Automated Identification of Desynchronisation Attacks on Shared Secrets

Sjouke Mauw[1,2], Zach Smith[1], Jorge Toro-Pozo[1], and Rolando Trujillo-Rasua[2,3]

[1] CSC, University of Luxembourg
[2] SnT, University of Luxembourg
[3] School of Info Technology, Deakin University

Abstract. Key-updating protocols are a class of communication protocol that aim to increase security by having the participants change encryption keys between protocol executions. However, such protocols can be vulnerable to desynchronisation attacks, a denial of service attack in which the agents are tricked into updating their keys improperly, so that they are no longer able to communicate. In this work we introduce a method that can be used to automatically verify (or falsify) resistance to desynchronisation attacks for a range of protocols. This approach is then used to identify previously unreported vulnerabilities in two published RFID grouping protocols.

1 Introduction

Key-updating protocols form a class of communication protocols in which participants change their encryption keys between executions. Such protocols are used in several domains - the Signal protocol uses the Diffie-Hellman Double Ratchet algorithm [19], and the Gossamer protocol [18] also uses updating keys. Many grouping protocols [12, 21], which aim to prove that two or more RFID tags are simultaneously present, also make use of updating keys.

There are several formally defined security properties which demonstrate the benefits of key-updating protocols. For example, *forward privacy*, introduced by Avoine [2], prevents an attacker from learning about past sessions, even after compromising a participant. *Post-compromise security*, as defined by Cohn-Gordon et al. [5], states that if an adversary compromises an agent, their influence can be reversed if they do not continually monitor communication.

Such goals are typically realised by security protocols which update encryption keys, for example by using a one-way hash function. This way, if an adversary learns the encryption keys used in a single session, they cannot reconstruct past keys. However such methods introduce the problem of requiring the protocol participants to *synchronise* their key updates - so that their local states remain consistent.

The synchronisation requirement of key-updating protocols has created new attack vectors. If improperly designed or implemented, an attacker can cause

agents to update their keys in an improper manner, preventing them from correctly interpreting communications from their partner. This kind of Denial-of-Service attack is called a *desynchronisation attack* [7]. Such attacks allow an adversary to prevent future runs of a communication protocol, stopping the protocol from achieving its intended purpose.

Security properties for communication protocols can be formally verified using symbolic analysis. This type of analysis is well-supported by a range of automated proving tools such as ProVerif [3] and Tamarin [17], which typically attempt to reduce analysis of the protocol to a bounded case. This is especially true in the case of *stateless* protocols, where information between sessions is never carried forwards to future executions. However, key-updating protocols are inherently *stateful* - information must be preserved between sessions. This can cause problems in analysis due to the explosion of the state space. Indeed, reachability queries are in general an undecidable problem [10, 4].

Existing formalisms of desynchronisation resistance. Desynchronisation represents a class of attacks that are not covered by traditional definitions. A protocol that is impervious to such attacks is said to be *desynchronisation resistant*, and while there is a strong intuitive understanding of what this property means, there are few attempts at formal definitions in the literature.

There exist a variety of works that either claim a form of desynchronisation resistance [25, 15, 13, 22] or provide a desynchronisation attack on published protocols [14, 16, 23]. Both types of papers only provide an informal treatment of the topic, without automated tool support. Only few papers provide a formal definition of a desynchronisation attack or desynchronisation resistance. We will briefly discuss two of these approaches, namely the work of Van Deursen et al. [6] and the work of Radomirović and Dashti [20].

Van Deursen et al. [6] introduce desynchronisation in the context of RFID protocols. They say an RFID reader *owns* a tag if it knows a secret key allowing it to authenticate the tag in absence of the adversary. A protocol is then said to be desynchronisation resistant if being owned is an *invariant* property. For example, if there is a time t such that a tag T is owned by a reader R, then at time $t + 1$ there must exist some reader R' (who may be the same or different to R) which 'owns' T. The authors demonstrate how existing RFID protocols violate their definition. They do not provide, however, any means for formally verifying that it holds for an arbitrary protocol.

A second existing approach that relates to desynchronisation resistance is the work on *derailing attacks* by Radomirović and Dashti [20]. In a *derailing attack*, a protocol is led away from its intended execution by an adversary. Reachable states in the protocol are labelled as *safe*, *unsafe*, or *transitional*, describing whether a desirable 'success' condition is reachable from the current point. A protocol is said to be *susceptible to derailing attacks* if there exists a reachable state S such that in absence of the adversary, there are no safe states that are reachable from S.

Contributions. In this paper, a formal definition of desynchronisation resistance is given in terms of the traces of a security protocol. The definition we provide can be seen as an extension of the two theories above. Like Radomirović and Dashti, our definition concerns the reachability of certain states, and an examination of the transitions between them. Like Deursen et al., the knowledge of secret keys is an important factor in our definitions. However, we go further by providing a set of conditions for key-updating protocols that allows for automated verification (or falsification) of desynchronisation.

Organisation. In Section 2, a detailed introduction to multiset rewriting theory is given, presenting the language that will be used throughout the paper. In Section 3, a series of definitions regarding reachability are provided, and used to create a formal definition of desynchronisation resistance. In Section 4, the model is refined to focus on sequential key-updating protocols. A set of security properties are provided that are proved to be sufficient to ensure desynchronisation resistance in this setting. Section 5 shows the result of applying this analysis to existing secret-updating protocols by using the automated verification tool Tamarin. Novel attacks are found on a number of protocols in the literature. Finally in Section 6, we discuss future work, as well as related concepts.

2 Security Protocol Model

In order to model security protocols in which shared secrets are updated, a *multiset rewriting* model will be used. Multiset rewriting is a common basis for modelling stateful systems. In a stateful system, different sessions can be dependent on each other, with information that is dynamic between executions.

A protocol specification covers a set of rules that govern how a multiset describing the protocol state is allowed to proceed. This state consists of information such as the messages that have been sent by different participants of the protocol, markers denoting if certain stages of the protocol have been successfully reached, and the knowledge and actions of an adversary who seeks to undermine the protocol's successful execution.

2.1 Multiset Rewriting

The multisets used in our model are built on terms constructed from an order-sorted signature, such as those described by by Goguen and Meseguer [11]. An *order-sorted signature* is a triple $(\mathcal{S}, \leq, \Sigma)$, where $\leq$ is a partial ordering on a set of types $\mathcal{S}$, and Σ is a collection of functions between types. For two types s and t we define $\Sigma_{s,t}$ to be the functions in Σ which map from type s to type t. Further, we use the standard notation for the Cartesian product of sets, so for example:

$$f \in \Sigma_{\mathbb{R}^2,\mathbb{N}} \ ::= \ f \colon \mathbb{R} \times \mathbb{R} \to \mathbb{N}.$$

Our model must track not only the kinds of communications that can be sent over a network, but also auxiliary information about the state, such as an

agent's encryption keys. To do this, we define two top types msg and fact, and further define subtypes public, nonce $<$ msg, and agent, const $<$ public.

