Looping for Good: Cyclic Proofs for Security Protocols

Felix Linker
Department of Computer Science, ETH Zurich
Zurich, Switzerland
flinker@inf.ethz.ch

Christoph Sprenger
Department of Computer Science, ETH Zurich
Zurich, Switzerland
sprenger@inf.ethz.ch

Cas Cremers
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
cremers@cispa.de

David Basin
Department of Computer Science, ETH Zurich
Zurich, Switzerland
basin@inf.ethz.ch

Abstract

Security protocols often involve loops, such as for ratcheting or for manipulating inductively-defined data structures. However, the automated analysis of security protocols has struggled to keep up with these features. The state-of-the-art often necessitates working with abstractions of such data structures or relies heavily on auxiliary, user-defined lemmas.

In this work, we advance the state-of-the-art in symbolic protocol verification by adapting cyclic induction proof systems to the security protocol domain. We introduce reasoning rules for the Tamarin prover for cyclic proofs, enabling new, compact proofs, and we prove their soundness. Moreover, we implement new, simple, and effective proof search strategies that leverage these rules. With these additions, Tamarin can prove many lemmas that previously required, often complex, auxiliary lemmas. We showcase our approach on fourteen case studies, ranging from toy examples to a detailed model of the Signal protocol. Our work opens an exciting new research area where automatic induction helps scale security protocol verification, as we provide a fundamentally new and general induction mechanism.

CCS Concepts

• Theory of computation → Verification by model checking; **Logic and verification**; **Automated reasoning**; **• Security and privacy → Formal security models**; **Logic and verification**; **Security protocols**.

Keywords

Security protocol verification, Formal methods, Cyclic induction, Tamarin prover

ACM Reference Format:
Felix Linker, Christoph Sprenger, Cas Cremers, and David Basin. 2025. Looping for Good: Cyclic Proofs for Security Protocols. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765131

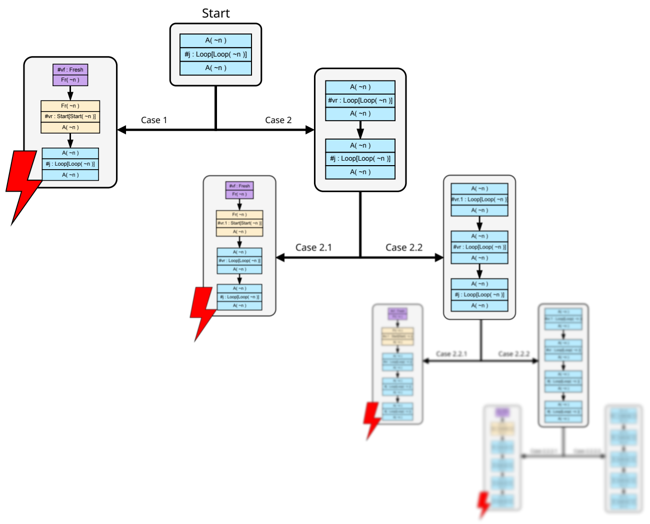

Figure 1: Example of non-termination in Tamarin. Without using trace induction, Tamarin can only apply one proof method, which results in a case split. Left-side case leads to a contradiction (↯) whereas right-side case recurses infinitely.

1 Introduction

Recent years have seen substantial progress in security protocol verification. Numerous detailed models of real-world protocols have been constructed and verified using state-of-the-art protocol verification tools such as Tamarin and ProVerif, including TLS 1.3 [24, 25, 8], Noise [29], 5G [40, 3], EMV [5], Signal [32] and variants [23], and Apple's iMessage [34].

Despite these advances, scaling automated verification to large case studies remains challenging. One of the main challenges is that modern security protocols often use complex loop structures, for example, for message transmission, key renewal, and ratcheting in secure messaging. These structures introduce sources of non-termination for protocol verification. Proof techniques such as induction and the use of auxiliary lemmas, which are supported by Tamarin [38] and ProVerif [11], can help reduce the proof search space and thus alleviate some of the non-termination problems. However, stating inductive conjectures and useful auxiliary lemmas requires the users' ingenuity and expertise, which hampers proof automation. In this work, we improve upon the state-of-the-art in proof automation for modern protocol designs by tackling these non-termination issues.

Tamarin uses constraint systems to represent the set of executions violating a given property. Constraint reduction essentially corresponds to a backwards search from an attack state to an initial state. By the reduction rules' soundness and completeness, reaching an initial state corresponds to an attack, while reducing all constraint systems to contradictions corresponds to a proof. Non-termination manifests itself as a never-ending backwards attack construction. When explored in Tamarin's interactive mode, these constructions usually exhibit a repeating behavioral pattern, such as the one shown on the right-hand branch of Figure 1. In this work, we formalize when such repeating patterns contradict the executions' well-foundedness. For that, we use the proof method of *infinite descent* that was already used by Euclid, but first formalized by Fermat [42]. They applied it, for instance, to prove the non-existence of solutions to number-theoretic problems.

Modern accounts of deductive reasoning formalize such arguments using *cyclic proofs*, e.g., [13, 17]. Deductive reasoning rules decompose a proof goal into sub-goals, for example, proving $P \wedge Q$ can be reduced to proving P and proving Q. Applying these rules induces a proof tree, whose leaves are given by axioms, e.g., $A \vee \neg A$. Cyclic proofs allow one to prove a statement $P(\sigma(x))$ that instantiates a more general statement $P(x)$ encountered earlier in the proof by introducing a *backlink* from $P(\sigma(x))$ to $P(x)$. In this way, cyclic proofs generalize proof trees to proof graphs. Moreover, an additional *global soundness condition* ensures that all cycles correspond to well-founded inductive arguments. This form of inductive reasoning does not use any explicit induction rules.

Cyclic proofs' advantage over traditional inductive proofs is that inductive reasoning in general requires a priori ingenuity to choose an inductive conjecture, induction scheme, induction variables, and the position in a proof to apply induction. These choices, sometimes even called "eureka steps" to underscore their difficulty, make the automation of inductive proofs notoriously hard [18, 19]. In contrast, in cyclic proofs, these choices are made *implicitly* and *a posteriori*: The inductive statement and the position where to apply induction arise by the formation of backlinks, whereas the induction scheme and the induction variables are determined by the global soundness condition. These properties of cyclic induction can both simplify proofs and offer the promising potential for proof automation, even though, in general, undecidability still calls for user-provided inductive statements [18, 19]. Given its advantages over traditional induction, cyclic induction has been applied in many areas of logic and computer science, including first-order logic with inductive predicates [13], equational reasoning about functional programs [31], program logics [44, 51], program synthesis [30], and μ-calculi [46, 49].

In this work, we formalize and implement cyclic proofs for security protocol verification in Tamarin, we prove its soundness, and we show that it substantially improves proof automation. Doing this required overcoming several challenges, both in terms of the logical formalism and proof search.

First, Tamarin uses constraint solving rather than a form of sequent calculus, as commonly used for cyclic proofs. It was initially unclear whether cyclic proofs could be adapted to this setting. Besides formulas describing protocol properties, Tamarin's constraint systems also include complex *graph constraints*, which describe execution steps and their dependencies. The interpretation of constraint systems as sequents that derive contradictions from the given constraints clarified the connection to existing cyclic proof systems and facilitated their adaption to our setting. Moreover, Tamarin's backward search for an initial state suggested the set of timepoints labeling execution steps as the natural well-founded domain for cyclic induction.

Second, we find that effective backlink formation in cyclic proofs requires two structural rules from sequent calculi, weakening and cut, which we add to Tamarin's constraint reduction rules. *Weakening* generalizes a constraint system by discarding some constraints and *cut* introduces a new constraint and its negation in separate cases. To restrict the search space and achieve a high degree of automation, we use the structural rules in a strictly controlled way: our use of weakening is based on the protocol's looping structure, and we use cut only with formulas that are either (i) instances of formulas that already exist in the constraint system or (ii) ordering constraints that preserve some of the weakened information.

Finally, as the backlink search is NP-complete, we devise an effective heuristic to perform backlink checks incrementally and discard impossible cases early.

We evaluate cyclic proofs and compare them with proofs by explicit trace induction. Our results show that our controlled use of the structural rules together with only minimal (and generic) changes to Tamarin's existing proof search strategies are sufficient to prove many lemmas with no or substantially fewer auxiliary lemmas than needed with trace induction.

Contributions. Our contributions are three-fold. First, we introduce cyclic proofs for protocol verification. In particular, we extend Tamarin's constraint reduction rules with structural rules for weakening and cut, define cyclic proofs for Tamarin, and prove their soundness. Second, we implement this proof system in Tamarin, which requires a major overhaul of Tamarin's internal structure. We also provide a set of heuristics to guide effective proof search in the cyclic proof system, including the controlled application of the structural rules. Third, we evaluate our approach on fourteen case studies ranging from simple protocols to a detailed model of the Signal protocol. We show that our implementation effectively reduces the number of auxiliary lemmas required. In particular, we require no auxiliary lemmas to prove message secrecy for Signal. Although our work is based on Tamarin, the ideas are, in principle, transferable to other tools such as ProVerif.

Our work opens an exciting new area in which automatic induction helps scale protocol verification. Whereas current automatic tools only provide limited support for induction, we provide a fundamentally new and general induction mechanism. A wide variety of protocols will benefit from this. For example, protocols with nested loops, such as those using double-ratchets, protocols with arbitrarily many participants, such as consensus protocols, and protocols involving participant groups, such as those employing threshold cryptography.

2 Background on Tamarin

Tamarin [39] is a model checker for security protocol verification, which incorporates a constraint solving algorithm based on symbolic backwards search. In this section, we briefly introduce Tamarin's underlying verification theory. For more details see [38].

