
1 requires acc(guard(version), versionPerm)
2 ensures  acc(receipt(key, version), versionPerm)
3 func Create(ghost version, ghost versionPerm) (key []byte) {/*...*/}

4 requires acc(receipt(key, version), versionPerm)
5 ensures  acc(guard(version), versionPerm)
6 func Delete(key []byte, ghost version, ghost versionPerm) {/*...*/}
```

In practice, the developer may have to create several `v`-versioned values and will consume some amount of permission of `guard(v)` for each of them. For each created value, they will receive the same amount of permission of `receipt(key, v)` as the amount of permission of `guard(v)` they consumed to create it. When they want to increment the version of the session, they will have to delete all `v`-versioned values and will consume all `receipt(key, v)` fractional permissions. They will receive the same amount of permission of `guard(v)` as the consumed `receipt(key, v)` permission for each deleted value. Therefore, we know that no versioned value readable in version `v` remains in memory when we have full permission on `guard(v)`. The following lemma expresses this property:

Lemma 3.1 *We denote* `perm` *the function taking a predicate and returning its available permission fraction. If there exists a* $v \in \mathbb{N}$ *such that* $\texttt{perm}(\texttt{guard}(v)) = 1$*, then* v *is the current session's version, and no* v*-versioned value exists in memory.*

Consequently, we require full permission on `guard(v)` to call the function that increments the version of the session.

Additionally, this approach naturally supports concurrency, because separation logic allows us to split the permission of the `guard` predicate into several fractions and hand each fraction to a different thread. Doing so, each thread can independently perform creation and deletion operations on versioned values because they only require a fraction of the permission of the `guard` predicate.

Similarly to counting permissions, the developer cannot bypass the deletion mechanism because they cannot obtain `guard` or `receipt` fractional permissions without going through the library functions. This is assuming that the developer does not use Gobra's `inhale` statement to bypass the permission system.

Note that the receipt predicate takes the created byte array `key` as an argument. This is used at deletion time to verify that we delete the actual value associated with the receipt, before returning the `guard(v)` permission fraction. Otherwise, the developer could simply delete arbitrary (unversioned) values to transform some `receipt(key, v)` permission fraction into a `guard(v)` permission fraction, and thus violate lemma 3.1.

Choosing the right permission amounts

With these library specifications, the developer has some flexibility in choosing fractional permission amounts when invoking library functions. Indeed, upon creating or deleting a versioned value, the developer chooses the permission amount of the `guard` or `receipt` that will be consumed. Picking too large permission amounts that would result in a contradiction of lemma 3.1 correctly results in verification errors. The chosen permission amounts thus only affect completeness.

When creating versioned values, the developer has to choose small enough permission amounts of the `guard` predicate to consume so that all desired values can be created. For example, if they want to create 3 versioned values, but consume $1/2$ of `guard` for the two first values, they will not be able to create the third value because they will not have enough permission of `guard` left. This does not affect the soundness of the methodology but simply prevents the developer from implementing their goal. It is therefore in their interest to carefully choose permission amounts. Note that unboundedly many versioned values can be created even when it is not statically known how many values will be created in the implementation. For example,

this is doable by consuming 1/2 of the `guard` predicate permission for the first created value and dividing the consumed permission by 2 for each subsequent created value, i.e. consuming 1/4 for the second value, 1/8 for the third value, etc.

When deleting a versioned value, the developer has to specify the same permission amount of the `receipt` predicate to consume as the amount that was created by the library when creating the versioned value. Indeed, automatically picking the available permission amount of `receipt` in the current context could be insufficient in cases where some permission amount of `receipt` has leaked or is hidden[1], and is either way not currently supported by Gobra. This is why we require the developer to manually specify the right permission amount of `receipt` to consume. If the developer calls the delete function and specifies a smaller amount than the permission amount of the `receipt` predicate that was created by the library, while there will be no immediate verification error, the developer will not be able to recover the full permission to the `guard` predicate. This will prevent them from incrementing the version of the session. Again, the methodology's soundness remains unaffected, but the developer has to be careful when specifying the permission amount of `receipt` to consume to avoid completeness issues. The following lemma expresses this property:

Lemma 3.2 *Let v represent the current version of the session and n the number of v-versioned values in the current state. $\forall i \in [\![1, n]\!]$, we denote key_i the i-th v-versioned value. Additionally, we assume the developer always deletes a versioned value specifying the same permission amount as the one they used to create it, and more generally that no permission amount of the predicates `guard` and `receipt` has leaked or is hidden. Then, we can express the following property:*

$$\left(\texttt{perm(guard(v))} + \sum_{i=1}^{n} \texttt{perm(receipt(}key_i\texttt{, v))} \right) = 1$$

To conclude, we have obtained a satisfactory approach to track the number of versioned values, which meets all of our requirements. Our approach is, contrary to counting permissions, supported by Gobra. As we wanted, our approach also supports arbitrary representations of the application state, concurrency and unboundedly many versioned values. Despite giving more flexibility to the developer, this method is sound and constitutes the core of our deletion mechanism.

[1]We say that some permission amount to a heap location or a predicate x *leaks* when the total available permission amount of x in the current context is not fully returned to the caller in a postcondition of the current function, so a permission amount of x is lost forever. We say that some permission amount of x is *hidden* when it is *folded* in a predicate y, meaning that a permission amount of x is replaced with a permission amount of y.

3.3 Extension of the library

We introduced a general mechanism to enforce the deletion of versioned values before the end of their time frame, defined in terms of versions allowed by their secrecy label. This mechanism is central to the methodology. In this section, we explain how we integrate it in Arquint's et al. Reusable Verification Library [1], to provide the developer with functions handling versioned values.

We start by explaining how we store and access the current version of a session. Then, we explain how we adapted the functions that create a value, e.g. a nonce, to support versioned values according to our methodology. Similarly, we present our secure deletion function made to delete those versioned values. We continue by introducing two special cases that required particular effort to support versioned values and fit our methodology: key ratcheting and encryption/decryption. Afterward, we describe our function for increasing the version of a session when all "old" values have been deleted.

Finally, we give a glimpse of an alternative approach to implement a similar deletion mechanism, based on obligations, which can be used instead if the targeted verifier supports obligations.

3.3.1 Storing the current session version

We will later see several library functions that have to compare a version appearing in a secrecy label to the *current* version of the session. Retrieving the current version from the global trace directly is not possible because, as it is a concurrent data structure, there might be interleaving operations by other participants or sessions between the moment we retrieve the version and the moment we use it. However, we assume that each session has its own version, which can only be altered by this session. Hence, we turn the version into a local field within the library that can only be altered by corresponding library calls.

Currently, the library already handles the storage of the current protocol participant and session. These are stored in a field called `owner` that is either a participant identifier (p) or a session identifier (p, s), where p is a participant and s is a session. This field is used in the concurrent data structure, as a key to a dictionary mapping identifiers to their corresponding snapshot, where we recall that snapshots are local copies of the global trace. Since `owner` is used as a key to the snapshots mapping, we cannot include the version in the `owner` field because this would result in a different map lookup whenever the session's version is incremented.

Therefore, we store the current version of a session in a separate field, as an

```
requires versionPerm >= 0
requires versionPerm == 0 ==>
    CanFlow(l.Snapshot(), nonceLabel, Readers(set[p.Id]{l.Owner()}))
requires versionPerm > 0 ==>
    acc(guard(l.Version()), versionPerm) &&
    l.Owner().IsSession() &&
    CanFlow(l.Snapshot(), nonceLabel,
        Readers(set[p.Id]{l.OwnerWithVersion()}))
ensures  err == nil && versionPerm > 0 ==>
    acc(receipt(nonce, l.Version()), versionPerm)
ensures  err == nil ==> Mem(nonce)
func (l *LabeledLibrary) CreateNonce(ghost nonceLabel SecrecyLabel,
    ghost versionPerm perm) (nonce []byte, err error) {
    // ...
}
```

Figure 3.2: Specification of `CreateNonce` showcasing the changes to support versioned values. Preconditions, postconditions and arguments that are not relevant to the changes have been omitted. `CanFlow` represents the flow relation between secrecy labels, as defined in section 2.2.3, and takes a snapshot of the global trace and two secrecy labels as arguments.

integer. The session is stored as part of the library state, which makes it easily accessible to all library functions. For a function taking a `LabeledLibrary` struct `l`, holding the state of the library, the current version of the session will be referred to as `l.Version()` in the future code figures. The owner is similarly obtainable using `l.Owner()`. Additionally, it is often useful when working with secrecy labels to know the complete version identifier (p, s, v). This is the combination of the owner and version fields, which is returned by `l.OwnerWithVersion()`.

3.3.2 Creation of versioned values

It is easy to categorize functions that create values: they all return full permission to a new memory predicate `Mem` of the created value. The library provides three functions to create values: `CreateNonce` to create some random value, `GeneratePkeKey` to create a public/private key pair, and `GenerateDHKey` to create a secret key for Diffie-Hellman key exchange. Those three functions are implemented very similarly and have required the same changes to support versioned values. Therefore, we will only discuss the `CreateNonce` function in this section. Its simplified specification is provided in Figure 3.2.

This function takes a `versionPerm` argument, which is required to be non-negative. The developer can use this argument to specify whether they want to create a versioned or unversioned nonce. Choosing 0 means that the nonce will be unversioned, and choosing a strictly positive value means versioned. In both cases, when the function completes with no error, it returns full permission of the memory predicate `Mem(nonce)`, expressing that the nonce is now stored in memory.

Unversioned nonce

In this case, the developer chooses 0 for the `versionPerm` argument. As required by lines 2-3, the provided secrecy label has to flow to the library owner because otherwise, the developer could create a nonce that they are not allowed to read, contradicting the modeling of corruption. Depending on if sessions are used, the library owner is either a participant (p) or a session (p, s) identifier. The other preconditions do not have to be satisfied because they are only relevant to the versioned case.

Because no version identifier (p, s, v) flows to (p, s) or (p), we know when verifying this precondition that the nonce label must be composed of some non-versioned identifier, e.g. is unversioned. Therefore, when the developer chooses 0 for the `versionPerm` argument and proves that the nonce label is unversioned, then the library behaves as the original implementation and creates an unversioned nonce, without consuming any permission of `guard`.

Versioned nonce

In this second case, the developer chooses a strictly positive `versionPerm` value. This value is used to specify the amount of permission of `guard` to consume (line 5), as discussed in section 3.2.2. Additionally, the library owner must be a session identifier (line 6) and not a participant, because a version identifier (p, s, v) requires a session s to be defined. Finally, the nonce label must flow to the library owner *at the current version* (lines 7-8). This is to at least ensure that the created nonce is readable by the current session and version.

However, the flow relation is not enough to ensure that the nonce label is actually versioned. To do so, one would have to prove that the nonce label *cannot* flow to the library owner (without version), as shown in this precondition:

```
1  requires versionPerm > 0 ==>
2      !CanFlow(l.Snapshot(), nonceLabel, Readers(set[p.Id]{l.Owner()}))
```

Establishing this precondition is however not possible because it would entail proving that no reader specified in the nonce label has been corrupted. Indeed, if one of the readers had been corrupted, then the nonce label would flow to *Public*, so it would flow to any secrecy label, including the library owner. Therefore, we decided to not enforce this condition, as it is not necessary to ensure soundness of the methodology. If the developer "accidentally" creates a nonce with an unversioned label, but uses a strictly positive `versionPerm` (meant to be used for versioned values), verification will not fail. However, they will be bound by the constraints of the deletion mechanism. For now, this means that they will be forced to safely delete their unversioned nonce before the next increment of the session's version

```
1  requires versionPerm > 0
2  requires acc(receipt(value, l.Version()), versionPerm)
3  requires Mem(value)
4  ensures  acc(guard(l.Version()), versionPerm)
5  func (l* LabeledLibrary) DeleteSafely(value []byte,
6      ghost versionPerm perm) {
7      // Overwrite the value with zeros
8      for i := range value {
9          value[i] = 0
10     }
11     // Prevent the compiler from optimizing the entire function
12     runtime.KeepAlive(value)
13     return
14 }
```

Figure 3.3: Implementation and specification of `DeleteSafely`, used to delete a versioned value. Preconditions, postconditions and arguments that are not relevant to the deletion mechanism have been omitted.

(but we will see in section 3.3.4 that another way is possible). Either way, this will just result in additional proof obligations without any benefits for proving security properties because the nonce's secrecy label is still the same, i.e. unversioned.

Finally, the postcondition (lines 9-10) states that the developer will receive the same (strictly positive) amount of permission of `receipt` as the amount of permission of `guard` consumed. This is indeed what was defined in section 3.2.2.

3.3.3 Secure deletion of versioned values

The library provides the function `DeleteSafely` to securely delete a versioned value. Its implementation and simplified specification are provided in Figure 3.3.

This function behaves like the opposite of the `CreateNonce` function in the *versioned* case. It takes a strictly positive `versionPerm` argument, which is used to specify the amount of permission of `receipt` to consume (line 2). The same amount of permission of `guard` is then returned (line 4). Recall that the `DeleteSafely` function does not check that the given `receipt` permission amount for this value is the full amount of permission available. But as discussed in section 3.2.2, it is in the developer's interest to specify the full amount if they want verification to succeed, i.e. eventually being able to increment the session's version. Additionally, `DeleteSafely` consumes the memory predicate of the deleted value (line 3), expressing that the value is no longer stored in memory.

To securely erase a value from memory, we use the implementation shown in lines 7-12 of Figure 3.3. This ensures that the memory is zeroed out and that this operation is not optimized away by the compiler. Line 12 currently seems sufficient to prevent the Go compiler from eliminating the function's

body. Various alternatives have been surveyed by Yang et al. [17] for the C language.

3.3.4 Creating values with multiple versions for ratcheting

When presenting the Diffie-Hellman Ratchet [13] in section 3.1.1, we explained the purpose of the ratcheting step, which is to derive a new communication key from the previous one. Once a new communication key has been derived, the previous one is no longer needed and should be deleted. Intuitively, the new key should exist in a more recent time frame than the previous one, and both keys should coexist for a short period of time, from the ratcheting step to the deletion of the previous key.

In terms of versions, this means that if the previous key K_n is versioned with the current version v, it is not sufficient to create the new key K_{n+1} only with version $v+1$ because it could not coexist with K_n. Additionally, we have seen in section 3.3.2 that creating a versioned key requires it to be readable at least in the current context. This means that when creating K_{n+1}, we would have to prove that it is readable in the current version v. To make K_{n+1} readable both in version v and $v+1$, we could for example use the secrecy label $[(p,s,v),(p,s,v+1)]$, where p and s are the participant and session of the library owner. This label now allows K_n and K_{n+1} to coexist in version v. Then, after the ratcheting step, the developer would want to increment the session's version to $v+1$. The methodology enforces that K_n is deleted before the version increment.

However, while this label makes K_{n+1} theoretically readable in version $v+1$, it does not yet align with our deletion methodology. Because K_{n+1} is created while the current version is v, a permission fraction of `guard(v)` (and not `guard(v+1)`) will be consumed[2]. This fraction can currently be restored only by deleting K_{n+1} *before* incrementing the session's version to $v+1$, which would defeat the purpose of the ratcheting step.

Conversion function

To solve this problem, we introduce a generic way to migrate values from one version to the next one if this is permitted by their secrecy labels. Intuitively, a versioned value `key` created in version v has consumed some permission of `guard(v)` and returned some permission of `receipt(key, v)`. If `key`'s secrecy label also flows to the next version $v+1$, we want to allow `key` to continue to exist in version $v+1$. To do so, we offer the developer to *convert* `key`'s receipt to a new `receipt(key, v+1)`. At the same time, we return the consumed `guard(v)` permission but consume an equivalent amount of `guardNext(v+1)`

[2]For the sake of clarity, note that in this and subsequent paragraphs, v and `v` denote the same version value.

```
1  requires l.Owner().IsSession()
2  requires versionPerm > 0
3  requires acc(receipt(value, l.Version()), versionPerm)
4  requires acc(guardNext(l.Version() + 1), versionPerm)
5  requires CanFlow(l.Snapshot(), GetLabel(valueT),
6      Readers(set[p.Id]{l.OwnerWithNextVersion()}))
7  ensures  acc(guard(l.Version()), versionPerm)
8  ensures  acc(receipt(value, l.Version() + 1), versionPerm)
9  func (l* LabeledLibrary) ConvertToNextVersion(value []byte,
10     valueT Term, versionPerm perm)
```

Figure 3.4: Specification of the abstract `ConvertToNextVersion` function, converting consumed and emitted `guard` and `receipt` permission fractions to the next version when the secrecy label of the value allows it. Preconditions and postconditions that are not relevant to the conversion mechanism have been omitted.

permission, to make the permission amounts behave *as if* the developer had created `key` directly in version $v+1$. Note that we use a different predicate `guardNext(v+1)`, playing the same role as `guard(v+1)`, for reasons that will be explained in the next section. Ultimately, this means that before incrementing the session's version to $v+2$, the deletion methodology will either force the developer to safely delete `key` or convert it once more if permitted by its secrecy label.

From this intuition, we can define the `ConvertToNextVersion` function shown in Figure 3.4. Similarly to `CreateNonce`, the library owner must be a session identifier (line 1) and not a participant. A positive fraction of the existing `receipt` permission and the next version's `guard` permission is consumed (lines 2-4). Finally, the secrecy label of the value must flow to the next version (lines 5-6). In return, `ConvertToNextVersion` returns the same amount of permission of the current version's `guard` (line 7) that was consumed when creating the value. It also returns a `receipt` for the next version (line 8).

The `ConvertToNextVersion` function solves our ratcheting problem. While in a protocol session with version v, we can obtain a new key K_{n+1} with secrecy label $[(p,s,v),(p,s,v+1)]$, where p and s are the participant and session of the library owner, from a key K_n versioned with version v. At this point, the two keys coexist in version v. The developer can then convert K_{n+1} to the next version, meaning that it replaces the obligation to delete K_{n+1} in version v with an obligation to delete K_{n+1} in version $v+1$. After this conversion, the developer can delete the old key K_n and has now obtained full permission to the `guard(v)` predicate. They can now increment the session's version to $v+1$.

Handling mutliple guards

While the conversion function solves the initial ratcheting problem, it introduces a new challenge: we now have to deal with two `guard` predicates at the same time. One, `guard(v)`, for the current version and one, `guardNext(v+1)`

for the next version. Otherwise, the conversion function would not be able to consume a fractional amount of the next version's `guardNext` predicate. Because a session's version is initialized at 0, it implies giving the developer full permission to both `guard(0)` and `guardNext(1)` predicates at the beginning of a protocol session.

Some parts of the implementation of our deletion mechanism rely on the existence of a single `guard(v)` predicate, where v is necessarily the current version of the session. In particular, some helper functions that play a role in the deletion mechanism do not have access to the session state, so do not know the current version. In these cases, `guard(v)` can be used to bind the current version `v` to the function's argument.

To prevent developers from accidentally calling these helper functions with the next version instead of the current one, we introduce a distinct predicate:

```
1 pred guardNext(v uint32)
```

The `guardNext` predicate is another abstract predicate, like `guard`, and is used in our methodology for the same purpose as `guard`, but for the next version. Using this simple adaptation, we can rely on the fact that at any time, the existing `guard` predicate has the current version as argument. This is expressed in the following lemma:

Lemma 3.3

$$\forall v_1, v_2 \in \mathbb{N}, \mathit{perm}(\mathit{guard}(v_1)) > 0 \land \mathit{perm}(\mathit{guard}(v_2)) > 0 \Rightarrow v_1 = v_2$$

3.3.5 Encryption and decryption of versioned values

In section 3.3.2, we categorized functions that create values as the ones returning full permission to a new memory predicate of the created value. Functions that fit into this category, which we have not mentioned yet, are *encryption* and *decryption* functions. Indeed, encryption functions take (and return) read permission to a plaintext, encrypt it to create a ciphertext, and return full permission on a new memory predicate for the ciphertext. Invertly, decryption functions take (and return) read permission to a ciphertext, decrypt it to create a plaintext, and return full permission on a new memory predicate for the plaintext. Both functions create a new value that is stored in memory (a ciphertext and a plaintext respectively), we should therefore make sure they are compatible with our deletion methodology.

Encryption

Encryption is the simpler case. By definition of secrecy labels, an encryption function creates a *public* ciphertext, which is unversioned. Therefore, we do not need to change the encryption function to support versioned values.

Decryption

Since a decryption function creates a plaintext, whose value is the same as the original plaintext used for encryption, the returned plaintext might contain versioned values.

As a simple example, suppose that a participant creates a versioned value *key*, and then encrypts and decrypts it. They will obtain a copy $key_{copy} = key$ of the initial versioned value, with its own memory predicate. While the current deletion methodology forces the developer to delete (or convert) the versioned value *key* before the next version increment, we need to adapt the methodology to decryption functions to also enforce the deletion of key_{copy}.

Intuitively, we want to treat the decryption function like a creation function and consume some permission of `guard` only when the created plaintext is versioned. However, we can not refer to the plaintext or its secrecy label in the preconditions of the decryption function to distinguish between the case where the plaintext is versioned, in which we require a fraction of the `guard` predicate, and the case where the plaintext is unversioned.

A possible solution to this problem would be to always require partial permission of `guard` when calling the decryption function. Then, this permission could be returned later when the developer manages to prove that the decrypted plaintext is unversioned. However, this is not ideal because it would always require the developer to use the `guard` predicate to decrypt, even when verifying a protocol that does not use versioned values at all and should thus not be affected by the deletion mechanism.

Instead, we take advantage of an existing encryption property implemented in all encryption functions of the library: everyone who can read the key used for decryption can possibly obtain the plaintext and must thus be able to read it. In terms of secrecy labels, this means that the secrecy label of the plaintext must flow to the secrecy label of the used secret key[3]. In particular, this means that a versioned plaintext must be encrypted with a versioned key. Upon decryption, while we do not know the secrecy label of the plaintext to be created, we know the secrecy label of the key used for decryption. We thus over-approximate and treat the plaintext as versioned if it was encrypted using a versioned key. This is a sound overapproximation that should not burden the developer in practice.

Proving that the decryption key is versioned involves proving that its secrecy label *does not* flow to the library owner. We have already seen when creating a versioned nonce section 3.3.2 that such a negation is not provable in general. Instead, we adopt the same workaround and require the developer

[3]In asymmetric encryption, while we encrypt with the public key, we consider the label of the private (decryption) key for this property.

```
1  requires versionPerm >= 0
2  requires versionPerm == 0 ==>
3      CanFlow(l.Snapshot(), GetLabel(keyT),
4          Readers(set[p.Id]{l.Owner()}))
5  requires versionPerm > 0 ==>
6      acc(guard(l.Version()), versionPerm) &&
7      l.Owner().IsSession()
8  ensures  err == nil ==> Mem(res)
9  ensures  err == nil && versionPerm > 0 ==>
10     acc(receipt(res, l.Version()), versionPerm)
11 func (l *LabeledLibrary) AeadDec(ghost keyT tm.Term,
12     ghost versionPerm perm) (res []byte, err error) {
13     // ...
14 }
```

Figure 3.5: Specification of `AeadDec`, showcasing the changes to support versioned values in AEAD decryption. Preconditions, postconditions and arguments that are not relevant to the changes have been omitted.

to specify whether the decryption key is versioned or not, using a `versionPerm` argument. The simplified specification of the AEAD decryption function `AeadDec` is provided in Figure 3.5. Like in the `CreateNonce` function, the `versionPerm` argument must be non-negative (line 1) and is used to specify whether the decryption key is versioned or not.

If the developer chooses 0 for the `versionPerm` argument, the decryption key is meant to be unversioned, which the developer has to prove on lines 2-4. In this case, this is the only modification to the original implementation of the `AeadDec` function. Because the decryption key is unversioned, the resulting plaintext is necessarily also unversioned, which justifies that the deletion methodology does not apply in this case.

If the developer chooses a strictly positive `versionPerm` value, the decryption key is meant to be versioned. As explained before, we do not create a proof obligation that the decryption key is versioned. As mentioned in section 3.2.2, this over-approximation does not affect soundness but only forces an implementation to safely delete a plaintext in case it would not be necessary. We then consume a partial permission of `guard` (line 6) and return the same amount of permission of `receipt` (lines 9-10). Additionally, the library owner must be a session identifier (line 7) for versions to make sense. Then, if decryption succeeds, the resulting plaintext is returned and a memory predicate is issued (line 8), expressing that it is now stored in memory.

Finally, let's reason about the case where the encryption key is versioned, but not the encrypted content. When it happens, the developer is forced to hand a strictly positive permission fraction of `guard`. They will receive the same amount of permission of `receipt` for the plaintext in return as if it was versioned. Since the plaintext is unversioned and exists in the current version, it means that its secrecy label allows the current session or participant to

```
1  requires versionPerm > 0
2  requires acc(receipt(value, l.Version()), versionPerm)
3  requires CanFlow(l.Snapshot(), GetLabel(valueT),
4      Readers(set[p.Id]{l.Owner()}))
5  ensures  acc(lib.guard(l.Version()), versionPerm)
6  func (l* LabeledLibrary) GuardFromUnversionedReceipt(value []byte,
7      valueT Term, versionPerm perm)
8  }
```

Figure 3.6: Specification of the abstract `GuardFromUnversionedReceipt` function, removing an unversioned value from the deletion mechanism. Preconditions and postconditions that are not relevant to the deletion mechanism have been omitted.

```
1  requires acc(guard(l.Version()), 1/1)
2  requires nextPerm >= 0
3  requires acc(guardNext(l.Version() + 1), nextPerm)
4  ensures  l.Version() == old(l.Version()) + 1
5  ensures  acc(guard(l.Version()), nextPerm)
6  ensures  acc(guardNext(l.Version() + 1), 1/1)
7  func (l* LabeledLibrary) BumpVersion(nextPerm perm) {
8      exhale acc(guard(l.Version()), 1/1)
9      exhale acc(guardNext(l.Version() + 1), nextPerm)
10     l.manager.version = l.Version() + 1 // Increment the version
11     inhale acc(guard(l.Version()), nextPerm)
12     inhale acc(guardNext(l.Version() + 1), 1/1)
13 }
```

Figure 3.7: Implementation and specification of `BumpVersion`, incrementing the session's version. In the function's body, `exhale` is used to consume permission and `inhale` to emit permission. Note that after the session's version is incremented line 10, a call to `l.Version()` returns the incremented version, not the old one. Preconditions, postconditions and statements that are not relevant to the deletion mechanism have been omitted. In particular, incrementing the session's version requires permission to modify the library, which does not appear in this simplified specification.

access it. This means that it could be converted to the next version infinitely with the `ConvertToNextVersion` function, which would allow the value to keep existing in memory despite version increments. Although it works, converting this value for each version would be a verification burden in practice. This is why we offer an additional library function, provided in Figure 3.6, which checks if a value associated with a receipt is unversioned (lines 3-4), and if so returns the consumed `guard` permission (line 5). This conceptually removes the corresponding values from our deletion tracking mechanism.

3.3.6 Increasing the version of a session

After having seen all cases in which a versioned value can be created, for one or several versions, and deleted, we explain how to increment the version of a session. This is done using the `BumpVersion` function, whose simplified implementation is provided in Figure 3.7.

As explained in the core idea of the methodology (section 3.2.2), the developer

must have deleted all v-versioned values before incrementing the session's version v to $v + 1$. This means that the developer must have obtained the full permission of the `guard(v)` predicate, which corresponds to the precondition on line 1. Indeed, by lemma 3.1, full permission to `guard(v)` implies that all v-versioned values have been deleted.

Additionally, we have seen in section 3.3.4 that, if allowed by their secrecy label, v-versioned values can be converted to version $v + 1$, which consumes a partial permission of the `guardNext(v+1)` predicate. This means that before incrementing the session's version, the developer has some amount of permission of `guardNext(v+1)` that may vary between 0 and 1. Thus, the `BumpVersion` function takes a `nextPerm` argument, and turns `nextPerm`-many permissions of `guardNext(v+1)` into `nextPerm`-many permissions of `guard(v+1)` (see the preconditions lines 2-3 and the postcondition line 5). This happens because the `BumpVersion` function increments the session's version (done on line 10, ensured on line 4). Now that the session's version is $v + 1$, the methodology explained in section 3.3.4 requires us to use the predicate `guard(v+1)` instead of `guardNext(v+1)`. Finally, the `BumpVersion` function returns full permission to the `guardNext(v+2)` predicate (postcondition line 6).

After incrementing the session's version to $v + 1$, we are in a similar situation as in version v but with all predicates shifted by one version. We have partial permission to `guard(v+1)`, and we still may have some receipts from values converted before the version increment. As those receipts are for version $v + 1$, we could delete them to obtain full permission to `guard(v+1)`. In parallel, permissions to `guard(v+1)` and `guardNext(v+2)` allow us to create and convert new values, exactly like before.

Note that similarly to section 3.2.2, it is in the developer's interest to specify in `nextPerm` the full amount of permission of `guardNext(v+1)` that they have. If they provide a lower amount, the function call to `BumpVersion` will still succeed, but they will not be able to retrieve the full `guard(v+1)` permission afterward, which will prevent them from incrementing the session's version to $v + 2$.

Summary

In this section, we have seen how we adapted the Reusable Verification Library to support versioned values, and how the implemented mechanism enforces the deletion of those values at the end of their time frame. While the end result is functional and sound, we have come across various challenges that we had to solve by complexifying the methodology a bit. In the next section, we will give a glimpse into an alternative approach to implementing the deletion mechanism more simply, in case obligations are supported by the targeted program verifier.

3.4 Alternative design using obligations

An obligation [4], defined for a thread, specifies an action to be eventually performed by this thread. An obligation is initially *emitted*, and once the action is performed, the obligation is *discharged*.

In this section, we first present how we could use obligations to implement the deletion mechanism in a simpler way. Then, as Gobra does not support obligations, we present a way to add them to the language to be able to implement the deletion mechanism. This section will not be as exhaustive as the description of our deletion mechanism and only aims at giving an intuition of how obligations could be used to implement it.

Idea

With a verifier supporting obligations, we could, upon creation of a time-sensitive value, emit an obligation to delete it before the next version increment. In practice, the obligation would be a construct linking the versioned value to its version number. The obligation can then be discharged upon calling the secure deletion function on the value. Then, when incrementing the protocol's session version, we check that all existing obligations are valid: meaning that the version defined in the obligation is greater than or equal to the new version number. This means that "old" obligations, with the previous version number, must have been discharged before the version increment.

The obligation mechanism would therefore be composed of two main parts: an obligation construct, and a way to check that all existing obligations satisfy certain properties.

Adding obligations to Gobra

In this section, we present a way to add obligations to Gobra to solve the particular problem of implementing our deletion mechanism. To add obligations to Gobra, we first need to define the obligation construct. In our case, it should be parameterized by the value to delete and its version number. We therefore introduce the keyword `obligation` to define an obligation in Gobra, and we use a syntax similar to the definition of an abstract predicate:

```
1 obligation Obl(key []byte, version uint32)
```

This construct could then be emitted upon creation of a versioned value, by returning it in the creation function's postcondition. Invertly, the secure deletion function would require the obligation as a precondition, and would not return it to discharge it.

The second part of the mechanism is to have an assertion capable of checking certain properties for all existing obligations. This assertion could then be evaluated upon incrementing the session's version. To do so, our main challenge is to ensure that this assertion has access to all existing obligations in its scope. Indeed, if the obligations are framed around a function call, they will not be visible to the callee. And if obligations are not returned to the caller after a function call, they will leak and will not be visible to the caller.

To avoid this, we introduce *frame checks*, to ensure that all obligations are passed to a function call, and leak checks, to ensure that all obligations are returned to the caller at the end of a function call. We can implement these frame and leak checks by using inhale-exhale expressions available on the Viper level. These expressions can be used in preconditions and postconditions and will only be evaluated either in the caller or in the callee context, which gives us sufficient expressivity. When verifying Gobra files, the Gobra code is first translated to Viper code, which is then verified. To add frame and leak checks to all Gobra functions, we could modify the translation process to automatically add such checks to all Viper functions. In our case, a frame check and a leak check can be defined in Viper with the same assertion, using the `forperm` construct to quantify over all obligations in the current context:

```
1 define check [true, forperm k: Ref, v: Int [Obl(k, v)] :: false]
```

Without going into details, this `check` assertion, when used as a precondition, behaves as a frame check and makes sure that the function caller passes all obligations in its scope to the callee. When used as a postcondition, it behaves as a leak check and makes sure that the function callee returns all obligations in its scope to the caller. Now that all obligations are always available in the current context, we could check the version of all existing obligations upon incrementing the session's version, with a similar `forperm` expression.

Therefore, using frame and leak checks, we provide a simple way to implement obligations in Gobra to ultimately implement a deletion mechanism. Note that this mechanism, given for the sake of simplicity, is not optimal yet. Indeed, it requires passing all obligations in scope, even when calling a function that does not need to read them, which can be a verification burden for the developer. The approach could be improved by allowing framing obligations around a function call in the case that the callee and all transitively called callees do not increment the version.

Chapter 4

Case study

Now that we have presented our methodology and its implementation in the Reusable Verification Library [1], we apply it to a protocol to showcase its use. We demonstrate that our implemented deletion mechanism is expressive enough to support a protocol performing key ratcheting.

We start by explaining how we adapted the WireGuard [8] protocol to perform Signal-like key ratcheting. Then, we present how we implemented and verified this protocol using our methodology, and the various challenges we faced during this process. Finally, we detail the remaining work to achieve the verification of our protocol's security properties.

4.1 Choosing a protocol to verify

First, we decide on a protocol to verify. We build on the WireGuard [8] protocol to make the verification efforts manageable, as it has already been implemented and verified with the Reusable Verification Library. Additionally, we use the Signal [9] protocol as inspiration, as it uses key ratcheting to achieve forward secrecy and post-compromise security, while WireGuard does not.

In this section, we start by presenting the Diffie-Hellman ratchet [13], a protocol that is a core component of Signal's ratcheting process. We continue by explaining how we use WireGuard's handshake to obtain the prior knowledge required by the Diffie-Hellman ratchet. Finally, we present our final choice of protocol, a modified version of the Diffie-Hellman ratchet, and explain why we made this choice.

4.1.1 Ratcheting protocols

We have already mentioned the Signal protocol in the previous chapters, which is based on the Double Ratchet algorithm. Signal [9] is a state-of-the-art

protocol, whose verification of a Go implementation[1] has, to the best of our knowledge, never been attempted. However, we do not consider that we have sufficient time to verify such a complex protocol implementation in the scope of this thesis. Optimally, we would like to verify a protocol implementation that uses the same principles of ephemeral keys and ratcheting to achieve comparable security properties.

Such a protocol exists and has already been introduced in section 3.1.1: the Diffie-Hellman (DH) Ratchet [13]. Recall that the DH Ratchet is the core component of the Signal protocol. Participants frequently exchange their Diffie-Hellman public keys and use them to obtain a shared secret that is used to derive new ephemeral keys to encrypt their messages. The DH Ratchet aims to provide both forward secrecy and post-compromise security, which are the same security properties that Signal aims to achieve. If the DH Ratchet can provide the same guarantees as the more complex Signal protocol, it is because the DH Ratchet uses a simpler communication model, where participants alternate sending and receiving messages. In Signal, the protocol has to cope with a participant sending multiple messages in a row, as well as out-of-order and lost messages.

While the DH Ratchet now appears to be a good candidate for our case study, we have to handle the fact that it requires some prior knowledge, as it was shown in Figure 3.1 page 16. We need to provide the initiator and responder with a shared secret K_0, which will be the initial key of their key chain. Additionally, we need to provide the initiator with the public key g^{y_0} of the responder, where y_0 is the private of the responder. We cannot use the DH Ratchet without this prior knowledge, which requires us to add an initial key agreement to the protocol.

4.1.2 Initial key agreement and chosen protocol

Similarly to the Diffie-Hellman Ratchet, the Signal protocol starts with an initial key agreement before running the Double Ratchet algorithm. Signal uses the X3DH [9] key agreement protocol, which is a state-of-the-art protocol that provides forward secrecy and cryptographic deniability. Additionally, X3DH is specifically designed to work asynchronously when one user is offline. To focus on the ratcheting part of the protocol, we instead pick WireGuard's handshake protocol to perform the initial key agreement, which has already been implemented and verified with the Reusable Verification Library. In practice, to build our protocol, we start from the existing WireGuard implementation and change its message transport phase, which currently keeps using the same session key for all messages, to use key ratcheting instead such that fresh keys are used for each transport message.

[1] Note that there is no official Go implementation of Signal, but some unmaintained inofficial implementations exist.

WireGuard starts with a handshake based on a Diffie-Hellman key exchange, called the *Noise IKpsk2* [12] handshake from the Noise protocol. This handshake establishes symmetric keys between the initiator and responder, which is the first part of the prior knowledge required by the DH Ratchet.

Then, it is easy to add a communication step between the initiator and responder, in which the responder computes a Diffie-Hellman key pair and sends its public key to the initiator, authenticated with the symmetric key established by the handshake. Therefore, by combining the IKpsk2 handshake with this additional communication step, we obtain the prior knowledge required by the DH Ratchet. At this point, we have already successfully adapted the WireGuard protocol to perform Signal-like key ratcheting. However, we explain in the following paragraphs how we made an additional modification to simplify the verification process of this protocol.

Recall the original ratcheting process that we presented in Figure 3.1 (page 16); when the initiator sends a message encrypted by a key K_1 to the responder together with its new public key g^{x_1}, the responder uses this public key g^{x_1} to compute the shared secret K_1 and decrypt the message. Typically, the public key g^{x_1} is sent in the associated data of the AEAD encrypted message, so its integrity is protected, but it remains publicly readable so the responder can immediately access it to compute K_1.

In the existing methodology and its current implementation, the Reusable Verification Library, the message comes with a (ghost) *message invariant*. It is a property, part of the trace invariant, about the content of the message used for verification purposes. Upon encrypting the message to send, the initiator has to prove this message invariant, and upon receiving and decrypting the message, the responder obtains the message invariant unless corruption has occurred, in which case the message was potentially sent by the attacker. In our case, for the first message sent by the initiator on Figure 3.1, the message invariant would contain information about the public key g^{x_1}. In particular, it would specify that g^{x_1} is a Diffie-Hellman public key with base generator g and exponent x_1, where x_1 is a nonce readable by the initiator. This information can then be used for verification purposes on the responder side, notably to know that $g^{x_1 y_0}$ is a Diffie-Hellman shared secret.

However, in our case, the message invariant is obtained only *after* decrypting the message. However, we would need to know that $g^{x_1 y_0}$ is a shared secret *before* decrypting to satisfy the preconditions of the decryption function. In particular, the decryption function requires us to prove that K_1 is a valid AEAD decryption key, which in turn requires us to know that $g^{x_1 y_0}$ is a shared secret.

This is why we slightly modified the DH Ratchet protocol to make it verifiable with our existing methodology and library. We present our modified protocol in Figure 4.1, and our new first communication round (running after the

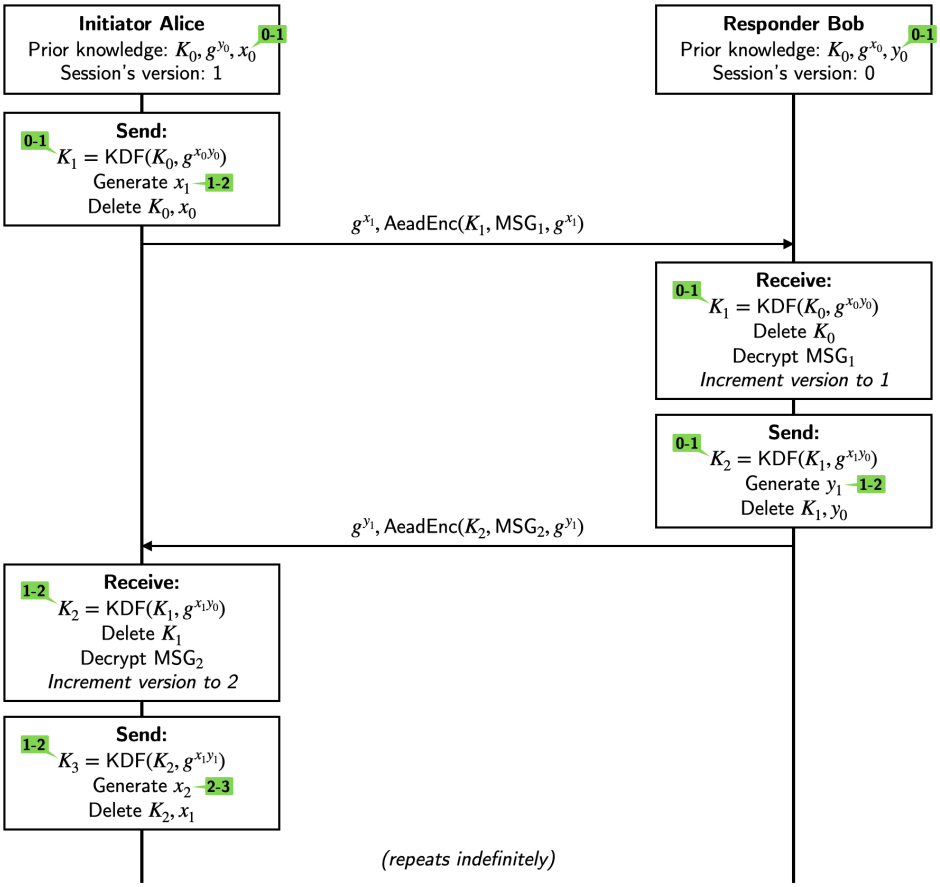

Figure 4.1: Our *ratcheting protocol*: a modified protocol based on the Diffie-Hellman Ratchet, in which the decryption key is computed using the previous DH public key instead of the just-received one. Here, `AeadEnc(K, MSG, ad)` refers to the AEAD encryption of the message `MSG` with the key `K` and the associated data `ad`. `KDF(a, b)` refers to a key derivation function that derives a new key from two inputs `a` and `b`. Green bubbles, which are linked to versioned values, indicate with their numbers the versions in which the value can be read from the participant's point of view. For example, the top-right bubble indicates that y_0 can be read in versions 0 and 1 of Bob's current session.

IKpsk2 handshake) in Figure 4.2. For the sake of readability, we now refer in the remainder of the thesis to our modified DH Ratchet protocol as the *ratcheting protocol*. Additionally, we will refer to the entire protocol that we verify, i.e. the IKpsk2 handshake followed by the first communication round and the ratcheting protocol, as the *full ratcheting protocol*.

The intuition behind the ratcheting protocol is that we replicate the DH Ratchet protocol, but instead of using the just-received public key, we use the public key received in the previous message to compute the shared secret. This first requires us to modify the prior knowledge of the participants. Now,

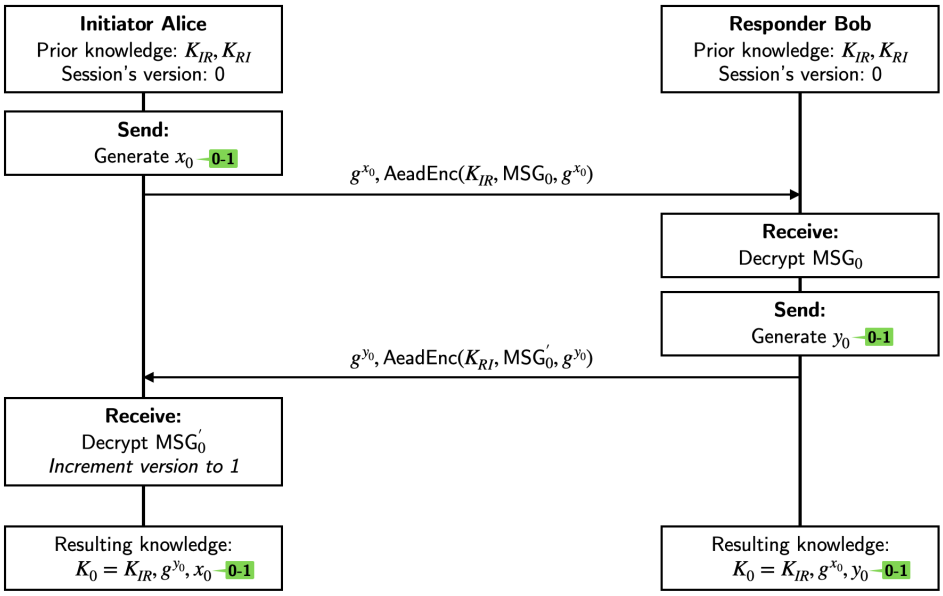

Figure 4.2: First communication round happening after the IKpsk2 handshake. It uses the shared keys K_{IR} and K_{RI} established by the handshake to authenticate the DH public keys g^{x_0} and g^{y_0} of the initiator and responder respectively. This exchange of public keys is necessary to initiate the ratcheting protocol, as it provides the required prior knowledge. `AeadEnc(K, MSG, ad)` refers to the AEAD encryption of the message `MSG` with the key `K` and the associated data `ad`. Like in Figure 4.1, green bubbles, which are linked to versioned values, indicate with their numbers the versions in which the value can be read from the participant's point of view.

in addition to the initial shared secret K_0, both participants need to know the other's DH public key and their (versioned) DH private key. This is the purpose of the new first communication round shown in Figure 4.2. The initiator first sends its public key g^{x_0} to the responder, authenticated with the symmetric key K_{IR} established by the handshake. Then, the responder replies with its authenticated public key g^{y_0}.

Coming back to the ratcheting protocol shown in Figure 4.1, when computing the first shared secret K_1, the initiator uses its private key x_0, whose associated public key g^{x_0} is already known by the responder, from the first communication round. This was not the case in the DH Ratchet protocol. The first message sent by the initiator to the responder is otherwise the same as in the DH Ratchet protocol and contains the initiator's new authenticated public key g^{x_1} in addition to the encrypted message. Upon reception of this message by the responder, instead of using the public key g^{x_1} to compute the shared secret K_1, the responder uses the previous public key g^{x_0} received in the first communication round. This allows the responder to obtain K_1, decrypt the message, and obtain the associated message invariant containing information about the new public key g^{x_1}. Then, g^{x_1} is used just after by the

responder to compute the next shared secret K_2, which is used to encrypt the next message sent to the initiator. The same observations can be made upon reception of the second message by the initiator, and all subsequent messages.

This ratcheting protocol is therefore simpler to verify than the DH Ratchet and aims to provide similar security properties. Intuitively, forward secrecy still holds because past communication keys are deleted and cryptographically not retrievable from long-term secrets and current communication keys.

Additionally, we show that post-compromise security still holds. Similarly to the original DH Ratchet protocol, we consider that the attacker compromised Alice's K_n communication key *after* her computation of the new K_{n+1} key, and after she and her peer Bob have deleted their DH private key they used to compute K_{n+1}. At this point, the attacker is unable to obtain K_{n+1} because they cannot compute the associated DH shared secret. Therefore, in this restricted compromise scenario, future communication remains secure. The protocol is therefore healed and achieves post-compromise security.

In the end, we have presented a full ratcheting protocol that draws its security properties from the frequent renewal of ephemeral keys. This protocol is mainly based on the Diffie-Hellman Ratchet protocol and has been slightly adapted to facilitate its verification, but achieves similar security properties. We will now present how we implemented and verified this protocol using our methodology and its implementation in the Reusable Verification Library.

4.2 Implementation and verification

This section explains how we implemented and verified the full ratcheting protocol presented in the previous section. We started the implementation on top of the existing WireGuard implementation, which already implements the IKpsk2 handshake. Our focus was on adapting the message transport phase to use the ratcheting protocol. In particular, the implementation work shed light on additional challenges related to the verification library. This section describes these challenges and how we solved them.

We start by explaining how we were able to model key ratcheting with our methodology. Then, we present how we used usage constraints to distinguish between different types of messages and their associated message invariants. Next, we focus on the first communication round and explain its specificities in terms of verification. Afterward, we describe how we added verification annotations to comply with the requirements of our deletion mechanism when verifying the ratcheting protocol. Finally, we explain how corruption is handled in the methodology and how we can integrate it into our verification annotations.