The set of terms over $\mathcal{S}$ is defined iteratively, as follows. First, for each type $s \in \mathcal{S}$ we build two infinite carrier sets N_s and V_s of *names* (i.e. known values) and *variables* (i.e. unknown or uninstantiated values) of type s. We refer to these types of terms as *atoms*. We will often use the following notation for variables:

$$x, y\colon \mathsf{nonce}, \qquad m, k\colon \mathsf{msg}, \qquad A, B\colon \mathsf{agent}.$$

From here, successive terms are built by the application of functions from Σ on the atoms. Given a term t, we define the set of *subterms* of t as follows. If t is an atom, then $subterms(t) = \{t\}$. Otherwise, we have $t = f(t_1, t_2, \ldots, t_n)$ for some function symbol $f \in \Sigma$. In this case, we define

$$subterms(t) = \{t\} \cup \{subterms(t_1), \ldots, subterms(t_n)\}.$$

A term t is *ground* if $subterms(t) \cap V_s = \emptyset$, and we denote the set of all (ground) terms of type s as Ter_s ($GTer_s$). A (ground) substitution σ is a partial function from variables to (ground) terms of the same type or supertype. Given a substitution σ and a term t, we write $t\sigma$ to denote the application of the substitution. Given a set $S = \{t_1, \ldots, t_n\}$, we write $S\sigma = \{t_1\sigma, \ldots, t_n\sigma\}$. We say σ is a *grounding substitution* for S if all terms in $S\sigma$ are ground.

The model is extended with an equational theory E, which describes the semantics of the functions in Σ. Pairs (l, r) in E define an equivalence relation $\simeq_E$ on terms constructed using $(\mathcal{S}, \leq, \Sigma)$.

Example 1. We define the pair operator $\langle \_, \_ \rangle \in \Sigma_{\mathsf{msg}\times\mathsf{msg},\mathsf{msg}}$, and the corresponding projection functions $fst, snd \in \Sigma_{\mathsf{msg},\mathsf{msg}}$ such that $fst(\langle x, y\rangle) = x$ and $snd(\langle x, y\rangle) = y$.
Standard reason applies to equations, for example, $fst(\langle\langle x, y\rangle, z\rangle) \simeq_E \langle x, y\rangle$.

A multiset is a set, M, counted with multiplicity - an individual element k can be represented multiple times, and we write $|k|_M$ to denote the number of occurrences of k in M, with $|k|_M = 0$ if $k \notin M$. Given a set S, we write $\mathcal{M}(S)$ to denote the collection of all multisets that can be written using elements of S.

The multisets we will study are a restricted subset of those constructible using the order-sorted signature $(\mathcal{S}, \leq, \Sigma)$ above. In particular, we define the *universe* of states, $\mathbb{U}(\Sigma)$ as:

$$\mathbb{U}(\Sigma) = \mathcal{M}(\{f(t_1, \ldots, t_i) \mid i \geq 0 \wedge f \in \Sigma_{\mathsf{msg}^i,\mathsf{fact}} \wedge \forall k \in \{1 \ldots i\}.\ t_k \in GTer_{msg}\}).$$

Each element $S \in \mathbb{U}(\Sigma)$ represents a single valid *state* of a protocol execution. We now look at how we can move from one state to the next.

A *rule* r is defined by a pair (lhs, rhs) of multisets. Suppose σ is a grounding substitution for lhs. A *rule application* $r\sigma$ is a mapping $\mathbb{U}(\Sigma) \to \mathbb{U}(\Sigma)$. It acts on a state $S \in \mathbb{U}(\Sigma)$ by identifying a submultiset of S equal to $\sigma(lhs)$, and replacing

it with $\sigma(rhs)$. Note that multiset rules must respect the equational theory E, so that $S \simeq_E S' \implies r\sigma(S) \simeq_E r\sigma(S')$. We express protocol rules as labelled transitions.

Example 2. Consider the protocol rule `Combine`:

$$\frac{\mathrm{A}(x) \quad \mathrm{A}(y)}{\mathrm{B}(x,y)}\ \texttt{Combine},$$

which takes two terms of type fact built with symbol A, and returns a new fact which contains the subterms of the two previous terms. Let $S = \{\mathrm{A}(a), \mathrm{A}(b), \mathrm{A}(c)\}$. The substitution $\sigma = \{x \mapsto a, y \mapsto b\}$ maps:

$$\{\mathrm{A}(a), \mathrm{A}(b), \mathrm{A}(c)\} \xrightarrow{r\sigma} \{\mathrm{B}(a,b), \mathrm{A}(c)\}$$

Definition 2.1 (Protocol specification) *A* protocol specification P *is defined by a tuple* $(\Sigma, E, R, S^{start})$ *where:*

- $\Sigma = (F, \mathscr{F})$ *is a collection of function symbols of signature types* $\Sigma_{\mathsf{msg}^*,\mathsf{msg}}$ *and* $\Sigma_{\mathsf{msg}^*,\mathsf{fact}}$*, respectively.*
- E *is an equational theory over* $\Sigma_{\mathsf{msg}^*,\mathsf{msg}}$*.*
- R *is a collection of* rules*.*
- $S^{start} \subseteq \mathbb{U}(\Sigma)$ *is a collection of potential starting states.*

The set of starting states will usually be infinite, as they carry the details of a specific execution - the number of participating agents, their encryption keys, and so on.

A *trace*, τ, on P is a choice of starting state $S^0 \in S^{start}$ and a finite ordered list of rule applications $(r_1\sigma_1 \dots r_n\sigma_n)$ such that each successive application $S^0 \xrightarrow{r_1\sigma_1} \dots \xrightarrow{r_n\sigma_n} S^n$ is valid.

The intermediate states in a trace can be reconstructed from the choices of rule applications. Given a trace $\tau = (S^0, (r_1\sigma_1 \dots r_n\sigma_n))$, a second trace τ' is an *extension* of τ, writing $\tau \sqsubseteq \tau'$, if $\tau' = (S^0, (r_1\sigma_1 \cdots r_n\sigma_n \dots r_{n+k}\sigma_{n+k}))$.

Given a trace τ we write $\mathrm{firstState}(\tau)$ and $\mathrm{lastState}(\tau)$ to denote the first state and the (implicit) last state in the trace. We write $\mathrm{rules}(\tau)$ to denote the set of rules $\{r_1, \dots, r_n\}$ in τ. We write $\mathrm{traces}(P)$ to denote the set of all possible traces on the protocol P.

We define an *event fact*, $\mathrm{E}^\star$ to be a fact which appears only on the right-hand side of rules in R. Such facts can never be removed from the state of the protocol. Intuitively, while standard facts mark the *current* situation of a state, event facts form an indelible history of all important occurrences in a trace. As such, we define the multiplicity of an event fact in a trace without ambiguity as $|\mathrm{E}^\star(t_1 \dots t_n)|_\tau := |\mathrm{E}^\star(t_1 \dots t_n)|_{\mathrm{lastState}(\tau)}$,

We define a quasi-order on event facts within traces, $<_\tau$, as follows. Given two event facts $\mathrm{E}^\star(t_1 \dots t_n), \mathrm{F}^\star(s_1 \dots s_m)$, we say $\mathrm{E}^\star(t_1 \dots t_n) <_\tau \mathrm{F}^\star(s_1 \dots s_m)$ if there exists a prefix $\tau' \sqsubseteq \tau$ such that:

$$\big(|\mathrm{E}^\star(t_1 \dots t_n)|_{\tau'} > 0\big) \wedge \big(|\mathrm{F}^\star(s_1 \dots s_m)|_{\tau'} = 0\big) \wedge \big(|\mathrm{F}^\star(s_1 \dots s_m)|_{\tau} > 0\big).$$

In particular, this means that $\mathrm{F}^\star$ was added to the state at some point after $\mathrm{E}^\star$. In addition, we write $\mathrm{E}^\star(t_1 \dots t_n) \leq_\tau \mathrm{F}^\star(s_1 \dots s_m)$ to indicate that $\mathrm{E}^\star(t_1 \dots t_n) <_\tau \mathrm{F}^\star(s_1 \dots s_m)$ or $\{\mathrm{E}^\star(t_1 \dots t_n), \mathrm{F}^\star(s_1 \dots s_m)\} \subseteq \mathrm{firstState}(\tau)$.