2.1 Protocol Specifications

Tamarin works in the *symbolic model*, where protocol messages are represented as elements of a term algebra. An *equational theory E* encodes the semantics of these terms. Typically, E models cryptographic operations and their relationships. For example, the equation `sdec(senc(m, k), k) = m` defines the symbolic model of symmetric encryption, formalizing that decrypting an encryption with the same key yields the original message. Along with the equational theory, the protocol and attacker behavior are specified by a set of multiset rewriting rules R. A Tamarin protocol model then is a pair (R, E).

2.1.1 *Multiset Rewriting Rules.* In Tamarin, a global state consists of a multiset of ground facts (facts may repeat). Similar to predicates in first-order logic, *facts* are built by applying fact symbols to terms. A fact is *ground* if its terms have no variables. Facts typically model the local state of participants or the network, e.g., who possesses which keys, or what messages have been sent out on the network.

Multiset rewriting rules in Tamarin have the form `l --a-> r`, where `l`, `a`, and `r` are all multisets of facts. `l` encodes the facts that are required to apply the rule (the *premises*), `r` encodes the effects of a rule (the *conclusions*), and `a` labels the rule application for subsequent reference in lemmas with *action facts*. Facts are either persistent (preceded by a `!`) or linear. When a rule is applied, the linear facts in `l` are removed from the global state and replaced by the facts in `r`. For example, the following rule models sending a nonce, encrypted using a symmetric key, and has no action facts as labels: `[ Fr(n), !Key(k) ] --> [ Out(senc(n, k)) ]`.

The persistent fact `!Key` models storing a long-term key. `Fr` and `Out` are reserved, linear facts. `Fr` models generating a random, unguessable value, and `Fr` facts are provided by the built-in rule Fresh. `Out` models sending a message over an insecure network. Tamarin provides built-in support for an active network adversary that can intercept, reroute, replay, modify, and insert any message derivable from their knowledge. Utilizing the equational theory and dedicated adversary rules, the adversary can reason about all messages sent. For example, consider the rule above. If `k` leaked to the adversary, they could learn n by constructing the term `sdec(senc(n, k), k)`, which reduces to the plaintext `n` (using the equation above). The adversary can construct this term because they know `k` by assumption and the ciphertext because it was previously sent.

Example 1 (Running example). *Consider the following model.*

```
rule Start: [Fr(x)] --[Start(x)]-> [A(x)]
rule Loop:  [A(x)]  --[Loop(x) ]-> [A(x)]
rule Stop:  [A(x)]  --[Stop(x) ]-> []
```

This model consists of three rules, `Start`, `Loop`, *and* `Stop`, *which model a loop that starts with the generation of a fresh nonce, is executed a non-deterministic number of times, and may eventually stop.*

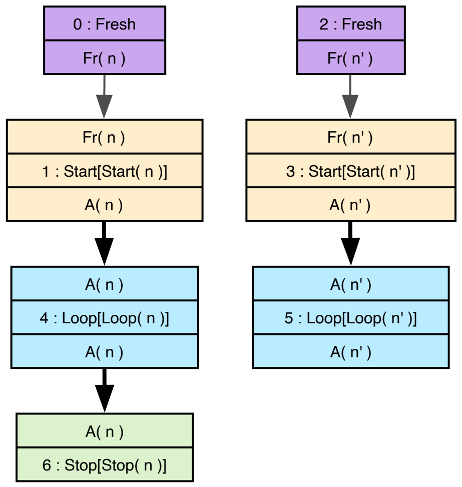

Figure 2: Dependency graph.

2.2 Semantics

2.2.1 *Executions and Traces.* A Tamarin model's multiset rewriting rules induce a labeled transition system. An *execution* is a sequence of labeled state transitions, starting in the (empty) initial state. Each execution has a corresponding *trace* consisting of the sequence of sets of action facts labeling each transition. The semantics of a Tamarin model is its set of traces.

2.2.2 *Dependency Graphs.* Dependency graphs capture the sequence of rule applications and dependencies between rule premises and conclusions. They are closely related to the constraint systems used in Tamarin's protocol analysis and they enable effective proof search (Sections 2.4 and 2.5). Figure 2 shows a dependency graph corresponding to two runs of the Loop example from Example 1 with a single loop iteration, one of them completed with the Stop rule. Here, n and n' are fresh values. Every node corresponds to a ground rule instance with three parts. The upper part are the rule's premises, the lower part are the rule's conclusions, and the middle part are the rule's *timepoint*, name, and action facts. When a rule has no premises or conclusions, these parts are omitted. The timepoints reflect at which point in time a rule was applied. Note that the two Loop example runs are interleaved.

Definition 1 (Dependency graph). *A* dependency graph *is a tuple $dg = (I, D)$ where I is a finite list of nodes, each of which corresponds to a ground instance of a multiset rewriting rule from the underlying Tamarin model, indexed by the set $idx(I) = \{0, \ldots, |I|-1\}$, and $D \subseteq \mathbb{N}^2 \times \mathbb{N}^2$. We write $(i, u) \rightarrowtail (j, v)$ for $((i, u), (j, v)) \in D$, when D is clear from the context. Here, (i, u) denotes rule i's (in I) conclusion u and (j, v) rule j's premise v. We require that edges correctly connect premises and conclusions, i.e., $i < j$ and the conclusion (i, u) is equal (modulo E) to the premise (j, v). Furthermore, each premise must have exactly one incoming edge, every linear conclusion has at most one outgoing edge, and* Fresh *rule instances are unique.*

To illustrate Tamarin's edge notation, consider the top-right edge in Figure 2. It connects the first premise and conclusion of rule instances 2 and 3 and is thus $(2, 0) \rightarrowtail (3, 0)$. The trace induced by a dependency graph is the sequence of the action labels of the rules in I. One can show that the dependency graphs of a model (R, E) induce the same set of traces as the set of traces derived from the model's executions ([38, Theorem 3]). The trace of the dependency graph shown in Figure 2 is:

$$\left[\emptyset, \{\mathsf{Start}(n)\}, \emptyset, \{\mathsf{Start}(n')\}, \{\mathsf{Loop}(n)\}, \{\mathsf{Loop}(n')\}, \{\mathsf{Stop}(n)\}\right].$$

2.3 Protocol Properties

In Tamarin, protocol properties are expressed as first-order logic *trace properties*, which are given by closed *trace formulas*. Atomic trace formulas are false $\bot$, action formulas $f@i$, and predicates $i \doteq j$ (timepoint equality), $i < j$ (timepoint ordering), and $t = u$ (term equality), where f is an action fact, i and j are temporal variables representing timepoints, and t and u are terms. Trace formulas can be combined with the usual logical connectives $\neg$, $\wedge$, $\vee$, and $\Longrightarrow$ as well as universally and existentially quantified. We write $fv(\varphi)$ for (all) free variables and $fv_{tmp}(\varphi)$ for the free temporal variables of a formula φ. We assume that all trace formulas are *guarded*. This means that all existentially quantified formulas are of the form $\exists\vec{x}.\, f@i \wedge \varphi$ and all universally quantified formulas are of the form $\forall\vec{x}.\, f@i \Longrightarrow \varphi$, where $\mathrm{set}(\vec{x}) \subseteq fv(f@i)$, i.e., all bound variables are free in the action formula $f@i$. Intuitively speaking, guardedness requires that all instantiations of quantified variables are related to a protocol execution (and its trace) via some action fact $f@i$. We write $\widehat{\varphi}$ for $\neg\varphi$'s negation normal form.

Trace properties to be proved in Tamarin are called *lemmas*. For example, the following property expresses that every $\mathsf{Stop}(x)$ action is preceded by a $\mathsf{Start}(x)$ action:

$$\forall x, j.\, \mathsf{Stop}(x)@j \Longrightarrow \exists i.\, \mathsf{Start}(x)@i \wedge i < j. \qquad (1)$$

Trace formulas are interpreted over traces. Given a valuation θ of the free variables of φ, we write $(tr, \theta) \vDash_E \varphi$ to mean that the trace tr satisfies φ under the valuation θ, which respectively maps temporal and term variables to timepoints and terms. We do not formally define the satisfaction relation here, but appeal to intuition (see [38] for details). A protocol model (R, E) satisfies a universal trace property φ, written $R \vDash_E \varphi$, if all traces of (R, E) satisfy φ. Tamarin also supports existential trace lemmas to show the existence of a trace with a given property. Henceforth, we only consider universal trace properties without further mentioning it, but cyclic induction can be applied to existential trace lemmas too.

2.4 Constraint Systems

To verify that a universal trace property φ holds for a model (R, E), Tamarin negates the property and searches for an attack, i.e., a dependency graph satisfying the negated property. If none is found, this is a proof that the property holds.

2.4.1 *Syntax.*

Tamarin uses constraint systems to symbolically describe sets of dependency graphs, i.e., potential attacks. Tamarin uses five types of constraints:

(1) Formula φ: Trace formula φ must hold.
(2) Node $i : ri$: Rule ri is applied in node i.
(3) Edge $(i, u) \rightarrowtail (j, v)$: The fact at index u of node i's conclusion is used for node j's premise at index v.

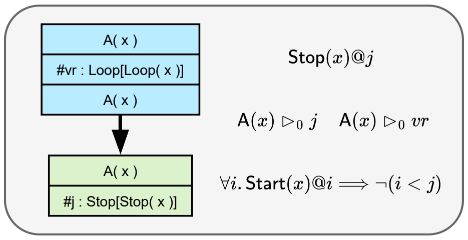
A(x)
#vr : Loop[Loop(x)]
A(x)
A(x)
#j : Stop[Stop(x)]
$\mathsf{Stop}(x)@j$
$\mathsf{A}(x) \rhd_0 j \quad \mathsf{A}(x) \rhd_0 vr$
$\forall i.\, \mathsf{Start}(x)@i \Longrightarrow \neg(i < j)$

Figure 3: Example constraint system.

