Secure Deletion of Sensitive Data in Protocol Implementations

Master's Thesis Project Description

Hugo Queinnec

Supervisors: Linard Arquint, Prof. Dr. Peter Müller
Department of Computer Science, ETH Zurich

Start: 12th March 2023
End: 12th September 2023

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 that an attacker might have obtained in the past and tries to send them again. 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.

Guaranteeing security properties of protocols is a challenging task, especially in the presence of a strong attacker (developed in section 2.1). 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.

When reasoning about correctness, 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 [1] and ProVerif [2]). 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. [3] propose a promising methodology for verifying security properties for protocol implementations, they cannot reason about sensitive data that must be deleted in a timely manner. This is a limitation for proving advanced security properties, such as forward secrecy and post-compromise security, for protocols that periodically renew keys, like Signal [4]. 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 leaked *in the past*. This is sometimes referred to as *healing*, which allows a 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 session, where a communication key is used to generate its successor and is then deleted. The goal of this thesis is to extend the methodology by Arquint et al. to prove that time-sensitive data, such as this communication key in the case of Signal, has been securely deleted. This allows us to prove forward secrecy and post-compromise security properties for protocol implementations that employ a key rotation scheme.

2 Background

2.1 Symbolic Protocol Analysis

This research work is a continuation of the study of symbolic methods for protocol analysis. 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. We assume that all operations are performed on symbolic *terms* instead of bytes, which is why we abstract concrete bit-strings to terms.

Furthermore, we consider a Dolev-Yao [5] 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 the participant's state).

A common procedure for modeling protocols in this setting is to consider a trace. This 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 and ProVerif can verify certain security properties for a protocol *model* by analyzing all its possible traces.

When considering the verification of protocol *implementations*, traces can also be used. The overall idea is to extend the implementation with a trace data structure that is used to record protocol operations. In order to reason about the whole protocol (and not a single execution), we define a *trace invariant.* It is a property that every prefix of any trace of the protocol must satisfy. Verifying the protocol implementation ultimately consists in proving that each action of a participant or the attacker maintains the trace invariant, and then proving that the invariant implies the intended security properties.

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

2.2 DY*

DY* [6] 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.

DY* uses a *global trace* to model the global execution state of the protocol. 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 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, as explained in detail in section 3.2.

Each participant's state comprises several values that together constitute the program state of the participant at that point in time. Each of these values is assigned a secrecy label, indicating which participants are allowed to read the value.

Protocol implementations written in F* and verified with DY* are executable, but require a special runtime environment that provides access to the trace. 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.

2.3 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. [3] 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 [7] verifier.

This methodology is inspired by DY* and also uses a global trace to provide a global view over the entire system. To address the issue of arbitrary code structure in existing implementations, the trace is treated as a concurrent data structure. It allows arbitrary interleavings of operations, in a more fine-grained manner than arbitrary interleavings of protocol steps in DY*, which is crucial for soundness in this setting. As existing implementations manage 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 program state stored at the local participant-level with the global trace invariant. 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.

Verification is based on separation-logic, which allows us to reason about heap manipulations. Furthermore, separation-logic's resources are used to prove injective agreement, which is to the best of our knowledge not possible with DY*.

Similarly to DY*, this methodology comes with a library that allows to reuse protocol-independent parts (verified only once) across different protocol implementations.

3 Secure deletion of data

3.1 Security properties

As mentioned, some protocols achieve strong security properties by frequently renewing their keys within the same protocol session. This is notably the case in the Double-Ratchet Protocol [8], which is a major component of the Signal protocol.

A session of the Double-Ratchet protocol requires frequent renewal of a shared secret between two participants using the Diffie-Hellman ratchet. Then for every message, a *communication key* is derived (from the current shared secret, using a key derivation function) and used to encrypt the message. The Double-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, he 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 he 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(s), and not all previous ones.

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

Post-compromise security, as briefly introduced earlier, is not achievable after the *unrestricted* compromise of a participant: the initiator would have no guarantee to be in communication with the intended peer because the attacker could act exactly like the peer. We instead consider the formalized notion of post-compromise *via state* [9]: a participant can communicate securely with a peer during a session even when long-term keys have been leaked as long as there is some secret data available exclusively to the participants. 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 the future communication depends. Although post-compromise security is not directly related to secure deletion, it requires a fine-grained reasoning about the data occurring in a participant's state at a particular point in time. As we will see in section 3.3, the same mechanism as for forward secrecy can be applied.

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

3.2 Existing approach and its shortcomings

DY* uses a *version* label for each session, in order to indicate temporality. Initially, all versions are 0 and are incremented at some times to represent new protocol phases. Values with version x can only occur in states within phase x of the protocol. DY* enforces this restriction with a suitable invariant over states. Thus, only data of the current version can be stored, ensuring that neither outdated keys nor future keys are present in memory. Building on this, they are able to prove forward secrecy and post-compromise security for protocols like Signal.

However, DY* does not enforce that outdated keys are securely deleted from memory. They only show that they are not present in the current scope, but outdated keys are not explicitly zeroed out from memory. Moreover, this solution is not applicable to existing implementations because their state is not stored on a trace, making it impossible to express such an invariant. 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.

However, some existing implementations may contain long-running or non-terminating methods. It would not be sufficient to enforce the invariant only at the beginning and at the end of these methods, as this would result in an unrealistic attacker model where the attacker is assumed to not have access to the program state most of the time. Therefore, we want to have an invariant that always holds over the local state instead of only at certain times. This invariant must be designed to account for the situation during key generation where keys of both current and next versions may coexist in the state for a *short period*.