We reserve several symbols in $\mathscr{F}$ for all protocols, with the following interpretations:

- Net(msg) represents a message on the communication network.
- Fr(nonce) represents that the nonce in the argument has been freshly generated. By convention, we require that freshly generated terms are atomic.
- K(msg) represents that the adversary 'knows' the term in the argument.

Additional event facts are introduced as a consequence of the security requirements of the protocol being analysed. In Section 4, we will introduce several more event fact symbols used in order to analyse key-updating protocols.

2.2 The Adversary

An important concept in discussing security properties is an understanding of the adversary's capabilities. In this work, the Dolev-Yao adversary model [8] is used. The Dolev-Yao adversary is assumed to have full control over the communication network. We make the *perfect cryptography assumption*: the adversary is incapable of decrypting messages without the appropriate key.

The adversary knowledge is modelled using facts K. The initial knowledge of the adversary is defined by the starting states of the protocol specification, but at a minimum contains all terms of type public. A set of additional protocol rules describe the capabilities of the adversary. These protocol rules allow the adversary to eavesdrop, block or modify messages that are sent on the communication network. We assume that all protocols being studied contain (at least) the set of adversary rules provided in Figure 1. The set of rules which model the actions of the adversary is denoted as *Adv*.

$$\frac{\mathrm{Net}(x)}{\mathrm{Net}(x) \quad \mathrm{K}(x)}\ \texttt{Eavesdrop} \qquad \frac{\mathrm{Net}(x)}{\mathrm{K}(x)}\ \texttt{Block}$$

$$\frac{\mathrm{K}(x)}{\mathrm{K}(x) \quad \mathrm{Net}(x)}\ \texttt{Inject} \qquad \frac{\mathrm{K}(x_1, \dots, x_n)}{\mathrm{K}(f(x_1, \dots, x_n))}\ \texttt{Function}$$

Fig. 1: The minimal set of adversary rules.

We often also grant the adversary the limited ability to *corrupt* an agent, learning the value of any secret keys they hold. This is done through either the choice of starting states, or additional adversary rules.

2.3 Security Claims

Given a protocol P, a *security claim* on P is a first-order logic statement about the existence and ordering of event facts in traces of P. We note that the validity of security claims is dependent upon a faithful description of the protocol in

question. For example, in order to make security claims about the *secrecy* of certain knowledge, we should expect the protocol specification to contain $\text{Secret}^\star(t)$ (or similar) facts denoting the terms that are believed to be secret.

3 Desynchronisation Resistance

The intuition behind desynchronisation is that the protocol reaches a state from which it can no longer proceed in a meaningful way. In order to define precisely what this means, we must start with a notion of *reachability*. We refine this definition to progressively stronger versions, before introducing our definition of *desynchronisation resistance*.

Reachability is a property describing the ability of the protocol to transition from a given state to some desirable situation. We will want to ensure that in any reasonable conditions, the adversary cannot prevent the protocol from completing, but rather only delay it.

Definition 3.1 (State Reachability) *Given a protocol $P = (\Sigma, E, R, S^{start})$, a set of rules $W \subseteq R$ and two states $S, S' \in \mathbb{U}(\Sigma)$, we say that S' is* reachable from S avoiding W*, denoted by $S \rightsquigarrow_{\neg W} S'$, if:*

$$\forall \tau \in \text{traces}(P).\, \text{lastState}(\tau) = S \implies \\ \exists \tau' \in \text{traces}(P). \tau \sqsubseteq \tau' \wedge \text{lastState}(\tau') = S' \wedge \text{rules}(\tau' \setminus \tau) \cap W = \emptyset.$$

Note that we pay particular attention to the idea of reachability avoiding certain rules. We wish to show that no matter which actions an adversary takes, it is possible for the execution of a protocol to continue once the adversary becomes inactive. As such, we use $\rightsquigarrow_{\neg Adv}$ to denote reachability in absence of the adversary, and $\rightsquigarrow$ for the particular case when no rules are forbidden.

Given a protocol $P = (\Sigma, E, R, S^{start})$ and a state $S \in \mathbb{U}(\Sigma)$ we define the set of states reachable from S as $\text{reachable}(S) = \{S' \in \mathbb{U}(\Sigma) \mid S \rightsquigarrow S'\}$. Overloading notation, we define the set of states reachable by P as $\text{reachable}(P) = \bigcup_{S^0 \in S^{start}} \text{reachable}(S^0)$.

Next, the notion of reachability is extended from the context of states to the context of event facts.

Definition 3.2 (Event Reachability) *Let P be a protocol, $S \in \mathbb{U}(\Sigma)$ a state, W a set of rules and $\text{E}^\star$ an event fact. We say that $\text{E}^\star$ is* reachable from S avoiding W*, denoted by $S \rightsquigarrow_{\neg W} \text{E}^\star$, if:*

$$\exists S' \in \mathbb{U}(\Sigma).\ (S \rightsquigarrow_{\neg W} S') \wedge (|\text{E}^\star|_S < |\text{E}^\star|_{S'}).$$

Intuitively, given a trace τ that contains S, it is possible to extend τ in such a way that the event fact $\text{E}^\star$ is reached. Like before, we will write $S \rightsquigarrow \text{E}^\star$ to indicate $S \rightsquigarrow_{\neg \emptyset} \text{E}^\star$.

Reachability captures the idea that a desired state or event can be achieved once. However, we desire that our protocol not only be able to successfully

complete once, but arbitrarily many times. To do this, we need a definition stronger than standard reachability.

Desynchronisation occurs when two agents who were originally able to finish a protocol execution lose this ability.

Definition 3.3 (Desynchronisation Resistance) *A protocol P is* desynchronisation resistant *if:*

$$\begin{aligned}\forall A, B\colon \mathsf{agent}, S^0 \in S^{start}.\ S^0 \rightsquigarrow_{\neg Adv} \mathrm{Complete}^\star(A,B) \implies \\ \big(\forall \tau \in \mathrm{traces}(P).\ \mathrm{firstState}(\tau) = S^0 \implies \\ \mathrm{lastState}(\tau) \rightsquigarrow_{\neg Adv} \mathrm{Complete}^\star(A,B) \vee \\ \mathrm{Corrupt}^\star(A) \in \tau \ \vee \mathrm{Corrupt}^\star(B) \in \tau\big).\end{aligned}$$

Intuitively, if A and B are able to complete the protocol once without any actions being performed by the adversary, then they will always be able to do this, except in the case that one of the participants been corrupted, willingly giving secret data to the adversary.

4 Verifying Desynchronisation Resistance

In this section we look at a specific instantiation of the theory in the previous sections, and show that it can be used to verify desynchronisation resistance. We also provide a 'lower bound' to desynchronisation resistance, proving that violating this combination of properties results in an attack.