(4) Premise $f \rhd_v i$: Node i has fact f as its premise number v. This constraint is used to generate new node constraints and to connect them to this premise using edge constraints. We call unsolved premise constraints *open premises*.
(5) Chain $(i, u) \dashrightarrow (j, v)$: The adversary learned the term at node j's premise v by deconstructing the term at node i's conclusion u, possibly transitively. This constraint reasons about which messages an adversary may have learned specific terms from.

A *constraint system* Γ is a set of constraints over a given protocol model (R, E). Let $\mathbb{C}$ be the set of all constraint systems defined over the given protocol. We extend the functions $fv(\cdot)$ and $fv_{tmp}(\cdot)$ from formulas to constraint systems as expected.

2.4.2 *Semantics.*

A *model* for a constraint system Γ is a pair (dg, θ) consisting of a dependency graph dg and a valuation θ of the free variables of Γ. We write $(dg, \theta) \Vdash_E \Gamma$ if the model satisfies all constraints in Γ. A dependency graph dg is a *solution* for Γ if there is a valuation θ such that $(dg, \theta) \Vdash_E \Gamma$. We write $sols(\Gamma)$ for the set of all solutions of Γ. Again, we will not define $\Vdash_E$ in detail (see [38]), but appeal to intuition.

Example 2. *Figure 3 shows a constraint system with two node, one edge, two premise, and two formula constraints. The constraint system is satisfied by the dependency graph in Figure 2 under the valuation* $\theta = [vr \mapsto 4, j \mapsto 6, x \mapsto n]$.

Observe that we defined two logical relations for trace formulas. We write $(tr, \theta) \vDash_E \varphi$ if a trace tr satisfies φ, and $(dg, \theta) \Vdash_E \varphi$ if a *dependency graph* dg satisfies φ. There is a close connection between $\vDash_E$ and $\Vdash_E$, namely, for any protocol model (R, E) and trace formula φ, there exists a trace tr and valuation θ such that $(tr, \theta) \vDash_E \varphi$ if and only if there exists a dependency graph dg and valuation θ such that $(dg, \theta) \Vdash_E \{\varphi\}$, i.e., $sols(\{\varphi\})$ is non-empty. This reduces the formula satisfaction problem to a constraint solving problem.

2.4.3 *Timepoints and Temporal Order.*

Each constraint system Γ induces a *temporal order* $\prec_\Gamma$ on its temporal variables, which is defined as the minimal transitive relation such that $i \prec_\Gamma j$ if

$$i < j \in \Gamma \;\vee\; \exists u, v.\, (i, u) \rightarrowtail (j, v) \in \Gamma \;\vee\; \exists u, v.\, (i, u) \dashrightarrow (j, v) \in \Gamma.$$

We write $\preceq_\Gamma$ for the reflexive closure of $\prec_\Gamma$. Note that timepoints in dependency graphs are linearly ordered (as they are a subset of $\mathbb{N}$), whereas a constraint system's temporal order only partially orders temporal variables, reflecting that the execution order of certain rules may be irrelevant (e.g., two parallel protocol runs can be arbitrarily interleaved). They are related as follows.

Lemma 1 ([38, Lemma 7]). *Suppose $(dg, \theta) \Vdash_E \Gamma$. Then (i) if $i \lessdot_\Gamma j$ then $\theta(i) < \theta(j)$ and (ii) if $i \leq_\Gamma j$ then $\theta(i) \leq \theta(j)$.*

2.5 Constraint Reduction and Derivation Trees

We are now prepared to explain the idea behind Tamarin's proof methodology. To prove that $R \vDash_E \varphi$ holds for a protocol model (R, E), Tamarin uses a tree-search, constraint-solving algorithm. Every node in the search tree corresponds to a constraint system. Initially, the tree only contains the root constraint system $\{\widehat{\varphi}\}$, which transforms proving the validity of φ into proving the unsatisfiability of $\neg\varphi$, i.e., into an attack search problem.

2.5.1 Constraint Reduction Rules. Constraint reduction rules refine the search tree's leaf constraint systems and are of the form

$$\Gamma \rightsquigarrow \{\Gamma_1, \ldots, \Gamma_n\}\,. \tag{2}$$

Such a rule encodes that the constraint system Γ can be solved by solving the constraint systems $\Gamma_1, \ldots, \Gamma_n$. We will often write concrete reduction rules as $\Gamma \rightsquigarrow \Gamma_1 \parallel \ldots \parallel \Gamma_n$, where the cases are easier to distinguish. Given constraint systems Γ and Δ, we write (Γ, Δ) for $\Gamma \cup \Delta$ and use constraints as singleton constraint systems when clear from context, for example φ stands for $\{\varphi\}$.

To apply a constraint reduction rule to a constraint system, the rule's free variables must be instantiated with a substitution so that the constraint reduction rule matches the constraint system. We therefore close the relation $\rightsquigarrow$ under substitutions, that is, $\Gamma\theta \rightsquigarrow \{\Gamma_1\theta, \ldots, \Gamma_n\theta\}$ for all substitutions θ and rules (2).

There are different types of constraint reduction rules. For instance, there are rules that (i) work on formula connectives, similar to the deduction rules of first-order logic, (ii) introduce new node constraints from $f@i$ constraints, (iii) backwards-complete premise constraints $f \triangleright_v i$, by introducing new nodes and edges connected to open premises, (iv) enforce the single use of linear facts, or (v) derive contradictions, written as $\Gamma \rightsquigarrow \bot$.

2.5.2 Derivation Trees and Proofs. When a constraint reduction rule (2) is applied to a leaf Γ of the search tree, the constraint systems Γ_1 to Γ_n become that node's children. We formally define Tamarin's search tree as a *derivation tree*.

Definition 2 (Derivation Trees). *A derivation tree $\mathcal{D} = (\mathcal{N}, \mathcal{E}, \gamma)$ for a property φ is a tree $(\mathcal{N}, \mathcal{E})$ with nodes $\mathcal{N}$ and edges $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ together with a function $\gamma : \mathcal{N} \to \mathbb{C}$ labeling each node with a constraint system such that*

- *$\gamma(v_0) = \{\widehat{\varphi}\}$ for the root $v_0 \in \mathcal{N}$ of $\mathcal{D}$, and*
- *for each non-leaf node v, there is a constraint reduction rule with $\gamma(v) \rightsquigarrow \{\gamma(v') \mid (v, v') \in \mathcal{E}\}$.*

A leaf v is called an axiom leaf, *if $\gamma(v) = \bot$ (i.e., $\gamma(v)$ is contradictory) and an* open leaf *otherwise.*

Tamarin's constraint reduction algorithm terminates when it finds a *proof*, i.e., all leaves are axioms, or an *attack*, i.e., there exists a leaf constraint system that is *solved*. For theories that use the subterm operator [26] and reducible function symbols such as xor, it can also terminate with *unfinishable*, i.e., the result is unknown. Solved constraint systems Γ are sufficiently constrained to enable the extraction of a model $(dg, \theta) \Vdash_E \Gamma$ [38, Theorem 5]. In particular, such a constraint system contains no node constraints with an open premise, which allows the extraction of a dependency graph.

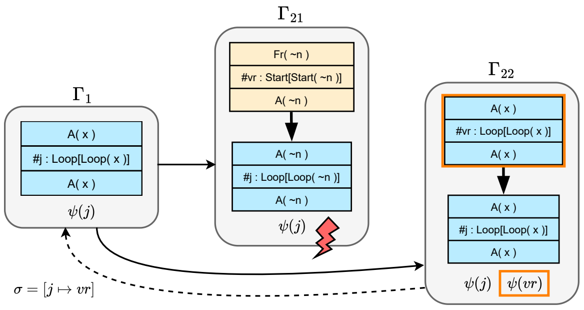

Figure 4: Proof of φ_1 using cyclic induction

A constraint reduction rule of the form (2) is *sound* if no solutions are lost, i.e., $sols(\Gamma) \subseteq \bigcup_i sols(\Gamma_i)$, and *complete* if no solutions are added, i.e., $sols(\Gamma) \supseteq \bigcup_i sols(\Gamma_i)$.[1] Since Tamarin's constraint reduction rules are both sound and complete, a proof of φ implies $R \vDash_E \varphi$ whereas a reduction sequence leading to a solved form implies $(dg, \theta) \Vdash_E \{\widehat{\varphi}\}$ for some dependency graph dg and valuation θ.

3 Overview of Cyclic Induction for Tamarin

Consider the Loop model from Example 1. One property we may wish to prove is that any Loop fact is preceded by a Start fact:

$$\varphi_1 = \forall j, x.\ \mathsf{Loop}(x)@j \implies \exists i.\ \mathsf{Start}(x)@i \wedge i < j.$$

In the following, we first consider a failed attempt at proving this property without induction. Afterwards we explain trace induction, and finally we introduce cyclic induction in Tamarin.

3.1 Proof Attempt without Induction

When attempting to prove φ_1, Tamarin first computes the negation normal form $\widehat{\varphi_1}$. After some simplifications, it introduces a node constraint containing the $\mathsf{Loop}(x)@j$ action fact and assumes no preceding Start fact exists (see Γ_1 in Figure 4):

$$\psi(j) = \forall i.\ \mathsf{Start}(x)@i \implies \neg(i < j).$$

Next, Tamarin solves the Loop node's open premise $\mathsf{A}(x)$. There are two cases, denoted by Γ_{21} and Γ_{22} in Figure 4, where the formula $\psi(vr)$ in Γ_{22} should be ignored for the moment. The constraint system Γ_{21} introduces a Start node, which immediately contradicts the formula $\psi(j)$. The system Γ_{22} introduces another Loop node. Without using induction, Tamarin enters an infinite loop from here on, repeatedly solving open $\mathsf{A}(x)$ premises, as in Figure 1.