```
1 trusted
2 requires versionPerm > 0 && acc(guard(currentVersion), versionPerm)
3 ensures  acc(receipt(res, currentVersion), versionPerm)
4 func ComputeKDFRatchet(k []byte, dhss []byte, ghost currentVersion
5     uint32, ghost versionPerm perm @*/) (res []byte) {
6     // ...
7 }
```

Figure 4.3: Specification of the trusted `ComputeKDFRatchet` function, taking the previous key `k` and Diffie-Hellman shared secret `dhss` as input, and whose body computes the KDF result `res`. This protocol-specific KDF implementation is not part of the library but is written by the developer as part of the protocol. Preconditions and postconditions that are not relevant to the deletion mechanism have been omitted.

4.2.1 Ratcheting using a key derivation function

In section 3.3.4, we explained how we designed our methodology to support key ratcheting, namely the derivation of a new key, versioned with a later version, from the previous one. However, note that for now, we have not discussed the concrete specification and implementation of the ratcheting step. In particular, we have not discussed how we could obtain a $(v+1)$-versioned key from a v-versioned key.

In practice, this ratcheting step is implemented by a key derivation function (KDF), taking the previous key as one of its inputs and returning the new key. However, the precise KDF function used and the arguments it takes depend on the protocol. It is therefore not possible to provide a generic implementation of it in the library. This is why we have not discussed its implementation yet. For our case study, the KDF function specification is given in Figure 4.3.

`ComputeKDFRatchet` is the concrete implementation of our KDF function, and as the underlying KDF computations come from a non-verified library, we have to trust it. As with any function creating a versioned value in our methodology, it consumes some permission fraction of `guard` predicate and returns the same amount of `receipt` for the created value. This ensures the soundness of our deletion mechanism.

Now, we want to define the secrecy label of the resulting KDF. The secrecy label of a term is defined by a function `GetLabel` in the library. It returns a static label depending on the properties of the term, or a label depending on the arguments of the term when the term is a function of other terms, e.g. the hash of a term. But first, to define a secrecy label for the KDF output, we need the library to *know* about our KDF. While we defined it in the protocol, the KDF function needs to be specified on the term level in the library. We define it as follows:

```
1 func KDFRatchet(Term, Term) Term
```

We additionally specify some of its properties, like injectivity, through axioms written in the library. To then bridge the gap between the protocol and the library definitions, we add to the protocol's `ComputeKDFRatchet` function a postcondition ensuring that the KDF result corresponds to the byte representation of the term resulting from applying the newly defined KDF function on the term level.

Now that the library knows about our KDF, we can define its secrecy label. Intuitively, the only rule about the secrecy of a KDF output is that it is at most as secure as knowing all of its inputs. Indeed, if you know all the inputs of a KDF, you can compute its output. However, the KDF output is not necessarily as secure as its inputs. For example, a *hash* function behaves similarly to a KDF function and is generally used to obtain a public output from a secret input. Therefore, the secrecy label of the output of a KDF should be weaker or equal to the intersection of the secrecy labels of its inputs.

In our case, we chose to define the secrecy label of the KDF output term `newk := KDFRatchet(k, dhss)` as the secrecy label of the Diffie-Hellman shared secret `dhss` only. Indeed, considering v the current session's version, because the input key k is not $(v+1)$-versioned, it is easy to see that `newk` cannot obtain a secrecy label containing $(p, s, v+1)$ from `k` alone. Instead, we show that the `dhss` term has to be $(v+1)$-versioned, therefore making `newk` $(v+1)$-versioned as well.

The shared secret $\texttt{dhss} = g^{xy}$ is obtained by exponentiating the public key g^y of the other participant with our private key x. As x is our key, we get to define its secrecy label at creation. In our ratcheting protocol, x is meant to be an ephemeral private used to derive the two next shared communication keys K_n and K_{n+1} (the first one for receiving and the other for sending, where n is even and odd for Alice and Bob respectively), as shown in Figure 4.1. As we want to use x to obtain a shared secret for the next round of communication, we make it readable in version $v+1$. Because any value also needs to be readable in the current version, x is given the label $[(p, s, v), (p, s, v+1)]$, where p and s are the respective participant and session creating x. From the message invariant, we obtain that y is readable by the other participant p'. Because p does not need to precisely know in which session and version y is readable by p', the message invariant only gives us the simplified secrecy label $[(p')]$ for y. Then, the labeling rules defined in the existing library state that the secrecy label of the shared secret g^{xy} is the union of x and y's secrecy labels, which is $[(p, s, v), (p, s, v+1), (p')]$.

This $(v+1)$-versioned secrecy label of `dhss` is exactly what we wanted for our `newk` term. We define the labeling of our `KDFRatchet` term in the library by stating the following postcondition for the library's `GetLabel(term Term) (res SecrecyLabel)` function:

```
1 ensures forall k, dhss Term ::
2     term == KDFRatchet(k, dhss) ==> res == GetLabel(dhss)
```

Therefore, we have adapted our labeling rule and shown that following our ratcheting methodology, we can obtain a $(v+1)$-versioned key from a v-versioned key.

4.2.2 Usages and message invariants

In section 4.1.2, we discussed how message invariants are crucial to the verification of our ratcheting protocol. Recall that they are a property proven by the initiator upon AEAD encryption of a message, and obtained by the responder upon AEAD decryption of the received message. It is important to note that each kind of message needs its message invariant. In our case, we have several kinds of messages: a variety of messages used in the IKpsk2 handshake, the messages used in the first communication round, and the messages used in all subsequent rounds in the ratcheting protocol. For the handshake messages, we have been able to fully reuse the existing message invariants from the WireGuard implementation, so we will not discuss them further. This leaves us with two kinds of messages to consider.

Because of these different kinds of messages and their associated message invariants, we need to be able to distinguish them in the library. To do so, the current implementation of the library, similarly to DY* [2], uses *usage constraints*. A usage is a way to annotate a term that should be used for a specific purpose. For example in the case of our ratcheting protocol, we use usages to distinguish Diffie-Hellman private keys, public keys, shared secrets, and K versioned communication keys. A usage can initially be defined upon creation of a value. This is the case when creating a Diffie-Hellman private key. But then, similarly to secrecy labels, usages of terms are obtained using a protocol-specific `GetUsage` function, which can return a usage depending on the arguments of the term when the term is a function of other terms. For example, it is in this `GetUsage` function that we define that the exponentiation of the generator with a key of usage *DH Private Key* has usage *DH Public Key*. Similarly, the exponentiation of a key of usage *DH Public Key* with a key of usage *DH Private Key* has usage *DH Shared Secret*.

Coming back to the two kinds of messages in our ratcheting protocol that we need to distinguish, we use usages to do so. Keys used in the first communication round already have a specific usage `KUsage` that they obtained at the end of the handshake. For subsequent ratcheting keys, we extend the `GetUsage` function to return a specific usage `KVersUsage` for keys resulting from the `KDFRatchet(k, dhss)` function, where `k` is a key of usage `KUsage` or `KVersUsage`, and `dhss` has a *DH Shared Secret* usage.

To be even more specific about the implementation, we found that it makes verification much easier to distinguish between messages coming from the initiator and messages coming from the responder, and to use slightly different message invariants for each. Usage constraints were the natural way to do so, by defining a usage `KVersIrUsage` for keys used to encrypt a message sent from the initiator to the responder, and a usage `KVersRiUsage` for keys used to encrypt a message sent from the responder to the initiator (and similarly defining `KIrUsage` and `KRiUsage` for the first communication round).

Ultimately, using usage constraints, we can distinguish between different kinds of messages upon AEAD encryption and decryption, and identify messages coming from the initiator or the responder. This simplifies the specification of message invariants because we can distinguish between messages and thus separately specify message invariants for kind of each message.

4.2.3 First communication round

As shown in Figure 4.2, the first communication round is a little different from the other rounds. Before it starts, the initiator and responder have shared secrets K_{IR} and K_{RI} resulting from WireGuard's handshake. The goal of this first round is to send authenticated public keys to each other so that they can start the ratcheting protocol. The ratcheting protocol also requires a shared secret K_0 as prior knowledge, which is defined in the first communication round as $K_0 = K_{IR}$.

This initial key K_0, coming from the WireGuard implementation, is not versioned so we will not be forced by the methodology to delete it. As it is only the first key, this will not have a big impact on the security properties of the protocol. However, by also adapting the handshake and thus increasing verification efforts further, we could consider implementing a versioned handshake resulting in a versioned K_0.

But because K_0 is not versioned, we do not have a `receipt` for it. This means that we cannot call the `DeleteSafely` function on it, as it requires such a receipt to consume it. This has an impact on the second communication round, more particularly on the initiator's first send and the responder's first receive of the ratcheting protocol, shown in Figure 4.1. Indeed, these functions are deleting K_0, but they do not use the particular `DeleteSafely` function that is meant to delete versioned values. As they do not, the preconditions and postconditions of these functions may carry different amounts of guard and receipt permissions compared to all subsequent rounds. This has to be kept in mind when working on verifying the protocol implementation in compliance with the requirements of the deletion mechanism. This work is explained in the next subsection.

4.2.4 Using the deletion mechanism

In this section, we explain how we practically comply with the requirements of the deletion mechanism to verify the protocol implementation. The first step in this process is to reason about the adequate times to increment the protocol session's version, at which point previous keys must be deleted. This choice will directly impact the resulting security properties of the protocol. Indeed, the more frequently we increment the session's version, the more we can reason about the temporal existence of keys with a fine granularity, which could lead to tighter security properties. Of course, there is no point in incrementing the version *too* frequently. If no data has been deleted between two versions, the last version increment is unnecessary and only results in added verification complexity without benefits.

In the case of our ratcheting protocol, we notice that with the notations of the Figure 4.1, for $n > 0$, the initiator's private key x_n is used in the computation of the communication keys K_{2n} and K_{2n+1}. We can obtain a similar result for the responder. We also notice that we delete a key K_{2n} just after we derive the next key K_{2n+1}. As a first intuition, it would seem natural to increment the session's version just after we derive K_{2n+1} so that we force the deletion of K_{2n} at this point. However, doing so is not possible. Indeed, at each round, a participant first derives a receiving key K_{2n} and then a sending key K_{2n+1}, using the `ComputeKDFRatchet` function. While they use a different Diffie-Hellman shared secret to derive each key, both shared secrets use the same private key x_n. Recalling the explanations about the labeling of the KDF result given in section 4.2.1, we see that the secrecy label of x_n defines the secrecy label of the KDF results K_{2n} and K_{2n+1} for the calling participant. Because the part of the secrecy label about the other participant p' is always the same, (p'), the secrecy label of K_{2n} and K_{2n+1} is the same. Therefore, as we give x_n the label $[(p, s, n), (p, s, n+1)]$, the secrecy label of K_{2n} and K_{2n+1} is $[(p, s, n), (p, s, n+1), (p')]$. Because of this, forcing the deletion of K_{2n} after deriving K_{2n+1} would require to increment the session's version to $n+2$, which would also force the deletion of K_{2n+1}.

Therefore, we have to use a coarser granularity for our versioning. We saw that the secrecy label of a K_{2n} communication key is fully determined by the label given to the Diffie-Hellman private key x_n used to derive it. We would then achieve the best possible granularity by incrementing the session's version between the computation of two private keys x_n and x_{n+1}. This way, supposing having x_n with secrecy label $[(p, s, n), (p, s, n+1)]$, we aim to obtain x_{n+1} with secrecy label $[(p, s, n+1), (p, s, n+2)]$. Let's increment the session's version at the end of the receive function, before calling the send function, and observe what happens.

We look at the initiator p, supposing they are in session s and version n, at the beginning of their receive function. We suppose having a private key