4.1 A Sequential Key Updating Environment

We will model a common synchronous authentication environment. In this scenario, a pair of agents each store a number of secret communication keys to be used with their intended partner. In an ideal execution, the keys stored by one agent will always correspond to those stored by their partner, regardless of the state of execution or the actions of the adversary.

Recall that a protocol specification is defined by a tuple $(\Sigma, E, R, S^{start})$, where Σ is further divided into the collections F and $\mathscr{F}$ of functions on terms and fact symbols. We provide next a framework composed of F, E, and $\mathscr{F}$. Depending on the protocol, it may be necessary to extend the equational theory. The set of rules R is a consequence of the protocol being examined.

$$\begin{aligned}F = \{&senc\colon \mathsf{msg}\times\mathsf{msg}\to\mathsf{msg},\ sdec\colon \mathsf{msg}\times\mathsf{msg}\to\mathsf{msg},\\ &aenc\colon \mathsf{msg}\times\mathsf{msg}\to\mathsf{msg},\ adec\colon \mathsf{msg}\times\mathsf{msg}\to\mathsf{msg},\\ &pk\colon \mathsf{msg}\to\mathsf{msg},\ h\colon \mathsf{msg}\to\mathsf{msg}\}.\\ E = \{&sdec(senc(msg,key),key) = msg,\\ &adec(aenc(msg,pk(ltk)),ltk) = msg\}.\end{aligned}$$

The function symbols in F represent the standard symmetric and asymmetric encryption and decryption functions, and E defines their semantics.

$$\mathscr{F} = \{\, \mathrm{ShKeys}(\mathsf{agent}, \mathsf{agent}, \langle \mathsf{nonce}, \ldots \rangle),\ \mathrm{Session}(\mathsf{agent}, \mathsf{agent}, \langle \mathsf{msg}, \ldots \rangle),\ \mathrm{AddKey}^\star(\mathsf{agent}, \mathsf{agent}, \mathsf{msg}),\ \mathrm{DropKey}^\star(\mathsf{agent}, \mathsf{agent}, \mathsf{msg}),\ \mathrm{Complete}^\star(\mathsf{agent}, \mathsf{agent}) \,\}.$$

The facts ShKeys and Session provide information about the knowledge of an agent. ShKeys facts represent their *long term* knowledge, in the form of communication keys for use with a named partner. Session facts are used to store session data for a single execution of the protocol. The AddKey* and DropKey* event facts mark changes to the stored keys of an agent. Further semantics to reinforce this intent are provided later.

Definition 4.1 (Starting States) *The set of starting states S^{start} is the set composed of all $S^0 \in \mathbb{U}(\Sigma)$ that satisfy the following conditions:*

(i) $\nexists x\colon \mathsf{msg}.\, \mathrm{Net}(x) \in S^0$,

(ii) $\nexists A, B\colon \mathsf{agent}, y\colon \mathsf{msg}.\ \mathrm{Session}(A, B, y) \in S^0$,

(iii) $\forall A, B\colon \mathsf{agent}, k_1, \ldots, k_n\colon \mathsf{msg}.\ \mathrm{ShKeys}(A, B, \langle k_1, \ldots, k_n \rangle) \in S^0 \implies \nexists l_1, \ldots, l_m\colon \mathsf{msg}, \langle k_1, \ldots, k_n \rangle \neq \langle l_1, \ldots, l_m \rangle.\ \mathrm{ShKeys}(A, B, \langle l_1, \ldots, l_m \rangle) \in S^0$,

(iv) $\forall A, B\colon \mathsf{agent}, k_i\colon \mathsf{msg}.\ \mathrm{ShKeys}(A, B, \langle \ldots k_i \ldots \rangle) \in S^0 \implies \mathrm{AddKey}^\star(A, B, k_i) \in S^0$,

(v) $\forall A, B\colon \mathsf{agent}, k\colon \mathsf{msg}.\ \mathrm{AddKey}^\star(A, B, k) \in S^0 \iff \exists k_1 \ldots k_n\colon \mathsf{msg}.\ \mathrm{ShKeys}(A, B, \langle \ldots k \ldots \rangle) \in S^0 \ \vee\ \mathrm{DropKey}^\star(A, B, k) \in S^0$

(vi) $\forall A, B\colon \mathsf{agent}, k\colon \mathsf{nonce}.\ \big(\mathrm{ShKeys}(A, B, \langle \ldots k \ldots \rangle) \in S^0 \wedge \mathrm{K}(k) \in S^0\big) \implies \mathrm{Corrupt}^\star(A) \in S^0 \vee \mathrm{Corrupt}^\star(B) \in S^0$.

We note the following intuitions behind the above requirements:

(i) A starting state may not contain messages.
(ii) A starting state may not contain session data.
(iii) An agent stores only one set of keys for use with each potential communication partner.
(iv) If a starting state contains an agent A who stores a secret key k_i for communicating with an agent B, then there is a corresponding AddKey* fact showing that A has added this key.
(v) If a starting state contains an AddKey* fact, then either the corresponding agent has that key in their knowledge, or there is also a corresponding DropKey* fact.
(vi) If a starting state contains an agent A who stores a secret key k_i for communicating with an agent B, and the adversary knows the value k_i, then either A or B is corrupt.

We point out that a starting state does allow for instances of the Complete* event fact. This does not interfere with any reachability claims, as these describe the ability to add *new* instances of these event facts to the trace.

In addition, we grant the adversary two capabilities. Firstly, the adversary is able to "corrupt" an agent, learning any secret keys they are holding. Second, we allow the adversary to "cancel" the session of an agent, causing them to lose any stored session data. For example, this models the ability of an adversary to block messages sent on the network until an agent assumes their partner has halted communication. We do this by requiring that the set of rules R contains the rules `Corrupt` and `Sess_Cancel`, defined below.

$$\frac{\text{ShKeys}(A, B, \langle k_1 \dots k_n \rangle)}{\text{K}(\langle k_1 \dots k_n \rangle)}\ \texttt{Corrupt} \qquad \frac{\text{Session}(A, B, y)}{}\ \texttt{Sess\_Cancel}$$

4.2 Satisfying Desynchronisation Resistance

Given a protocol constructed in the model above, we provide a set of conditions that are sufficient to satisfy desynchronisation resistance.

We start with a predicate stating whether two agents share a *common key* in a given state. Let P be a protocol and $S \in \text{reachable}(P)$. We say that two agents A and B have a common key in S, denoted $\text{CommonKey}_{A,B}(S)$, if and only if:

$$\begin{aligned}\exists k_1, \dots, k_n, l_1, \dots, l_m \colon \mathsf{msg}.\ \big(&\{k_1, \dots, k_n\} \cap \{l_1, \dots, l_m\} \neq \emptyset \wedge \\ &\text{ShKeys}(A, B, \langle k_1, \dots, k_n \rangle) \in S \wedge\ \text{ShKeys}(B, A, \langle l_1, \dots, l_m \rangle) \in S\big)\ .\end{aligned}$$

Now we define *reachability conditional on a common key* as the property of a protocol that two agents are able to complete the protocol with each other in absence of the adversary if and only if they have a common key.