3.2 Proof Using Trace Induction

Tamarin is already capable of proving the property φ_1 using trace induction. Trace induction can only be applied at the proof's start, and it splits the formula φ to be proven into two cases: (1) the base case formula $BC(\varphi)$, which holds if φ holds on the empty trace, and the step formula $IH(\varphi) \implies \varphi$, where the induction hypothesis $IH(\varphi)$ states that φ holds on all but a trace's last timepoint.

[1]We use the proof-theoretic definitions of these terms here, whereas in [38] the former relation is called completeness and the latter is called correctness.

In our example, the base case immediately leads to a contradiction. The step case results in a derivation tree similar to the one depicted in Figure 4. In all constraint systems, the induction hypothesis $IH(\varphi_1)$ is available:

$$\forall j, x.\ \mathsf{Loop}(x)@j \land \neg\mathsf{last}(j) \implies \exists i.\ \mathsf{Start}(x)@i \land \neg\mathsf{last}(i) \land i < j.$$

The presence of the Start node in Γ_{21} still immediately contradicts the formula $\psi(j)$. In the Loop case Γ_{22}, we can now apply the induction hypothesis to the new Loop node at vr, since it is not the last node. This creates a $\mathsf{Start}(x)@i$ fact with $i < vr < j$, which contradicts the formula $\psi(j)$.

3.3 Proof Using Cyclic Induction

In a proof based on cyclic induction, there is no explicit induction rule or induction hypothesis. Instead, one uses the ordinary constraint reduction rules and tries to detect loops such as the one in Figure 1. One folds the associated infinite proof tree into a finite tree with *backlinks* and then tries to prove that the resulting cyclic structure represents *well-founded* (non-circular!) reasoning. In this section, we provide some intuition for our cyclic proof system, which we subsequently formalize in Section 4.

3.3.1 Backlink Formation. We can detect loops by discovering constraint systems that are *subsumed* by more general ones appearing earlier in the proof. A constraint system Γ *subsumes* a leaf constraint system Γ' when $\Gamma\sigma \subseteq \Gamma'$ for some substitution σ. In this case, we can introduce a backlink from Γ' to Γ, thereby creating a *pre-proof graph* instead of a tree.

We now construct a cyclic proof of φ_1 for the Loop example. Looking at the system Γ_{22} in Figure 4, we see that the Loop node at vr repeats the initial Loop node at j in Γ_1. More precisely, the substitution $\sigma = [j \mapsto vr]$ maps the former node to the latter one. However, we do not have $\Gamma_1\sigma \subseteq \Gamma_{22}$ as $\sigma(\psi(j)) = \psi(vr)$ is missing from Γ_{22}. This motivates the introduction of a *cut* constraint reduction rule. Using this rule, we can add $\psi(vr)$ to the constraint system Γ_{22}, provided we can show in a separate case that adding its negation leads to a contradiction. Indeed, $\neg\psi(vr)$ contradicts $\psi(j)$ because $vr \prec_{\Gamma_{22}} j$. After cutting in $\psi(vr)$, we have $\Gamma_1\sigma \subseteq \Gamma_{22}$ and can thus add a backlink from Γ_{22} to Γ_1 (dashed arrow in Figure 4). This results in a (cyclic) pre-proof of the property φ_1.

3.3.2 Well-founded Reasoning. To avoid unsound circular reasoning, we must ensure that all cycles in a pre-proof correspond to well-founded inductive arguments. Suppose we have constructed a pre-proof for φ, but $\widehat{\varphi}$ is satisfiable, i.e., $(dg, \theta) \Vdash_E \widehat{\varphi}$ for some dg and θ. Since Tamarin's constraint reduction rules are sound, every satisfiable constraint system in the pre-proof is either solved or has a satisfiable child constraint system.

We build a path $\pi = n_0 n_1 \cdots$ as follows. Let n_0 be $\widehat{\varphi}$ (satisfiable) and n_{i+1} be one of the satisfiable children of n_i, as long as n_i has children in the pre-proof. Suppose π were finite. Then, its last constraint system must be a leaf in the pre-proof and thus cannot be satisfiable because all leafs in the pre-proof must be contradictory axiom constraint systems. Thus π must be infinite.

To establish that an infinite path π corresponds to well-founded reasoning, we require that every cycle in a pre-proof "progresses." Progress ensures that repeating patterns captured by backlinks occur at smaller and smaller timepoints along an infinite path,

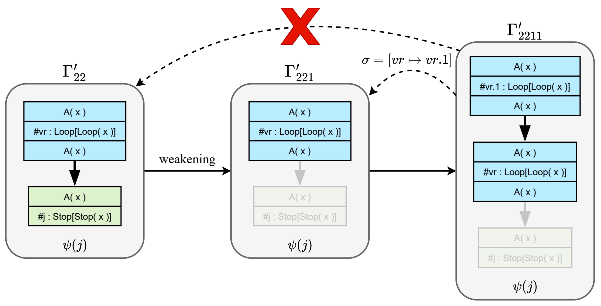

Figure 5: Weakening in cyclic induction proof of φ_2

which contradicts the well-foundedness of timepoints. This follows the intuition that repeatedly solving the same premises, creating increasingly larger constraint systems, does not lead to a finite attack.

More precisely, we say that a backlink from Γ' to Γ where $\Gamma\sigma \subseteq \Gamma'$ *progresses on* j when there is a temporal variable j such that $\sigma(j) \prec_{\Gamma'} j$ (as is the case in our example above). We ensure progress by requiring that every backlink progresses on at least one temporal variable j. If we were to traverse a backlink infinitely often, then j would decrease infinitely often. However, as temporal variables are interpreted in $\mathbb{N}$, this is impossible.

Additionally, we must ensure that backlinks are mutually compatible if they are part of the same strongly connected subgraph (SCS). It could happen that two backlinks ℓ_1 and ℓ_2 "destroy" each other's progress, for example, if each of them increases the variable on which the other progresses. To prevent this, we will define a notion of *preservation*, which we use to define a *discharge condition* for pre-proofs and an algorithm to check it.

3.3.3 Explicit Weakening. We now consider a second property, stating that every Stop fact is preceded by a Start fact:

$$\varphi_2 = \forall j, x.\ \mathsf{Stop}(x)@j \implies \exists i.\ \mathsf{Start}(x)@i \land i < j.$$

Trying to prove φ_2 by trace induction results in non-termination, but one can find a proof when using φ_1 as an auxiliary lemma.

We next show how to prove this property with cyclic induction but without any auxiliary lemmas. The initial node constraint is a Stop node at j and subsequently solving that node's premise $\mathsf{A}(x)$ produces two constraint systems, one for Start and one for Loop. The former leads to an immediate contradiction as before. The latter is depicted as Γ'_{22} in Figure 5. This figure only illustrates the backlink in the cyclic proof of φ_2 and omits all non-looping branches. Without any other steps, solving the premise $\mathsf{A}(x)$ again would lead to a constraint system similar to Γ'_{2211}, including the faded-out node. There is no backlink from this constraint system to Γ'_{22} because, to ensure progress, we must map vr to $vr.1$. However, if we do that, we require an edge from $vr.1$ to j in Γ'_{2211}, which is missing.

We solve this problem by simply "deleting" the Stop node at j. This is called *weakening* and is a well-established and sound proof rule for generalization; we define a *weakening rule* in Section 4. Without the Stop node, there is a backlink from Γ'_{2211} to Γ'_{221}, as illustrated in Figure 5. Note that, in contrast to the previous example,

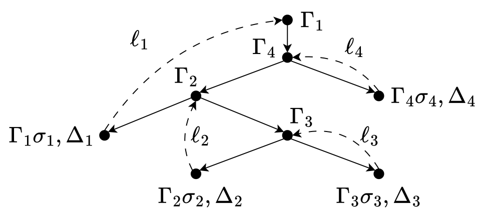
Γ_1
Γ_4
ℓ_1
ℓ_4
Γ_2
$\Gamma_4\sigma_4, \Delta_4$
Γ_3
$\Gamma_1\sigma_1, \Delta_1$
ℓ_2
ℓ_3
$\Gamma_2\sigma_2, \Delta_2$
$\Gamma_3\sigma_3, \Delta_3$

Figure 6: A pre-proof $(\mathcal{D}, \mathcal{L}, \Sigma)$. Solid arrows are edges in $\mathcal{E}$ and dashed arrows are backlinks in $\mathcal{L}$.

there is no need to use the cut rule here because the timepoint j in the formula $\psi(j)$ is not substituted by σ.

4 Cyclic Induction for Tamarin

4.1 Structural Constraint Reduction Rules

We extend Tamarin's constraint reduction rules with two additional rules: *weakening* and *cut*, which are well-known structural rules in logic. To maintain important invariants of Tamarin's constraint solving algorithm, we use restricted versions of these rules, but omit these rather technical restrictions for clarity (see [36]).