```
requires iter == 0 ==>
    acc(guard(v), p) && acc(guardNext(v+1), p)
requires iter == 1 ==>
    acc(guard(v), 2*p) && acc(guardNext(v+1), p) &&
    acc(receipt(privKey, v), p)
requires iter > 1  ==>
    acc(guard(v), 2*p) && acc(guardNext(v+1), p) &&
    acc(receipt(privKey, v), p) && acc(receipt(K, v), p)
ensures iter == 0 ==>
    acc(guard(v), p) && acc(receipt(newPrivKey, v+1), p)
ensures iter == 1 ==>
    acc(guard(v), 2*p) && acc(receipt(newK, v), p) &&
    acc(receipt(newPrivKey, v+1), p)
ensures iter > 1  ==>
    acc(guard(v), 3*p) && acc(receipt(newK, v), p) &&
    acc(receipt(newPrivKey, v+1), p)
func SendMessage(K []byte, privKey []byte, iter int, ghost v uint32,
    ghost p perm) (newK []byte, newPrivKey []byte) {
    // ...
}
```

Figure 4.4: Specification of the function `SendMessage` that sends a message in the ratcheting protocol. Preconditions, postconditions and arguments that are not relevant to the deletion mechanism have been omitted.

x_n with secrecy label $[(p,s,n),(p,s,n+1)]$, and a previous communication key K_{2n-1} with secrecy label $[(p,s,n-1),(p,s,n),(p')]$. As we explained, we first compute the key K_{2n} that has a secrecy label $[(p,s,n),(p,s,n+1),(p')]$. We can then delete K_{2n-1} as it is not needed anymore. Next, we decrypt the received message using K_{2n} Then we increment the session's version to $n+1$ and the receive function ends. Doing so, the methodology has *forced* us to delete K_{2n-1} because it does not flow to version $n+1$. Now, in the send function, we start by deriving the next communication key K_{2n+1}, which has a secrecy label $[(p,s,n),(p,s,n+1),(p')]$, like K_{2n}. We then compute the next private key x_{n+1}, choosing its secrecy label to be $[(p,s,n+1),(p,s,n+2)]$. Next, we delete K_{2n} and x_n, as they are not needed anymore. We finish by sending our message encrypted with K_{2n+1}, and the send function ends. In this case, we were not forced yet to delete K_{2n} and x_n, because we are still in version $n+1$. However, we will be forced to delete these two values at the end of the next receive, when we will increment the session's version to $n+2$. This granularity should be sufficient to later express our security properties.

Now that we have decided on the granularity of our versioning, we simply follow the requirements of the deletion mechanism that we described in section 3.3. These requirements mostly consist of specifying the "correct" amount of `guard` and `receipt` permissions in function calls. Additionally, recall the `ConvertToNextVersion` function introduced in section 3.3.4. The developer will have to convert values labeled with multiple versions (private keys x and communication keys K) when necessary, to keep using them in the next session's version.

To give a final intuition about the actual reasoning that the developer has

to do to comply with the deletion mechanism, we give in Figure 4.4 the preconditions and postconditions related to the `guard` and `receipt` predicates of the initiator's send function. The case `iter == 0` corresponds to the send operation in the first communication round. Here, a fraction of `guard(v)`, where v is the current version, is necessary to create the new Diffie-Hellman private key `newPrivKey`. As it is created with a label $[(p, s, v), (p, s, v + 1)]$, we then convert it to the next version $v + 1$. This consumes the fraction of `guard(v+1)` and returns both a fraction of `receipt(newPrivKey, v+1)` and a fraction of `guard(v)`. The case `iter == 1` corresponds to the second send operation of the initiator, so to the first send operation of the ratcheting protocol. Compared to previously, we additionally need another `guard(v)` fraction to compute the initial KDF, which in turn returns a `receipt(newK, v)` fraction. And we delete the previous private key, consuming the `receipt(privKey, v)` fraction and returning a `guard(v)` fraction. Finally, the case `iter > 1` is the same as before but with the additional deletion (using `DeleteSafely`) of the previous communication key, consuming the `receipt(K, v)` fraction and returning an additional `guard(v)` fraction.

4.2.5 Corruption

In the existing methodology, an attacker can corrupt a participant's memory and obtain the stored secrets. The methodology distinguishes between the corruption of a participant, leaking all of their long-term secrets, and the corruption of a particular session, leaking all of the secrets of the session in addition to the long-term secrets. With our extended methodology, we add the case of corrupting a version, leaking all of the ephemeral secrets of the version in addition to the session secrets and the long-term secrets. Corruption is a crucial notion to consider when verifying a protocol, as we need it to express strong security properties. In particular, certain security properties hold even under some cases of corruption. For example, post-compromise security states that a key remains secret even if the attacker has corrupted certain versions in the past.

In the Reusable Verification Library, corruption is modeled upon receiving and decrypting an AEAD encrypted message. Recall from section 4.1.2 that upon decryption, we obtain a message invariant containing properties about the message that were proved by the sender, unless the attacker performed the encryption. In the latter case, the attacker must have computed the encryption and accessed the encryption key and message, i.e. the secrecy labels of the encryption key and the message must flow to *Public*.

Coming back to our ratcheting protocol, we have seen in the previous section that an ephemeral communication key has a secrecy label of the form $[(p, s, v), (p, s, v + 1), (p')]$, where v is the current version of session s. If we learn that the message came from the attacker and thus the secrecy label

of the encryption key flows to *Public*, we learn that either version (p, s, v), or version $(p, s, v + 1)$, or the long-term secrets of participant p' must have been corrupted[2]. Indeed, corrupting any of these is the only way for the attacker to obtain the communication key, as proven in the secrecy lemma of the existing methodology.

In practice, this results in case distinctions while verifying our protocol implementation. Upon receiving and decrypting a message, we have to consider the case where we receive the message invariant and the case where we know that corruption occurred. In the second case, not obtaining the message invariant means that we learn nothing about the associated data of the encrypted message. In our ratcheting protocol, we do not learn that the associated data is an exponential of the form g^y, with a *DH Public Key* usage, where y is a private key readable by the other participant.

This has an impact on other verification annotations of the protocol because these case distinctions propagate to other preconditions and postconditions. For example, the receive function of the protocol implementation can return information about the received public key only when the message invariant is obtained, but not in case of corruption. And the subsequent send operation is supposed to compute a shared secret using the received public key, to then derive the next communication key. The preconditions of these functions must be satisfied in both cases, whether we obtained the message invariant or not. In practice, we make the receive function of the protocol return a ghost boolean parameter `newCorrupted` to inform the caller whether corruption must have occurred or not. We additionally introduce a ghost boolean parameter `corrupted` as input of the receive and send functions of the protocol, and we use it to express different preconditions and postconditions, and call different lemmas depending on the particular case we are in.

This work of integrating corruption in the verification annotations of the protocol implementation turned out to be a tricky and time-consuming task. In the scope of this thesis, we were not able to fully complete it. We discuss in detail in the next section the remaining work to be done to fully integrate corruption into our proof, to complete the verification of the strong security properties of our ratcheting protocol.

4.3 Remaining work

In this section, we detail how one could finish the verification of our ratcheting protocol. We list the missing parts of the verification work and we provide a clear path to complete them.

[2]Which is also the case when any session or any version of a session of p' has been corrupted.

First, we need to satisfy the AEAD encryption requirements in the corrupted case. Upon encryption, the current methodology requires us among others to prove that the encryption key has an appropriate usage. A usage is defined as "appropriate" for AEAD encryption by the developer in the `GetUsage` function. In our case, the usages `KVersIrUsage`, `KVersRiUsage`, `KVersIrUsage` and `KVersRiUsage` defined in section 4.2.2 are defined as being appropriate for AEAD encryption. However, in the corrupted case, we do not know the usage of the received associated data, which we expect to be the other participant's public key. The current usage rules prevent us from proving that the shared secret obtained from this public key and our private key has a *DH Shared Secret* usage. Hence, we cannot show that the encryption key obtained with `KDFRatchet` from the previous key and this shared secret has an appropriate usage for AEAD encryption. To solve this, we suggest changing the protocol-specific usage rules defined in `GetUsage` to make the usage of `KDFRatchet`'s output independent of the usage of its shared secret input. Thus, we would be able to prove that the encryption key has an appropriate usage for AEAD encryption even in the corrupted case.

To satisfy the AEAD encryption preconditions, we additionally have to show that there exists a label that flows to the encryption key and to which the secrecy label of the message flows. In the non-corrupted case, we use the secrecy label $[(p, s, v), (p, s, v + 1), (p')]$ of the encryption key, to which the message secrecy label $[(p, s), (p')]$ flows. In case of corruption, the secrecy label of the encryption key is harder to determine because we know nothing about the received associated data and we cannot assume that it is the other participant's public key. We have to make a disjunction on what the received associated data could be to compute the secrecy label of the encryption key in each case. By doing so, we can show that in all cases, the secrecy label of the encryption key flows to the label of the DH secret key used to compute it, which is $[(p, s, v), (p, s, v + 1)]$. As the secrecy label of the message flows to this label, we satisfy the AEAD encryption preconditions.