Property 4.2 (Reachable Conditional on Common Key) *We say that P satisfies* completion conditional on a common key *if:*

$$\begin{aligned}&\forall S^0 \in S^{start}, A, B \colon \mathsf{agent} \colon \\ &\quad S^0 \rightsquigarrow_{\neg Adv} \text{Complete}(A, B) \iff \text{CommonKey}_{A,B}(S^0).\end{aligned}$$

With these in mind, we now define several other properties describing the nature in which the shared keys used by agents in a protocol are updated. Property 4.3 and Property 4.4 give syntactic requirements on protocols. In particular, we require that a protocol's specification is consistent in the way that ShKeys linear facts are modified with respect to the addition of the AddKey* and DropKey* event facts. We also make the assumption that an agent always stores the same number of encryption keys for communicating with their partner.

Property 4.3 (Well-Formed Key Updates) *A protocol $P = (\Sigma, E, R, S^{start})$ satisfies* Well-Formed Key Updates *if the following two conditions hold for all rules $r \in R$:*

$$\begin{aligned}
&\mathrm{AddKey}^\star(A,B,k) \in rhs(r) \iff \\
&\quad \big(\exists\, k_1 \ldots k_n, l_1 \ldots l_m\,.\ \mathrm{ShKeys}(A,B,\langle k_1 \ldots k \ldots k_n\rangle) \in rhs(r) \wedge \\
&\qquad\qquad \mathrm{ShKeys}(A,B,\langle l_1 \ldots l_m\rangle) \in lhs(r)\ \wedge\ \forall\, i\,.\, l_i \neq k\big)\ , \\
&\mathrm{DropKey}^\star(A,B,k) \in rhs(r) \iff \\
&\quad \big(\exists\, k_1 \ldots k_n, l_1 \ldots l_m\,.\ \mathrm{ShKeys}(A,B,\langle k_1 \ldots k \ldots k_n\rangle) \in lhs(r) \wedge \\
&\qquad\qquad \mathrm{ShKeys}(A,B,\langle l_1 \ldots l_m\rangle) \in rhs(r)\ \wedge\ \forall\, i\,.\, l_i \neq k\big)\ .
\end{aligned}$$

Next we define the *Key Conservation* property. It states that every agent must keep the same number of keys during the execution of the protocol. We also require each rule to consider at most a single shared key fact.

Property 4.4 (Key Conservation) *A protocol* $P = (\Sigma, E, R, S^{start})$ *satisfies* Key Conservation *if for every rule* $r \in R$*, and every* $A, B\colon \mathsf{agent},\ k_1, \ldots, k_n\colon \mathsf{msg}$*, there exists an instance of* $\mathrm{ShKeys}(A, B, \langle k_1, \ldots, k_n\rangle)$ *on the left-hand side of* r *if and only if there is some* $l_1, \ldots, l_n\colon \mathsf{msg}$ *such that the right-hand side of* r *contains* $\mathrm{ShKeys}(A, B, \langle l_1, \ldots, l_n\rangle)$*.*

Next we define *Key Uniqueness* as the notion that a given encryption key will only be generated at most once. Once discarded by an agent they will never re-use it, nor can a different pair of agents ever (intentionally or otherwise) generate the same encryption key.

Definition 4.5 (Key Uniqueness) *A protocol* P *satisfies* Key Uniqueness *if for every* $\tau \in \mathrm{traces}(P)$ *and every* $A, B, A', B'\colon \mathsf{agent}$ *and every* $k\colon \mathsf{msg}$ *with* $\{A, B\} \neq \{A', B'\}$ *it holds that:*

$$\begin{aligned}
&\mathrm{AddKey}^\star(A,B,k) \in \tau \implies \\
&\quad |\mathrm{AddKey}^\star(A,B,k)|_\tau = 1 \wedge |\mathrm{AddKey}^\star(A',B',k)|_\tau = 0.
\end{aligned}$$

We next describe the properties of *Key Preparedness* and *Key Resilience*. Together with Key Uniqueness, these are the main security requirements that are to be verified. Intuitively, they provide a semi-strict ordering on the key updates of paired agents.

Definition 4.6 (Key Preparedness for agents A and B) *A protocol* P *satisfies* Key Preparedness for agents A and B *if*

$$\begin{aligned}
&\forall \tau \in \mathrm{traces}(P), \forall k\colon \mathsf{msg} \\
&\quad \mathrm{AddKey}^\star(A,B,k) \in \tau \implies \mathrm{AddKey}^\star(B,A,k) \leq_\tau \mathrm{AddKey}^\star(A,B,k).
\end{aligned}$$

Definition 4.7 (Key Resilience for agents A and B) *A protocol* P *satisfies* Key Resilience for agents A and B *if*

$$\begin{aligned}
&\forall \tau = (S^0, (r_i\sigma_i)) \in \mathrm{traces}(P), \forall k\colon \mathsf{msg} \\
&\quad \mathrm{DropKey}^\star(A,B,k) \in \tau \implies \\
&\qquad \mathrm{DropKey}^\star(B,A,k) \leq_\tau \mathrm{DropKey}^\star(A,B,k)\ \vee\ \mathrm{DropKey}^\star(A,B,k) \in S^0.
\end{aligned}$$

The second case in the Key Resilience claim accounts for the trivial case of a starting state containing DropKey* facts for which we cannot be sure of the source.

We note that the above properties are verifiable, either by examination of the protocol specification (4.2, 4.3, 4.4), or through verification of traces in an automated prover tool (4.5, 4.6, 4.7) . We denote the properties as WF, KC, KU, KP and KR respectively for Well Formedness, Key Conservation, Key Uniqueness, Key Preparedness and Key Resilience.

Theorem 4.8 (Sufficiency) *Let $P = (\Sigma, E, R, S^{start})$ be a protocol that satisfies Properties 4.2, 4.3, 4.4 and Definition 4.5. P satisfies desyncronisation resistance if for all $S^0 \in S^{start}$ and all agents A, B such that* $\text{CommonKey}_{A,B}(S^0)$*, one of the following conditions holds:*

- *Key Preparedness (Definition 4.6) for agents A and B holds, and Key Resilience (Definition 4.7) for agents B and A holds, or*
- *Key Preparedness (Definition 4.6) for agents B and A holds, and Key Resilience (Definition 4.7) for agents A and B holds.*

Before we begin the proof of Theorem 4.8, we provide some helper lemmas. We define the *strip*() function, which allows us to transform a state into a starting state.