The cut rule $\mathcal{S}_\Delta$ introduces a new set of formulas Δ and proves each formula's validity in separate cases. The weakening rule $\mathcal{S}_W$ simply removes some constraints.

$$\mathcal{S}_\Delta : \Gamma \rightsquigarrow (\Gamma, \Delta) \parallel \big\|_{\varphi\in\Delta}(\Gamma, \widehat{\varphi}) \qquad \mathcal{S}_W : \Gamma, \Delta \rightsquigarrow \Gamma$$

Clearly, $\mathcal{S}_\Delta$ is sound and complete and $\mathcal{S}_W$ is sound. However, $\mathcal{S}_W$ is incomplete, since $sols(\Gamma \cup \Delta) \subseteq sols(\Gamma)$. Hence, the models found after applying weakening may not constitute attacks.

4.2 Pre-Proofs

To introduce cyclic reasoning in Tamarin, we augment derivation trees (Definition 2) with *backlinks* that introduce cycles. We add a backlink from a leaf Γ' to an inner node Γ on the path from the root to Γ' whenever Γ is a more general constraint system than Γ', i.e., $\Gamma\sigma \subseteq \Gamma'$ for some substitution σ. To ensure sound reasoning, we must ensure that every cycle "progresses," which prevents the recurring pattern from repeating indefinitely. We formalize this intuition in Section 4.3 where we define when pre-proofs become full proofs.

Definition 3 (Pre-Proof). *A* pre-proof $\mathcal{P} = (\mathcal{D}, \mathcal{L}, \Sigma)$ *of a property φ consists of a derivation tree $\mathcal{D} = (\mathcal{N}, \mathcal{E}, \gamma)$ for φ, a* partial backlink function $\mathcal{L} : \mathcal{N} \rightharpoonup \mathcal{N}$, *and a* partial substitution function $\Sigma : \mathcal{N} \rightharpoonup \mathbb{S}$, *such that the domain of $\mathcal{L}$ and Σ is the set of open leaves of $\mathcal{D}$ and, for each open leaf v with $\sigma_v = \Sigma(v)$, the node $w = \mathcal{L}(v)$ lies on the path from $\mathcal{D}$'s root node v_0 to v and $\sigma_v(\gamma(w)) \subseteq \gamma(v)$.*

We will often write σ_v for $\Sigma(v)$ and we also use $\mathcal{L}$ to denote the function's graph, which consists of backlinks.

Definition 4 (Pre-proof graph). *The* pre-proof graph *of* $\mathcal{P} = (\mathcal{D}, \mathcal{L}, \Sigma)$ *is* $\mathcal{G}(\mathcal{P}) = (\mathcal{N}, \mathcal{E} \cup \mathcal{L}, \gamma)$. *For a backlink $\ell = (v, w) \in \mathcal{L}$, we denote the path from w to v in $\mathcal{D}$ by $\pi(\ell)$.*

Figure 6 gives an example of a pre-proof. Every leaf of the derivation tree either has an associated backlink or yields a contradiction.

A strongly connected subgraph (SCS) of $\mathcal{G}(\mathcal{P})$ is a subgraph of $\mathcal{G}(\mathcal{P})$ where there is a path from every node to every other node. A strongly connected component (SCC) is a maximal strongly connected subgraph. We henceforth only consider SCSs and SCCs that contain at least one edge without explicitly mentioning it.

Note that any SCS S in $\mathcal{G}(\mathcal{P})$ is characterized by the set $\mathcal{L}_S$ of backlinks it contains: S contains the nodes and edges of the paths $\pi(\ell)$ for some $\ell \in \mathcal{L}_S$ and the backlinks in $\mathcal{L}_S$. We say that a set $L \subseteq \mathcal{L}$ *induces* an SCS S if $L = \mathcal{L}_S$.

4.3 Cyclic Proofs and their Soundness

We next show when pre-proofs are sound, i.e., when a cyclic pre-proof for a property φ is a cyclic proof that indeed implies that φ holds. Following the intuition presented in Section 3.3.2, the critical part of a sound definition of (cyclic) proofs is defining suitable notions of progress and preservation that ensure that the inductive reasoning embodied in the pre-proof is well-founded.

Definition 5 (Progress and preservation). *Given a pre-proof $\mathcal{P} = (\mathcal{D}, \mathcal{L}, \Sigma)$ and a temporal variable $i \in \mathcal{V}_{tmp}$, a backlink $\ell = (v, w) \in \mathcal{L}$* preserves i *if i occurs free in all constraint systems labeling the nodes on $\pi(\ell)$ and $\sigma_v(i) \preceq_{\gamma(v)} i$, and* progresses on i, *if ℓ preserves i and $\sigma_v(i) \prec_{\gamma(v)} i$.*

Definition 6 (Proof). *A pre-proof $\mathcal{P} = (\mathcal{D}, \mathcal{L}, \Sigma)$ of a property φ is a* proof *of φ if it satisfies the following* discharge condition*: for each strongly connected subgraph S of $\mathcal{G}(\mathcal{P})$, there is a temporal variable $i \in \mathcal{V}_{tmp}$ such that some backlink $\ell \in \mathcal{L}_S$ progresses on i, and all backlinks $\ell' \in \mathcal{L}_S$ preserve i.*

We next state auxiliary lemmas for our soundness theorem. Proofs of Lemma 2, 3, and Theorem 1 are given in [36]. Point (iii) of Lemma 2 is needed, since timepoints are in general interpreted in $\mathbb{Q}$, which is not well-founded. However, timepoints related to action and node constraints are always interpreted in $\mathbb{N}$.

Lemma 2 (Local soundness). *Tamarin's constraint reduction rules are sound. In particular, given a rule $\Gamma \rightsquigarrow \{\Gamma_1, \ldots, \Gamma_n\}$ and a model (dg, θ) satisfying Γ, there is valuation θ' such that (i) (dg, θ') satisfies some Γ_k, (ii) θ agrees with θ' on all free variables common to Γ and Γ_k, and (iii) the property that all free temporal variables are valuated in $\mathbb{N}$ is preserved, i.e., if $\theta(fv_{tmp}(\Gamma)) \subseteq \mathbb{N}$ then $\theta'(fv_{tmp}(\Gamma_k)) \subseteq \mathbb{N}$.*

Lemma 3. *Let $(v, w) \in \mathcal{L}$ be a backlink and (dg, θ) a model satisfying $\gamma(v)$. Then $(dg, \theta \circ \sigma)$ satisfies $\gamma(w)$.*

Theorem 1 (Soundness). *Let $\mathcal{P}$ be a proof for a given protocol model (R, E) and a guarded trace property φ. Then $R \vDash_E \varphi$.*

Proof (sketch). Suppose for a contradiction that $\hat{\varphi}$ is (R, E)-satisfiable. Then there is a model $(dg, \theta_0) \Vdash_E \hat{\varphi}$. Using Lemmas 2 and 3, we construct an infinite sequence $\{(v_k, \theta_k)\}_{k\in\mathbb{N}}$ of pairs of nodes and valuations such that $\pi = \{v_k\}_{k\in\mathbb{N}}$ is an infinite path in $\mathcal{G}(\mathcal{P})$ and, for all $k \in \mathbb{N}$, $(dg, \theta_k) \Vdash_E \gamma(v_k)$ and, for all temporal variables $i \in fv(\gamma(v_k))$, $\theta_k(i) \in \mathbb{N}$. By the definition of proofs and Lemma 1, there is a temporal variable i such that the sequence $\{\theta_k(i)\}_{k\in\mathbb{N}}$ of timepoints eventually decreases infinitely often. This contradicts the well-foundedness of the natural numbers. □

Algorithm 1 Returns a discharging progress order $(\mathcal{L}, \sqsubseteq, \iota)$ for a pre-proof $\mathcal{P} = (\mathcal{D}, \mathcal{L}, \Sigma)$ if one exists and fails otherwise.

```
1:  let $C$ be a partitioning of $\mathcal{L}$ into sets inducing $\mathcal{G}(\mathcal{P})$'s SCCs
2:  return progress_order $\mathcal{P}$ $(\mathrm{Id}_{\mathcal{L}})$ $\emptyset$ $C$
3:
4:  function progress_order $\mathcal{P}$ $(\sqsubseteq)$ $\iota$ $C$ =
5:  if $\mathrm{dom}(\iota) = \mathcal{L}$ then
6:      return $(\mathcal{L}, \sqsubseteq, \iota)$
7:  else
8:      let $L \in C$ and $\ell \in L$ and $i \in \mathcal{V}_{tmp}$ such that
9:          $\ell$ progresses on $i$ and all $\ell' \in L$ preserve $i$
10:     if these do not exist then
11:         return failure
12:     else
13:         let $\sqsubseteq' = \sqsubseteq \cup \{(\ell', \ell) \mid \ell' \in L\}$
14:         let $C_L$ be the partitioning of $L \setminus \{\ell\}$ into subsets
15:             that induce the SCCs of $(\mathcal{N}, \mathcal{E} \cup L \setminus \{\ell\})$
16:         let $C' = (C \setminus \{L\}) \cup C_L$
17:         return progress_order $\mathcal{P}$ $(\sqsubseteq')$ $(\iota[\ell \mapsto i])$ $C'$
18:     end if
19: end if
```

4.4 Alternative Discharge Condition

We give an alternative condition for pre-proofs that can be easily implemented as an algorithm to check whether a pre-proof is a proof.

4.4.1 Progress orders. This condition is based on progress orders [45, 48][2], which organize backlinks into a partial order, labeled with temporal variables. A progress order that satisfies our alternative discharge condition ensures that there exists a temporal variable decreasing infinitely often along every infinite path. The order identifies such a variable as the greatest element for every SCS and, in turn, for every infinite path. We show that this alternative discharge condition is equivalent to the original one in Definition 6.