Overall, verifying our protocol implementation in case of corruption will require us to make several case distinctions and to call additional lemmas to complete each case's proof. While time-consuming, we are confident that it is completely doable.

Next, the last part of the verification work is to prove the security properties of our ratcheting protocol. All the verification annotations that we have introduced before, which model concepts like versions and corruption, were added to be able to express security properties. Suppose a participant p executes the ratcheting protocol in session s and version v, with a peer participant p'. We have shown that the current communication key K_n has a secrecy label of the type $[(p, s, v), (p, s, v + 1), (p')]$. K_n remains secret unless the attacker compromises the version v or $v + 1$ of the session s, or the other

participant p'.

However, our secrecy property is currently weaker than it could be, as the secrecy label of K_n contains p', which expresses that K_n can be obtained by the attacker if they compromise p' at any time. This property could be strengthened, as we can intuitively see that K_n is also available in only two versions of a session s' of p'. We leave it to future work to show that, when p is the initiator and for $n > 0$, the secrecy label of K_{2n} can be strengthened to $[(p, s, n), (p, s, n+1), (p', s', n-1), (p', s', n)]$ and the secrecy label of K_{2n+1} can be strengthened to $[(p, s, n), (p, s, n+1), (p', s', n), (p', s', n+1)]$. Showing this requires tightening the message invariant that we initially introduced in section 4.2.1.

Then, we can express that the key K_{2n+1} remains secret unless the previous key K_{2n} was made public before K_{2n+1} was computed, or unless the attacker compromises a version n or $n+1$ of the session s of participant p or of the session s' of participant p'. Indeed, if the attacker compromises K_{2n} before K_{2n+1} is computed, they could impersonate a protocol participant and eventually obtain K_{2n+1}. Similarly, we express that a key K_{2n} remains secret unless the previous key K_{2n-1} was made public before K_{2n} was computed, or unless the attacker compromises a version n or $n+1$ of the session s of participant p or a version $n-1$ or n of the session s' of participant p'. Overall, these secrecy properties express both forward secrecy and post-compromise security for our ratcheting protocol.

To sum up, we have shown the path to complete the verification of our protocol's security properties. This will require some modifications to existing annotations, including strengthening the message invariant and probably writing additional lemmas.

4.4 Results

We finish this case study by reporting some quantitative results about the verification of our full ratcheting protocol. Figure 4.5 provides the number of lines of code (LOC), the number of lines of specification (LOS) and the verification time for the implementations of the full ratcheting protocol and the original WireGuard. The number of lines of specification does not count the lines of the Reusable Verification Library, and the verification time does not include the time to verify the library. This is justified because the library is protocol-independent and can be verified once and for all.

Our full ratcheting protocol is composed of 759 lines of code and 6167 lines of specification, which is around 8.1 times more specification than code. However, note that the verification of the full ratcheting protocol is not complete yet and we thus expect a slightly higher ratio of specification to code once it is. In comparison, the original WireGuard implementation is

Protocol	LOC	LOS	Verification time [s]
Full ratcheting protocol	759	6167	669.5
WireGuard	550	5411	441.8

Figure 4.5: Lines of code (LOC), lines of specification (LOS) and verification time for the implementations in Gobra of the full ratcheting protocol and the original WireGuard. The numbers for the latter are provided for reference because the full ratcheting protocol builds on top of the original WireGuard protocol. We have measured the verification time by averaging over 3 runs on a 2020 Apple MacBook Pro with a 4-core Intel Core i5 processor and 16 GB of RAM.

composed of around 9.8 times more specification than code, but corruption has been fully integrated into its specification and its security properties have been verified.

The verification time of our full ratcheting protocol is 669.5 seconds, which is around 1.5 times more than the verification time of the original WireGuard implementation. This can be explained in part because our full ratcheting protocol contains additional specifications related to our deletion mechanism, which is not present in the specification of the original WireGuard.

Chapter 5

Conclusion

In conclusion, this thesis has successfully extended the methodology developed by Arquint et al. [1] for the modular verification of security protocol implementations, adding support for the specification of time-sensitive data and enforcing their secure deletion at the end of their lifetime. To the best of our knowledge, our contributions enable for the first time the verification of strong security properties relying on the secure deletion of sensitive data for real-world protocol implementations. By building upon standard separation logic predicates, the extended methodology remains agnostic to programming languages.

The generic extension we introduced is flexible enough to verify various protocol implementations that use ephemeral data, such as Signal [9] and its key rotation scheme. Our methodology can be used to prove strong security properties like forward secrecy and post-compromise security.

Furthermore, we have extended the Reusable Verification Library [1] for Go to contain our methodology. Using a solution inspired by counting permissions [15], we provide the library with the ability to track sensitive values in a protocol implementation and ensure that they are deleted in a timely manner. By implementing this solution in several functions of the library, we facilitate the work of the developer by ensuring in a sound way that the values they create can only exist during the periods of time specified for these values.

We evaluate our methodology with a case study on a Signal-like protocol implementation. This case study shows that our extended methodology is expressive enough to reason about complex mechanisms such as key ratcheting. While the full security proof could not be completed for time reasons, we provide a clear path toward its completion in this thesis.

In the next and final section, we propose some ideas for future work that could improve or extend this thesis.

5.1 Future work

Finishing the case study. The case study presented in chapter 4 could be completed by following the path we have outlined in section 4.3, resulting in the verification of forward secrecy and post-compromise security for this protocol implementation.

Adding obligation support to Gobra. As discussed in detail in section 3.4, adding obligations [4] to Gobra [16] would allow us to implement a simpler and more elegant solution for tracking sensitive values. This may result in a more intuitive methodology and simpler specification for library functions that developers can use more easily.

Improving how the methodology handles AEAD. When choosing the protocol to verify for our case study, in section 4.1.2, we had to slightly adapt the original Diffie-Hellman ratchet protocol to make it more easily verifiable. The reason for this comes from how the current methodology handles AEAD decryption, which hinders our ability to decrypt the message using a key that is derived from the associated data we just received. In future work, we could analyze how ratcheting protocols use associated data before computing the necessary decryption key. In particular, we should reconsider the preconditions and postconditions of the AEAD decryption function such that we can invoke this function in these situations and soundly obtain the associated message invariant unless the ciphertext has been created by the attacker.

Bibliography

[1] Linard Arquint, Malte Schwerhoff, Vaibhav Mehta, and Peter 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*, CCS '23, New York, NY, USA, 2023. Association for Computing Machinery.

[2] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. DY*: A modular symbolic verification framework for executable cryptographic protocol code. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 523–542. IEEE, 2021.

[3] Bruno Blanchet et al. Modeling and verifying security protocols with the applied pi calculus and proverif. *Foundations and Trends® in Privacy and Security*, 1(1-2):1–135, 2016.

[4] Pontus Boström and Peter Müller. Modular verification of finite blocking in non-terminating programs. Technical report, ETH Zurich, 2014.

[5] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. On post-compromise security. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 164–178, 2016.

[6] Whitfield Diffie and Martin E Hellman. New directions in cryptography. In *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*, pages 365–390. 2022.

[7] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[8] Jason A Donenfeld. Wireguard: next generation kernel network tunnel. In *NDSS*, pages 1–12, 2017.

[9] Moxie Marlinspike and Trevor Perrin. The x3dh key agreement protocol. *Open Whisper Systems*, 283:10, 2016.

[10] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*, pages 696–701. Springer, 2013.

[11] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings 17*, pages 41–62. Springer, 2016.

[12] Trevor Perrin. The noise protocol framework. *PowerPoint Presentation*, 2018.

[13] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm. *GitHub wiki*, page 10, 2016.

[14] Phillip Rogaway. Authenticated-encryption with associated-data. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, page 98–107, New York, NY, USA, 2002. Association for Computing Machinery.

[15] Matthias Roshardt. *Extending the Viper Verification Language with User-Defined Permission Models*. PhD thesis, Master's thesis, 2021.

[16] Felix A Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C Pereira, and Peter Müller. Gobra: Modular specification and verification of go programs. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I 33*, pages 367–379. Springer, 2021.

[17] Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. Dead store elimination (still) considered harmful. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1025–1040, 2017.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

SECURE DELETION OF SENSITIVE DATA IN PROTOCOL IMPLEMENTATIONS

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
QUEINNEC	HUGO

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

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

Place, date	**Signature(s)**
ZURICH, 24.09.2023	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.