Definition 4.9 (Strip Function) *We define the function strip(), which maps from states to states. We define strip(S) to be the multiset that is equal to S, but with all instances of* Session, K *and* Net *removed.*

Lemma 4.10 *Let P be a protocol which satisfies Key Conservation (Property 4.4) and Well-Formed Key Updates (Property 4.3). Suppose $S \in \text{reachable}(P)$. Then strip($S$) is a starting state of this protocol, as per the requirements of starting states in Definition 4.1.*

Proof. Points (i), (ii) and (vi) are immediate from the absence of corresponding facts. (iii) is a consequence of Key Conservation, (iv) and (v) from Well-Formed Key Updates. □

Lemma 4.11 *Let P be a protocol which satisfies Key Conservation (Property 4.4) and Well-Formed Key Updates (Property 4.3), and τ a trace of P with final state S. Suppose γ is a trace of P with starting state strip(S) that contains no adversary rules. Then $\gamma \cdot \tau \in \text{traces}(P)$ is a trace extension of τ.*

Proof. Suppose $\gamma = (strip(S), r_1\sigma_1 \dots r_n\sigma_n)$. We claim that the series of rule applications $r_1\sigma_1 \dots r_n\sigma_n$ are valid from the state S. Indeed, the rule application $r_1\sigma_1$ can be dependent only on ShKeys facts, as these are the only linear facts which can be in a starting state. These facts exist in both S and *strip*(S). By the same logic, the rest of the series of applications are also valid. □

Proof (Theorem 4.8). Assume that the agents A and B are not corrupt. Without loss of generality, we assume the first case holds - that we have Key Preparedness for A and B, and Key Resilience for B and A. Our proof proceeds in two steps. First, we show that the common key predicate is sufficient to ensure completion from *any* state, not just the starting states:

$$\forall S \in \text{reachable}(P):$$
$$\text{CommonKey}_{A,B}(S) \implies S \rightsquigarrow_{\neg Adv} \text{Complete}(A, B).$$

Secondly, we show that the common key property is *invariant*:

$$\forall S \in \text{reachable}(P), r \in R,$$
$$(\text{CommonKey}_{A,B}(S) \wedge S \xrightarrow{r\sigma} S') \implies \text{CommonKey}_{A,B}(S').$$

From these two claims, the result will immediately follow. To show the first point, we use the *strip*() function from Definition 4.9. Note that if A and B have a common key in S, then they have a common key in *strip*(S). Then, by Lemma 4.11, the claim follows.

For the second point, we must show that for any rule application $r\sigma$ in which a DropKey* event fact is added, the common key predicate is preserved. Indeed, the well-formedness properties of Property 4.3 ensure that these are the only possible rule applications which can affect the predicate.

Suppose we have $S \in \text{reachable}(P)$ such that $\text{CommonKey}_{A,B}(S)$, and a rule application $r_n\sigma_n$. We split into the cases when DropKey$^\star(A, B, k)$ is added, or when DropKey$^\star(B, A, k)$ is added. Suppose now $r_n\sigma_n$ adds DropKey$^\star(A, B, k)$, then:

$$\begin{aligned}
&\overset{KC}{\implies} \exists k' : \mathsf{msg} . && r_n\sigma_n \text{ adds } \text{AddKey}^\star(A, B, k') \\
&\overset{KP}{\implies} \exists i < n . && r_i\sigma_i \text{ adds } \text{AddKey}^\star(B, A, k') \\
&\overset{KU}{\implies} \nexists j . && r_j\sigma_j \text{ adds } \text{DropKey}^\star(A, B, k') \\
&\overset{KDR}{\implies} \nexists m . && r_m\sigma_m \text{ adds } \text{DropKey}^\star(B, A, k') \\
&\implies && \text{ShKeys}(B, A, \langle \ldots, k', \ldots \rangle) \in S
\end{aligned}$$

and so now k' is a common key after the rule application. Therefore the Common Key predicate is preserved.

Suppose instead $r_n\sigma_n$ adds DropKey$^\star(B, A, k)$, then:

$$\begin{aligned}
&\overset{KDR}{\implies} \exists i < n . && r_i\sigma_i \text{ adds } \text{DropKey}^\star(A, B, k) \\
&\overset{WF}{\implies} \exists j < i . && r_j\sigma_j \text{ adds } \text{AddKey}^\star(A, B, k) \\
&\overset{KU}{\implies} \nexists l \neq i . && r_l\sigma_i \text{ adds } \text{AddKey}^\star(A, B, k) \\
&\implies && \text{ShKeys}(A, B, \langle \ldots, k, \ldots \rangle) \notin S
\end{aligned}$$

and so k was not a common key before the rule application. Therefore since S contained some key k' that was a common key, so does the state after the rule application, and so the common key predicate is preserved. □

Theorem 4.8 provides a set of sufficient conditions to ensure that a protocol in our model satisfies desynchronisation resistance. We provide one example of a *necessary* condition to satisfy desynchronisation resistance: any protocol that fails to meet this condition also fails to provide resistance against desynchronisation attacks.

Theorem 4.12 (Necessity) *Let $P = (\Sigma, E, R, S^{start})$ be a protocol that satisfies Properties 4.2, 4.3, and 4.4. Let $S^0 \in S^{start}$ and $\mathrm{ShKeys}(A, B, k) \in S^0$ (i.e. A stores exactly one key for B) and assume P does not satisfy Key Preparedness (Definition 4.6) for A and B. Then P either contains no reachable key update rule applications for A, or it does not satisfy desynchronisation resistance.*

Proof. Suppose P contains at least one key update rule for A. We will construct a trace from which the $\mathrm{Complete}^\star(A, B)$ is no longer reachable without adversary interference.

Let $\tau = (S^0, r_1\sigma_1, \ldots, r_n\sigma_n)$ be a trace such that $r_n\sigma_n$ is a key update rule application for A that violates the Key Preparedness property. Consider the state $\mathit{strip}(\mathrm{lastState}(\tau))$. Note this state is reachable from $\mathrm{lastState}(\tau)$ through the rules Sh_Cancel and Block.

By Reachability Conditional on a Common Key (Property 4.2), there exist no traces starting from $\mathit{strip}(\mathrm{lastState}(\tau))$ that lead to the $\mathrm{Complete}^\star(A, B)$ event fact without adversary interference. Thus desynchronisation resistance is violated. □

5 Automated Verification

In this section we discuss the automated verification of the security properties from the previous section in the proving tool Tamarin. Tamarin uses multiset rewriting theory at its core, allowing for our model to be naturally implemented. We discuss the basic details of the implementation of the properties from Section 4 in Tamarin, before discussing two protocols that were analysed and shown to have attacks by using the Tamarin prover. In Appendix A we discuss some of the obstacles overcome in the implementation. The full implementations, along with diagrams and full descriptions of the attack traces can be found on our git repository [4], along with several other demonstrations of the security properties defined in this paper.

Definitions 4.5, 4.6, and 4.7 can be readily implemented in Tamarin. The remaining definitions used in our results can be verified syntactically from a protocol specification. With these considerations, our security properties can be analysed.

We note that the environment introduced in Section 4 is applicable to a large majority of key updating protocols. For example, many modern messaging applications make use of variations of the Diffie-Hellman Double Ratchet

[4] https://github.com/DesynchTamarin/desynch

algorithm, which satisfies Common-Key Reachability (Property 4.2), Key Conservation (Property 4.4), and Key Uniqueness (Property 4.5). Note that Well-Formedness is a consequence of the specification of the protocol, not the protocol itself. The Gossamer protocol in the RFID domain also satisfies these properties. As a consequence, the verification of these protocols is limited only by the power of the analysis tools involved.

5.1 Identified Attacks

Our analysis identified novel attacks in two papers from the domain of RFID grouping protocols. In particular, these protocols were shown to violate the conditions of Theorem 4.12.

A desynchronisation attack was found on the grouping protocol of Sundaresan, Doss, and Zhou [26]. The attack consists of a modified replay message, taking advantage of the algebraic properties of the exclusive-OR function, which is used to mask data. This replay causes an RFID tag to incorrectly authenticate the adversary as a valid reader, updating their key past a safe threshold. The intended execution of the protocol, and a trace which leads to a desynchronisation attack, can be found in Figure 2. A very similar attack can be found on another RFID grouping protocol, by Sundaresan, Doss, Piramuthu and Zhou [24].