Definition 7 (Progress order). *Let $\mathcal{P} = (\mathcal{D}, \mathcal{L}, \Sigma)$ be a pre-proof. A* progress order *$(\mathcal{L}, \sqsubseteq, \iota)$ for $\mathcal{P}$ is a partial order $(\mathcal{L}, \sqsubseteq)$ on the set of backlinks and a labeling function $\iota\colon \mathcal{L} \to \mathcal{V}_{tmp}$ assigning to each backlink ℓ a temporal variable $\iota(\ell)$ such that, for every SCS S of $\mathcal{G}(\mathcal{P})$, $\mathcal{L}_S$ has a $\sqsubseteq$-greatest element.*

Definition 8 (Alternative discharge condition). *A progress order $(\mathcal{L}, \sqsubseteq, \iota)$ for a pre-proof $\mathcal{P} = (\mathcal{D}, \mathcal{L}, \Sigma)$ discharges $\mathcal{P}$ if for all $\ell \in \mathcal{L}$: ℓ progresses on $\iota(\ell)$, and ℓ preserves $\iota(\ell')$, for all $\ell \sqsubseteq \ell'$.*

Note that every total order on backlinks is a progress order and for total orders the discharge condition corresponds to a lexicographic order on the temporal variables associated to the backlinks.

4.4.2 Algorithm for Checking Discharge Condition. We give an algorithm that, given a pre-proof $\mathcal{P} = (\mathcal{D}, \mathcal{L}, \Sigma)$, computes a discharging progress order $(\mathcal{L}, \sqsubseteq, \iota)$ if one exists and fails otherwise (Algorithm 1). The algorithm first determines a partitioning C of the set of backlinks $\mathcal{L}$ into sets inducing $\mathcal{G}(\mathcal{P})$'s SCCs (line 1) and then calls the recursive function *progress_order* (line 2). Besides the (fixed) pre-proof $\mathcal{P}$, this function has three parameters: the current ordering $\sqsubseteq$ on $\mathcal{L}$, the current labeling of backlinks with temporal variables ι, and a set C of subsets of $\mathcal{L}$, which partitions the set $\mathcal{L} \setminus \mathrm{dom}(\iota)$ into the subsets that induce the SCCs of the graph $(\mathcal{N}, \mathcal{E} \cup \mathcal{L} \setminus \mathrm{dom}(\iota))$. Initially, the relation $\sqsubseteq$ is the identity relation $\mathrm{Id}_{\mathcal{L}}$ on $\mathcal{L}$ and the labeling ι (and hence $\mathrm{dom}(\iota)$) is empty. Note that, since the initial partial order is flat, all elements of $\mathcal{L}$ are minimal.

[2] [45, 48] use the term "induction orders", which we avoid for clarity.

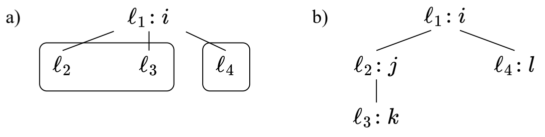

Figure 7: Progress order construction for Figure 6's pre-proof.

Each recursion adds some $\ell \in \bigcup C$ to $\mathrm{dom}(\iota)$. The idea is that progress and preservation for all SCSs S of $\mathcal{G}(\mathcal{P})$ containing backlinks in $\mathrm{dom}(\iota)$ is already covered (in the sense of Definition 6), while these properties remain to be shown for the remaining SCSs, each of which is induced by (a subset of) some $L \in C$.

The algorithm terminates and returns $(\mathcal{L}, \sqsubseteq, \iota)$ when the labeling ι covers all of $\mathcal{L}$ (line 6). Otherwise, it determines a backlink ℓ in one of the sets $L \in C$ and an associated variable on which ℓ progresses and which all elements in L preserve (lines 8 and 9). These exist if $\mathcal{P}$ is a proof. Otherwise the algorithm fails (line 11). The backlink ℓ then becomes the greatest element of L (line 13). The partition C is updated by removing the set L from C and replacing it by the sets in the partitioning of $L \setminus \{\ell\}$ into subsets inducing the SCCs of the graph $(\mathcal{N}, \mathcal{E} \cup L \setminus \{\ell\})$ (lines 14-16). The mapping $[\ell \mapsto i]$ is then added to ι and the function *progress_order* is recursively called with the updated parameters (line 17).

Example 3 (Computing a progress order). *Figure 6's pre-proof has four backlinks $\mathcal{L} = \{\ell_1, \ell_2, \ell_3, \ell_4\}$. Suppose the backlinks progress on ($\ell^<$) or preserve ($\ell^\leq$) the temporal variables i, j, k, and l as follows:*

$$\begin{array}{llll} \ell_1^< = \{i\} & \ell_2^< = \{j\} & \ell_3^< = \{k\} & \ell_4^< = \{l\} \\ \ell_1^\leq = \{i\} & \ell_2^\leq = \{i, j\} & \ell_3^\leq = \{i, j, k\} & \ell_4^\leq = \{i, l\} \end{array}$$

We now use the function progress_order (Algorithm 1) to compute a progress order as follows. Note that $\mathcal{L}$ induces the single SCC of the pre-proof graph. We start with $\sqsubseteq_0 = \mathrm{Id}_{\mathcal{L}}$, the identity relation on $\mathcal{L}$, ι_0 the empty labeling, and $C_0 = \{\mathcal{L}\}$. Observing that ℓ_1 progresses on i and all other backlinks preserve i, we place ℓ_1 above the other backlinks in $\sqsubseteq_1$ and set $\iota_1 = \iota_0[\ell_1 \mapsto i]$ as in Figure 7a. The updated set C_1 then partitions the set of remaining backlinks $\{\ell_2, \ell_3, \ell_4\}$ into the two sets $\{\ell_2, \ell_3\}$ and $\{\ell_4\}$, inducing the remaining SCCs (i.e., ignoring ℓ_1's backlink in the pre-proof graph). Next, we see that ℓ_2 progresses on j, which ℓ_3 preserves. Hence, we place ℓ_2 on top of ℓ_3 in $\sqsubseteq_2$ and set $\iota_2 = \iota_1[\ell_2 \mapsto j]$. Removing ℓ_2's backlink in turn, C_2 contains the sets $\{\ell_3\}$ and $\{\ell_4\}$, inducing the only remaining SCCs. As ℓ_3 progresses on k and ℓ_4 progresses on l, we add these as mappings to the final labeling ι in two additional recursive calls of progress_order.

The resulting progress order is depicted in Figure 7b. Note that, since ℓ_4 does not preserve j and k, requiring a linear progress order as a discharge condition would be too strong for this pre-proof.

The alternative discharge condition is equivalent to the original one from Definition 6, as stated in the following proposition, which we constructively prove based on Algorithm 1 in [36].

Proposition 1 (Equivalence of discharge conditions). *A pre-proof $\mathcal{P} = (\mathcal{D}, \mathcal{L}, \Sigma)$ is a proof if and only if there exists a progress order $(\mathcal{L}, \sqsubseteq, \iota)$ for $\mathcal{P}$.*

5 Implementation

Our implementation of cyclic proofs required a major overhaul of Tamarin. We changed around 200 files, inserted 61,000 lines of code, and deleted 24,000 lines of code. In this section, we describe this implementation, which is available at [35].

When Tamarin searches for proofs, it constructs a proof tree by repeatedly executing *proof methods*, which typically apply sequences of constraint reduction rules. At each node of the proof tree, Tamarin can choose one from the many available proof methods to apply, and heuristics guide which proof methods will be applied automatically by Tamarin. In this section, we describe new proof methods and heuristics for cyclic induction, and how our implementation verifies that constructed pre-proofs are valid.

5.1 Backlink Search

5.1.1 Cyclic Proof Methods. We implemented three new proof methods: *search backlink*, *cut*, and *minimize for cyclic proofs*, the last of which combines weakening and cut. When used naively, cut and weakening considerably enlarge Tamarin's search space. The set of formulas that could be cut in is infinite, and theoretically, every single constraint could be weakened. Thus, we must restrict the application of these proof methods.

When applying the proof method *search backlink* to a constraint system Γ, our implementation searches for a substitution such that some constraint system on the path from Γ to the root subsumes Γ. Although we implemented several optimizations to the backlink search (see Section 5.1.2), backlink search is an instance of the subgraph isomorphism problem, which is NP-complete [22]. Thus, we cannot exclude instances where backlink search is computationally expensive. We therefore implemented the backlink search as a proof method so that users or heuristics can avoid backlink search when it is too expensive.

When implementing the backlink search, we observed cases where we expected a backlink, but where "trivial" formula constraints were missing in the leaf constraint system. Whenever a backlink search finds such a "close" match, our implementation suggests cutting in missing constraints. Noteworthy, this is the only way for our implementation to suggest the *cut* proof method.

Finally, *minimize for cyclic proofs* applies weakening combined with cut to restore ordering information lost by weakening edge constraints. Weakening can be required to find backlinks (see Section 3.3.3). The proof method weakens all nodes that occur after the start of a loop and iteratively weakens nodes that provide premises of already weakened nodes. This proof method implements a simple weakening heuristic, which we discuss further in Section 6.2.2.

Implementing automated weakening has multiple benefits: (1) It enables finding some backlinks in the first place. (2) The resulting constraint systems are smaller, lowering the cost of backlink search. (3) Weakening can lead to termination rather than non-termination when no proof is found. As weakened constraints systems are smaller, they are more likely to become solved. Tamarin will report "unfinishable" for such constraint systems as they may not be counterexamples. This informs the user that automated methods to tackle loops failed. Without cyclic induction, failing to tackle a loop typically results in non-termination.