3.3 Our approach

This thesis extends the methodology of Arquint et al. by introducing a notion of temporality and by enforcing secure deletion of sensitive data. We aim for a language-agnostic solution that can be applied to existing implementations performing key rotations, such as implementations of Signal and the Double-Ratchet Protocol. This extension will allow us to prove forward secrecy and post-compromise security for these protocol implementations.

Our approach will use a mechanism similar to that of DY* *versions* to indicate the lapses of time during which sensitive data can be present in memory. However, we have to develop a modular verification technique to ensure that sensitive data is guaranteed to be absent in all other lapses of time.

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 in figure 1 only for illustration purposes, because the predicate is in fact abstract, 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.

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

Figure 1: Illustrative body of the current memory predicate in the Go reusable verification library.

For this thesis, we add a notion of temporality via a separate memory predicate. Thinking about how to design such a predicate will be a first challenge of this thesis. One possible approach is a new memory predicate that marks byte arrays that require secure deletion at the end of their lifetime and has an additional parameter specifying the byte array's *version* (i.e. lifetime).

However, we have to ensure that variables are actually deleted from memory as specified by the memory predicate. If we use *version*, it would mean to ensure that variables with a version `v` are deleted from memory before the protocol transitions to version `v+1`.

We can treat this new memory predicate not only as a predicate abstracting over memory but also as an obligation to securely delete the associated value from memory. To do this, we can enforce, when the protocol version changes, that all instances of this new memory predicate have an appropriate version number. Additionally, *leak checks* can be implemented to ensure that all predicate instances are returned after function calls. This enforces that byte arrays with a version have to be deleted with a dedicated secure deletion function before the protocol version is incremented. This delete function would be the only way to discharge obligations resulting from the new memory predicate.

Gobra does not support leak checks yet but Viper [10], on which Gobra builds, offers `forperm` expressions that allow us to encode them. In order to achieve our final goal of supporting properties like forward secrecy in the reusable verification library, we will have to extend the library with this new memory predicate, a library method that we assume securely deletes memory, and leak checks. For the latter, we have to extend Gobra to support obligations, building on Viper's `forperm` expressions.

4 Core Goals

- **Methodology.** Present a methodology that extends Arquint's and al. work on protocol implementation verification, and that would allow to verify the deletion of old data in a protocol implementation. This methodology would ultimately allow the verification of high-level security properties relying on data deletion, such as forward secrecy.
- **Extension of Gobra.** Extend the Gobra verifier to support the specific types of obligations required for this project, based on existing Viper primitives.
- **Extension of the Go Reusable Verification Library.** Extend the reusable verification library to support the deletion of old data, based on the new functionalities added to the Gobra verifier at the previous step. The library will then provide higher-level APIs to handle the deletion mechanism, that will be useful for verifying protocol implementations.
- **Implementation of an example protocol.** Using the previously extended verification library, implement and verify a small protocol relying on frequent deletion of sensitive data. Verifying a high-level property such as forward secrecy will showcase the potential of the methodology.
- **Evaluation.** Evaluate the newly developed methodology and its application to the example protocol, both qualitatively and quantitatively, by evaluating the performance of the new Gobra functionality for checking obligations. Discuss its potential and its limitations, and see how it compares to existing approaches, such as DY*.

5 Extension Goals

- **Case study of a solution based on linear types.** The Gobra verifier may benefit from a linear type system in the future. Byte-arrays with the new memory predicate act like linear resources that are created by a particular library call and have to be deleted by another one. Investigate what a solution to our initial problem might look like if we take advantage of the possibilities offered by a linear type system.
- **Verification of an existing protocol.** Find an existing protocol that uses memory deletion in order to obtain security properties such as forward secrecy and post-compromise security. Find an implementation of this protocol in Go, if possible an official implementation, and verify it using the developed methodology and extended verification library.

References

[1] S. Meier, B. Schmidt, C. Cremers, and D. 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.* Springer, 2013, pp. 696–701.

[2] B. Blanchet *et al.*, "Modeling and verifying security protocols with the applied pi calculus and proverif," *Foundations and Trends® in Privacy and Security*, vol. 1, no. 1-2, pp. 1–135, 2016.

[3] L. Arquint, M. Schwerhoff, V. Mehta, and P. Müller, "A generic methodology for the modular verification of security protocol implementations," *arXiv preprint arXiv:2212.02626*, 2022.

[4] M. Marlinspike and T. Perrin, "The x3dh key agreement protocol," *Open Whisper Systems*, vol. 283, p. 10, 2016.

[5] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on information theory*, vol. 29, no. 2, pp. 198–208, 1983.

[6] K. Bhargavan, A. Bichhawat, Q. H. Do, P. Hosseyni, R. Küsters, G. Schmitz, and T. Würtele, "DY*: A modular symbolic verification framework for executable cryptographic protocol code," in *2021 IEEE European Symposium on Security and Privacy (EuroS&P).* IEEE, 2021, pp. 523–542.

[7] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller, "Gobra: Modular specification and verification of go programs," in *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I 33.* Springer, 2021, pp. 367–379.

[8] T. Perrin and M. Marlinspike, "The double ratchet algorithm," *GitHub wiki*, p. 10, 2016.

[9] K. Cohn-Gordon, C. Cremers, and L. Garratt, "On post-compromise security," in *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, 2016, pp. 164–178.

[10] P. Müller, M. Schwerhoff, and A. 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.* Springer, 2016, pp. 41–62.