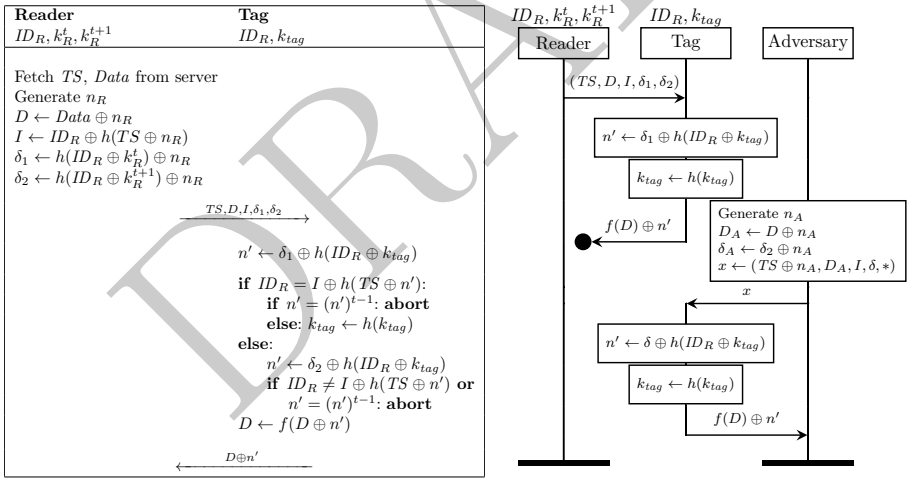

Fig. 2: The grouping protocol of Sundaresan et al. (left), and attack trace (right)

An attack was also found on the 'two-round grouping proof' of Abughazalah, Markantonakis and Mayes[1]. This protocol consists of a single message-response round which allows multiple tags to authenticate to a single RFID reader. However, a modified replay attack abuses a built-in measure that allows a tag to 'reset' its group key. In this instance, the adversary can launch countless replay messages, causing a tag to update its personal encryption key arbitrarily many times. Further information about the attack can be found in Appendix B.

6 Conclusion

Denial-of-Service attacks are often not considered in the analysis of security protocols, mainly because such attacks are hard to distinguish from regular omissions in the underlying communication channel. However, some types of DoS attacks are aimed at vulnerabilities at the protocol level. A typical example is formed by the class of desynchronisation attacks, which aim to disrupt all future communications between the protocol agents by desynchronising their communication keys.

Even though such desynchronisation attacks have been known for over a decade, formal analysis tools have been lacking. In this paper we have addressed this issue by developing a formal definition of desynchronisation resistance using a protocol model based on multiset rewriting. This definition has been operationalised by defining a set of sufficient conditions that can be easily validated by current state-of-the-art verification tools, such as Tamarin. We showed the applicability of our methodology by deriving two novel desynchronisation attacks on published RFID protocols.

References

1. Abughazalah, S., Markantonakis, K., Mayes, K.: Two rounds RFID grouping-proof protocol. In: 2016 IEEE International Conference on RFID, RFID 2016, Orlando, FL, USA, May 3-5, 2016. pp. 161–174 (2016)
2. Avoine, G.: Adversarial model for radio frequency identification. IACR Cryptology ePrint Archive 2005, 49 (2005)
3. Blanchet, B.: An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In: CSF'01. pp. 82–96 (2001)
4. Blanchet, B.: Using Horn clauses for analyzing security protocols. Formal Models and Techniques for Analyzing Security Protocols 5, 86–111 (2011)
5. Cohn-Gordon, K., Cremers, C., Garratt, L.: On post-compromise security. In: CSF'16. pp. 164–178. IEEE (2016)
6. van Deursen, T., Mauw, S., Radomirovic, S., Vullers, P.: Secure ownership and ownership transfer in RFID systems. In: ESORICS'09. pp. 637–654 (2009)
7. van Deursen, T., Radomirovic, S.: Attacks on RFID protocols. IACR Cryptology ePrint Archive 2008, 310 (2008)
8. Dolev, D., Yao, A.C.: On the security of public key protocols. IEEE Trans. Information Theory 29(2), 198–207 (1983)
9. Dreier, J., Duménil, C., Kremer, S., Sasse, R.: Beyond subterm-convergent equational theories in automated verification of stateful protocols. In: International Conference on Principles of Security and Trust. pp. 117–140. Springer (2017)
10. Durgin, N.A., Lincoln, P., Mitchell, J.C.: Multiset rewriting and the complexity of bounded security protocols. Journal of Computer Security 12(2), 247–311 (2004)
11. Goguen, J.A., Meseguer, J.: Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. Theor. Comput. Sci. 105(2), 217–273 (1992)
12. Juels, A.: "yoking-proofs" for RFID tags. In: 2nd IEEE Conference on Pervasive Computing and Communications Workshops (PerCom 2004 Workshops), 14-17 March 2004, Orlando, FL, USA. pp. 138–143 (2004)

13. Jung, S.W., Jung, S.: HRP: A HMAC-based RFID mutual authentication protocol using PUF. In: International Conference on Information Networking (ICOIN). pp. 578–582. IEEE (2013)
14. Kapoor, G., Piramuthu, S.: Vulnerabilities in some recently proposed RFID ownership transfer protocols. In: First International Conference on Networks & Communications. pp. 354–357. IEEE (2009)
15. Li, Q.S., Xu, X.L., Chen, Z.: PUF-based RFID ownership transfer protocol in an open environment. In: 15th International Conference on Parallel and Distributed Computing, Applications and Technologies. pp. 131–137. IEEE (2014)
16. Li, T., Wang, G.: Security analysis of two ultra-lightweight RFID authentication protocols. In: New Approaches for Security, Privacy and Trust in Complex Environments. pp. 109–120. Springer (2007)
17. Meier, S., Schmidt, B., Cremers, C., Basin, D.: The TAMARIN prover for the symbolic analysis of security protocols. In: International Conference on Computer Aided Verification. pp. 696–701. Springer (2013)
18. Peris-Lopez, P., Castro, J.C.H., Estévez-Tapiador, J.M., Ribagorda, A.: Advances in ultralightweight cryptography for low-cost RFID tags: Gossamer protocol. In: Information Security Applications, 9th International Workshop, WISA 2008, Jeju Island, Korea, September 23-25, 2008, Revised Selected Papers. pp. 56–68 (2008)
19. Perrin, T., Marlinspike, M.: The double ratchet algorithm. GitHub wiki (2016)
20. Radomirovic, S., Dashti, M.T.: Derailing attacks. In: Security Protocols XXIII - 23rd International Workshop, Cambridge, UK, March 31 - April 2, 2015, Revised Selected Papers. pp. 41–46 (2015)
21. Saito, J., Sakurai, K.: Grouping proof for RFID tags. In: 19th International Conference on Advanced Information Networking and Applications (AINA 2005), 28-30 March 2005, Taipei, Taiwan. pp. 621–624 (2005)
22. Srivastava, K., Awasthi, A.K., Kaul, S.D., Mittal, R.C.: A hash based mutual RFID tag authentication protocol in telecare medicine information system. J. Medical Systems 39(1), 153 (2015)
23. Sun, D., Zhong, J.: Cryptanalysis of a hash based mutual RFID tag authentication protocol. Wireless Personal Communications 91(3), 1085–1093 (2016)
24. Sundaresan, S., Doss, R., Piramuthu, S., Zhou, W.: A robust grouping proof protocol for RFID EPC C1G2 tags. IEEE Trans. Information Forensics and Security 9(6), 961–975 (2014)
25. Sundaresan, S., Doss, R., Zhou, W.: Secure ownership transfer in multi-tag/multi-owner passive RFID systems. In: Globecom 2013 - Symposium on selected areas in communications. pp. 2891–2896. IEEE (2013)
26. Sundaresan, S., Doss, R., Zhou, W.: Zero knowledge grouping proof protocol for RFID EPC C1G2 tags. IEEE Trans. Computers 64(10), 2994–3008 (2015)
27. The Tamarin Team: MS Windows NT Kernel Description (2018), https://tamarin-prover.github.io/manual/tex/tamarin-manual.pdf