5.1.2 Backlink Search Algorithm. We optimized our implementation of backlink search between two constraint systems by exploiting that substitutions must progress. To ensure progress, a substitution associated to a backlink must map at least some nodes to nodes that occur earlier in the constraint system. We generalize this observation and implement backlink search as a *DAG-prefix* search. DAG nodes are node constraints and $f@i$ formula constraints. DAG edges are edge constraints and $i < j$ formula constraints. These DAGs capture many important properties of constraint systems and provide structure to guide the backlink search. The DAG-prefix search attempts to match the smallest (w.r.t. to the edge-relation) nodes in the smaller DAG to the smallest nodes in the larger DAG and iteratively refines the resulting substitution by attempting to match the children of already matched nodes with one another. Should the substitution at some point match the two DAGs, we check whether it also applies to remaining constraints.

To speed up the DAG-prefix search, we color the DAG nodes in such a way that (a) we can check in constant time whether two nodes have the same color, and (b) two nodes can be matched only if they have the same color. Concretely, we color nodes annotated with rule instances using their rule name and nodes not yet annotated with rule instances with the list of action facts present at that node.

Technically, this search is incomplete. For example, it might be necessary to map a smallest node in the one graph to some node in the middle of the other graph. However, as constraint systems are constructed incrementally, it is likely that we considered such mappings earlier during proof search and need not consider them again.

5.2 Proof Search

5.2.1 Heuristics. We have adapted Tamarin's heuristics that rank proof methods during proof search. Tamarin supports two main heuristics: a general-purpose "smart" heuristic and an "injective" heuristic that is tailored for theories that heavily use loops. We amended both of these heuristics as follows.

Tamarin recognizes some proof methods as *looping* in that after applying them, the same proof method is typically available again. To avoid immediate non-termination, such proof methods will be applied in a round-robin fashion. We modified Tamarin's heuristics such that proof methods related to cyclic proofs are recognized as looping and prioritized like other looping proof methods.

The other heuristic we implemented is that Tamarin will search backlinks before minimizing for cyclic proofs. This ensures that weakening does not preclude backlink formation. Beyond that, we implemented no further heuristics. Thus, cyclic proofs can successfully be implemented with little guidance, as will become clear when we present our case studies in Section 6.

5.2.2 Discharging Pre-Proofs. So far, we showed how we implemented proof methods for finding backlinks, and how we rank these proof methods to find cyclic pre-proofs. The final step of a cyclic proof is to check whether the pre-proof discharges, i.e., whether Definition 8 applies. Implementing Algorithm 1 directly proved to be challenging as Tamarin's code-base is recursion-oriented. We thus slightly modified Algorithm 1 as follows.

The proof search in Tamarin recursively associates nodes with *results*, which can be: the constraint system is (i) unfinishable, (ii) contradictory, (iii) has a backlink, or (iv) a solution was found. Tamarin successively applies proof methods until it encounters constraint systems for which it can directly decide their result. These constraint systems become the pre-proof's leaves. Tamarin determines the inner nodes' results by combining their children's results. For example, the results solved and contradictory are combined to solved.

We modified Tamarin's proof search such that it recursively checks whether the pre-proof discharges. Our implementation inverts Algorithm 1. Observe that Algorithm 1 non-deterministically splits a set of SCCs into increasingly smaller SCSes to create a progress order. For every SCS, Algorithm 1 checks that it discharges. Instead of decomposing SCCs into SCSes, we compose SCSes to SCCs. Our implementation initialy stores each backlink in a singleton SCS, checks that this SCS discharges, and returns it. It then recursively composes SCSes into larger SCSs, eventually becoming SCCs, while maintaining the invariant that each SCS discharges. If at any point we fail to combine SCSes into a discharging SCS or SCC, we mark the proof as unfinishable.

There might be progress orders found by Algorithm 1 that our implementation does not find because our implementation traverses the pre-proof's SCSes following its tree structure, but Algorithm 1 searches non-deterministically. However, we never encountered proofs for which our implementation failed to find a progress order.

6 Case Studies and Discussion

We evaluated cyclic proofs on fourteen case studies, which include two models of the Signal protocol. Our evaluation shows that cyclic induction consistently reduces the number of auxiliary lemmas required to prove a conjecture compared to trace induction. In fact, cyclic induction typically requires *no* auxiliary lemmas. Many properties that previously required auxiliary lemmas, can now be proven without them. In particular for Signal, we observe that cyclic induction reduces model complexity, requiring fewer annotations, and simplifies proof search. There are only five exceptions. For two lemmas, cyclic induction requires the same auxiliary lemmas as trace induction. For one lemma, cyclic induction requires some, but fewer, auxiliary lemmas than trace induction. For another lemma, cyclic induction requires a different auxiliary lemma than trace induction. Finally, for one lemma, we were unable to construct a proof when using cyclic induction; however, we specifically engineered this lemma to be unprovable with cyclic induction. We discuss this further under limitations in Section 6.2.

We divide our case studies into three sets. The first set contains models that originate from Tamarin's standard examples and those of [34]. These models were developed to exhibit particular challenges of real-world looping protocols. The second set contains two models of Signal. The third set contains what we call *destructor-based theories*. We find that destructor-based theories are not well-suited for cyclic induction proofs and discuss this further in our limitations section.

We provide an overview of all case studies in Table 1. We analyzed the exact same theories and lemmas per induction scheme, i.e., theories were not modified for specific schemes. Table 1's first column shows to which set a theory belongs. Every set contains multiple theories (second column), which contain multiple lemmas (third column). We number lemmas for clarity. The columns grouped by "Provable…" show whether a given lemma is provable with: no induction ("w/o I"), trace induction ("w/ TI"), or cyclic induction ("w/ CI"). ✓ means that the lemma is provable. (X) means that the lemma is provable with the given scheme when using the referenced lemmas from the same theory as the auxiliary lemmas. For example, the lemma "Secrecy" from the "Crypto API" theory requires the lemma "Invariant" as an auxiliary lemma (cf. Table 1). Some lemmas are provable without induction but require inductive, auxiliary lemmas. As such lemmas are also provable with induction, we show the auxiliary lemmas required to prove the conjecture with induction in gray. No symbol means that we were unable to find a proof with the respective scheme. In the two columns named "Auto," we mark whether Tamarin was able to automatically construct a proof with its built-in heuristics using (i) no induction or trace induction or (ii) cyclic induction. The two columns grouped by "Using…" show whether the proof methods cut ($\mathcal{S}_\Delta$) or minimize for cyclic proofs ($\mathcal{S}_W$) were used in a cyclic proof. "User" in the column for $\mathcal{S}_W$ marks that user-specified weakening was required (see Section 6.2.2).

6.1 The Signal Case Study

Signal is the most widely used, end-to-end encrypted messaging protocol. It is so complex that previous attempts to prove its security in the symbolic model either drastically abstracted the protocol or even developed entirely new tools to tackle it (see Section 7.2). Signal uses the double-ratchet and X3DH key agreement protocols to achieve strong secrecy guarantees [41, 37]. The double-ratchet is a nested loop that establishes new shared secrets using Diffie-Hellman key exchange in an outer loop, and derives symmetric encryption keys from these shared secrets in an inner loop.

To evaluate cyclic induction, we proved message secrecy for two models of Signal (1 and 2). In our models of Signal, we assume that clients use authentic long-term and pre-key material for session establishment. Clients exchange messages over an insecure network, and they derive encryption keys using the double-ratchet and X3DH [41, 37]. We fully modelled the looping behavior of the double-ratchet, without abstracting it in any way. However, we did not model any features of Signal beyond the double-ratchet and X3DH, and, in particular, did not model skipped messages. The two models differ in that Signal 1 only allows long-term key reveal, whereas Signal 2 also allows revealing pre-keys and ephemeral keys.

We formalize message secrecy for Signal as follows:

$$\begin{aligned}
&\forall m, a, b, t.\,\mathsf{Send}(m, a, b)@t \\
\implies &\neg\exists x.\mathsf{K}(m)@x \\
&\vee \exists x.\mathsf{LtkReveal}(a)@x \vee \exists x.\mathsf{LtkReveal}(b)@x \\
&[\vee \exists x.\mathsf{RevealPre}(a)@x \vee \exists x.\mathsf{RevealPre}(b)@x \\
&\vee \exists x.\mathsf{RevealEph}(a, b)@x \vee \exists x.\mathsf{RevealEph}(b, a)@x].
\end{aligned}$$