A Tamarin Implementation Details

In Tamarin, executions always begin from the empty trace. The adversary knowledge is assumed to contain all public terms (such as the names of agents). To model this, we add a set of additional rules describing the establishment of shared keys between agents, as well as corruption rules where agents reveal their secret

information to the adversary. Such rules are commonplace, and are comparable to those found in the Tamarin User Manual [27].

Tamarin allows for the implementation of user-defined equational theories. However, it requires that they be *subterm convergent*. We note that progress has been made on implementing more permissive equational theories, such as the work by Dreier et al., which provides an extension allowing for *AC-convergent* [9] equational theories.

Because of this, in some cases we are required to under-approximate the equational theory of a protocol. The most notable example of this is with the exclusive-or (XOR) operator. This under-approximation means that any identified attack traces are still valid, but it is possible that Tamarin will incorrectly report that a property holds. This is a limitation of the implementation, not of the model itself.

Tamarin supports unbounded analysis, using induction arguments to successfully limit the search space in its backwards search approach. However, for stateful protocols, it is not uncommon for Tamarin to require assistance in finding proofs, sometimes failing to terminate. This means that at times we have aided the tool by manually identifying minor 'helper' lemmas which identify the key induction steps needed.

For ease of readability, we have assumed that the participants of a protocol can be assigned to *roles*. For example, in the RFID case, an agent may be a *tag* or a *reader*. As such, the event fact AddKey* is divided into the two event facts TagAddsKey* and ReaderAddsKey*.

B Attack on the two-round grouping proof of Abughazalah, Markantonakis and Mayes

Abughazalah, Markantonakis and Mayes provide a two-round RFID grouping proof protocol [1], which uses updating keys. An RFID tag stores two updating keys, for authenticating itself as well as identifying the group that it is a part of.

A system is in place to allow a tag to re-synchronise its group key if it is absent for a run of the protocol, and does not receive the needed message to cause it to update its key naturally. However, this system allows for replay attacks to cause a tag to desynchronise its personal key with that stored by the verifier.

The analysis of the protocol in Tamarin revealed that it fails to satisfy the conditions of Theorem 4.12, resulting in an attack.

Protocol Description The protocol is described in detail in the original paper. Here, we provide a simplified description of the protocol for the sake of conciseness. For example, the attack involves communication only between the reader and a single tag, so we focus on only looking at one tag. We also adopt the slightly adapted notation from Table 1. A diagram of the intended execution of the protocol is provided in Figure 3.

Table 1: Notation used in the protocol by Abughazalah et al.

ID_G	The identity of the reader, a secret value
ID_T	The identity of the tag, a secret value
k_G	A secret key for the group being tested
k_T	A secret key for the specific tag being tested
TS^t	An encrypted timestamp, used in construction of the proof
n_R	A fresh (random) nonce generated by the reader
n_T	A fresh (random) nonce generated by the tag
$h(\cdot)$	A cryptographic hash function

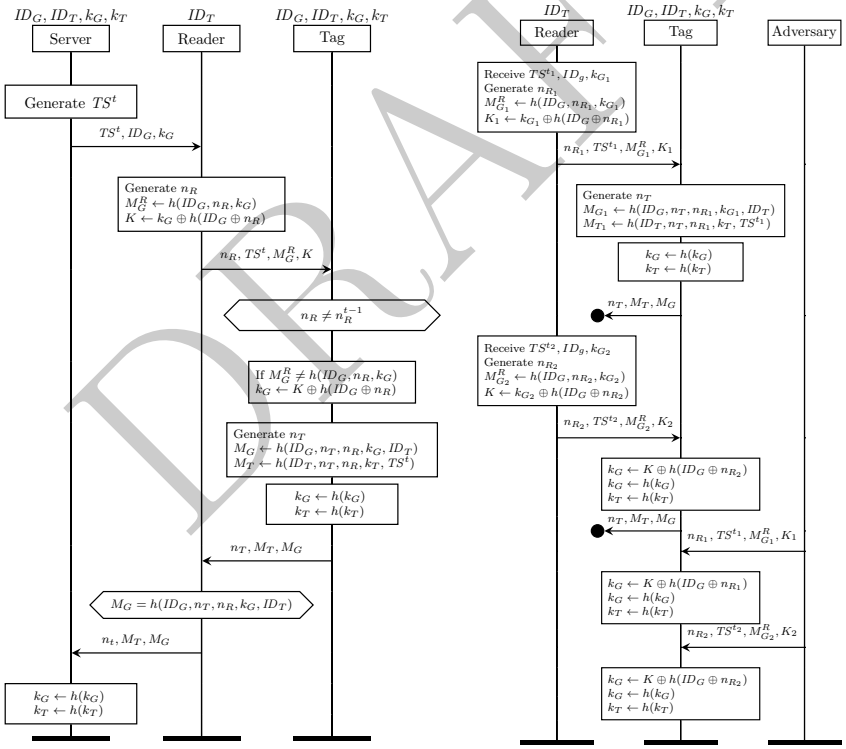

Fig. 3: Two-rounds grouping proof protocol (left) and attack trace (right).

Attack Trace Description The grouping protocol has the advantage of requiring only two exchanged messages during its main execution between the reader and tag. However, this results in a vulnerability which leads to a desynchronisation attacks. The protocol was analysed in Tamarin, with the server role merged into the reader role. This is because the reader and server are assumed to have a secure communications channel.

Note that blocking the tag's message to the reader during a run of the protocol leads to a situation where the tag updates its secret, but the reader will not. The next time the protocol runs, the tag will receive the first message from the reader. Regardless of whether the reader updated the group key k_G (which it may have, because of the presence of other tags in the group completing the protocol), the tag will authenticate to this message and update its key a further time.

The authors seem aware of this problem, and suggest that it is possible for the server to calculate future values of the tag's key in order to prevent desynchronisation. However, there exists the capability to perform replay former messages, causing the tag to update its personal key arbitrarily many times.

As mentioned in the protocol paper, each tag stores previous nonces that they successfully authenticated to. However, an RFID tag has limited memory capacity - a typical EPC Generation 2 tag (such as those mentioned in the paper) has around 512 bits of storage space, meaning that there is very little space to store previously received nonces.

As such, if an adversary is able to eavesdrop at least two runs of the protocol, a tag will readily accept a replay of a message from a previous execution. At this point, the tag will update its key. The adversary can then replay a different message, and repeat this cycle as long as desired.