Set	**Theory**	**Lemma**	**Provable…**					**Using…**	
			w/o I	**w/ TI**	**…auto**	**w/ CI**	**…auto**	$\mathcal{S}_\Delta$	$\mathcal{S}_W$
Set 1	Loop	(1) Start before Loop		✓	✓	✓	✓	✓	
		(2) Start before Stop	(1)	(1)	✓	✓	✓		✓
		(3) Loop before Stop		✓	✓	✓	✓		
		(4) Stop unique	(3)	(3)	✓	✓			✓
	Hash Chain	(1) Loop Start		✓	✓	✓	✓	✓	
		(2) Loop First		✓	✓	✓	✓		
		(3) Loop Success Ord		✓	✓	✓	✓		
		(4) Loop+Success inv.		✓	✓	✓		✓	✓+ User
		(5) Loop+Success	(3)-(4)	(3)-(4)	✓	(4)		✓	User
		(6) Success inv.		✓	✓	✓	✓		✓
		(7) Success	(6)	(6)	✓	✓	✓		✓
	Crypto API	(1) Invariant		✓	✓	✓			
		(2) Secrecy	(1)	(1)	✓	✓			✓
	Key Renegotiation	Secrecy		✓	✓	✓			
	Create, Use, Destroy	(1) Use		✓	✓	✓	✓		✓
		(2) Destroy		(1)	✓	✓		✓	✓
	Alternating Loop	(1) Outer Loop Step		✓	✓	✓	✓	✓	
		(2) Fresh Seed		(1)	✓	✓	✓		✓
		(3) Seed Secrecy		(1)	✓	✓	✓		
		(4) Key Secrecy		✓	✓	✓	✓	✓	
	Nested Loop	(1) Outer Loop Step		✓	✓	✓	✓	✓	
		(2) Fresh Seed		(1)	✓	✓	✓		✓
		(3) Seed Construction		(1)	✓	✓	✓		✓
		(4) Key Construction		✓	✓	✓	✓		
		(5) Key Secrecy	(1)-(4)	(1)-(4)	✓	✓	✓		✓
	Revealing Loop	(1) Loop Start		✓	✓	✓	✓		
		(2) Seeds Match		✓	✓	✓	✓		
		(3) IDs Match		✓	✓	✓	✓		
		(4) Secrecy	(1)-(2)	(1)-(2)	✓	(3)		✓	
		(5) Secrecy Variant	(1)-(2)	(1)-(2)	✓	✓		✓	
Set 2	Signal 1	(1) Outer Loop		✓	✓	✓	✓	✓	✓
		(2) Rk Secrecy		(1)		✓			✓
		(3) Ck Secrecy		✓	✓	✓	✓		
		(4) Secrecy	(2)-(3)	(2)-(3)	✓	✓			✓
	Signal 2	(1) Outer Loop		✓	✓	✓	✓	✓	✓
		(2) Rk Secrecy	(1)	(1)		✓			✓
		(3) Ck Secrecy		✓	✓	✓	✓		
		(4) Secrecy	(2)-(3)	(2)-(3)	✓	✓			✓
Set 3	Up and Down	(1) Correctness Invariant		✓	✓	✓	✓	✓	✓
		(2) Correctness End		(1)	✓	✓	✓	✓	✓
		(3) Gen Start		✓	✓	✓	✓	✓	✓
		(4) Destr Seed		(3)	✓	✓	✓	✓	✓
		(5) Gen Unique		✓	✓	✓	✓	✓	✓
		(6) Correspondence		(1)+(5)	✓				
	Loop Exits	(1) Correspondence		✓	✓	✓	✓	✓	✓
		(2) Correspondence Cut		✓	✓	✓		✓	✓
	TESLA 1	Authentic		✓	✓	✓		✓	
	TESLA 2	(1) Unique Keys		✓	✓	✓	✓	✓	
		(2) Key Expiry		(1)	✓	(1)		✓	
		(3) Authentic		(1)-(2)	✓	(1)-(2)		✓	

Table 1: Case studies to compare induction schemes. For a detailed explanation, see the beginning of Section 6.

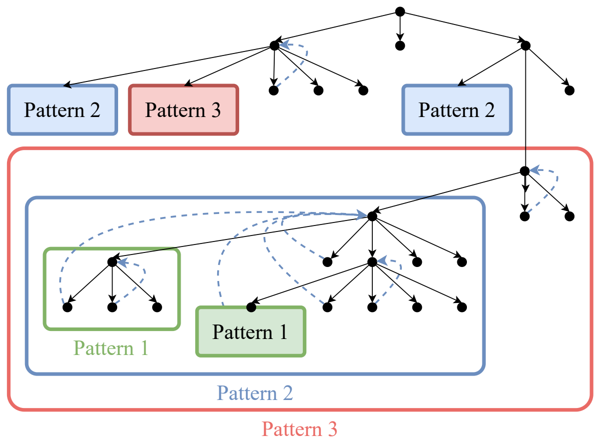

Figure 8: Proof graph of Secrecy for Signal 2. We abstract repeating patterns. The tree contains 31 backlinks.

The part marked in square brackets applies to Signal 2 only. The lemma formalizes that a message m sent from participant a to participant b remains confidential unless one of the following key compromises occurs: (i) One participant's long-term key was revealed (LtkReveal), or (ii) pre-key material of one participant was revealed (RevealPre), or (iii) ephemeral key material from a session between a and b was revealed (RevealEph).

When using cyclic induction, we could prove message secrecy as formalized above for both Signal case studies without requiring any auxiliary lemmas. The proofs for the Signal case studies contain 55 backlinks for Signal 1 and 31 backlinks for Signal 2. Figure 8 depicts the proof graph of the Signal 2 case study and illustrates a complex proof structure. Nevertheless, Tamarin can find a proof automatically when supplied with a simple heuristic that instructs Tamarin to deprioritize solving equations that can lead to case splits. For example, such equations describe the structure of Diffie-Hellman shared secrets, and we observed that solving these equations leads to a state-space blowup during proof construction. With this simple heuristic (along with the heuristics presented in Section 5.2.1), Tamarin finds a proof for each Signal case study in around 40 seconds, showing that cyclic induction can indeed be used to find complex, inductive proofs with little guidance.

For comparison, we also followed the proof methodology presented in [34] to prove message secrecy using trace induction for both Signal case studies. In both cases, we needed to write three very similar auxiliary lemmas to prove message secrecy. Moreover, writing these lemmas required changing the model substantially, adding more annotations to help formalize the lemmas. Our Signal case study shows that cyclic induction drastically simplifies the analysis of protocols like Signal when compared to trace induction and that it is easy to find a cyclic proof for both Signal case studies.

6.2 Limitations

6.2.1 Destructor-based Models. During our case studies, we found that cyclic induction often does not improve, and sometimes even hinders, proof construction for models that are *destructor-based*. We

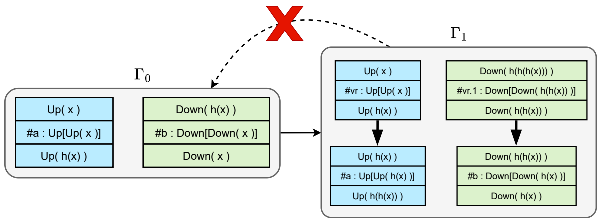

Figure 9: Illustration of diverging terms. Blue nodes are constructor rules, green nodes are destructor rules.

call models destructor-based if they use not only looping constructor rules, but also looping destructor rules. Constructor rules use the terms in their premises to *construct* larger terms in their conclusions. In contrast, destructor rules *destruct* the terms in their premises, creating smaller terms in their conclusions. We encountered this limitation when proving the TESLA protocol, but illustrate it on the smaller model called "Up and down."

The "Up and down" model contains two loops. The first loop generates a seed x and applies a hash function h an arbitrary number of times to that seed. The second loop starts when the first loop ends, and non-deterministically often removes the hash function. Thus, the first loop constructs a term $h^n(x)$, and the second loop destructs that term resulting in $h^m(x)$ $(m \leq n)$.

We call the problem of destructor-based models *diverging terms*. During proof construction for destructor-based theories, one will often encounter cases where both the construction and destruction loops are instantiated, as illustrated in Figure 9. One will generally be unable to find backlinks as the looping rules' terms diverge. Observe that the terms in the premises in Γ_0 in Figure 9 are x and $h(x)$. When solving for the constructor rule's premise, the term in the added premise rule remains x. However, when solving for the destructor rule's premise, the term in the added premise rule is $h(h(h(x)))$. Thus, to find a substitution matching the previous, smaller constraint system, we must both match x with x and $h(x)$ with $h(h(h(x)))$, i.e., x with $h(h(x))$, which is impossible.

While we were able to construct proofs using cyclic induction for destructor-based models, sometimes still reducing the number of auxiliary lemmas (see Table 1), we found it easier to construct proofs for the TESLA case studies with trace induction than with cyclic induction. TESLA is a broadcast authentication protocol for which we analyze two variants. For TESLA, the auxiliary lemmas originally provided for trace induction helped mitigate the issue of diverging terms. However, finding backlinks when using these auxiliary lemmas requires care, e.g., weakening and backlink search must only be applied to a few, selected nodes in the derivation tree, and applying them elsewhere typically yields "unfinishable."

6.2.2 Heuristics. Our heuristics are simple (see Section 5.2.1), but likely not optimal. Excluding Signal and TESLA, all our case studies are automatically provable when using trace induction but not when using cyclic induction. This comparison, however, is unfair as many trace induction proofs require writing auxiliary lemmas, which is an inherently manual task. With cyclic proofs, far fewer auxiliary lemmas are required. Thus, cyclic induction paves the way for future work on improved proof automation.

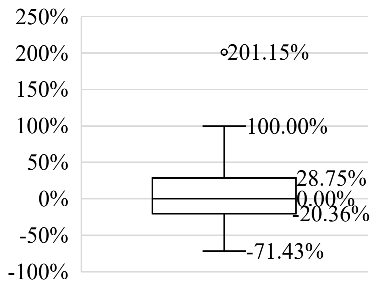

Figure 10: Timing of cyclic induction (CI) compared to trace induction (TI). 100% means that CI takes twice as long as TI.

There are, however, exceptions, and proving some lemmas with cyclic induction required auxiliary lemmas or user-specified weakening. User-specified weakening means that weakening is required to find a proof, but using *minimize for cyclic proof* is insufficient because it would weaken either too much or too little. We leave it as future work to develop better heuristics for cyclic proofs.

6.2.3 Performance. Backlink search is NP-complete, and we thus cannot exclude that there are theories where proof search takes considerable time. In practice, however, we find that our implementation is performant. We timed proof construction on a MacBook with an Apple M2 Max CPU and 32 GB of memory. Complex, real-world case studies such Signal and TESLA can be proven quickly. These case studies take around 45 and 2.5 seconds respectively to verify. Proving all other case studies took at most 0.12 seconds.

When compared to trace induction, we find that cyclic induction takes on average as long as trace induction, with one outlier, see Figure 10. Comparing the verification times directly, however, is unfair as trace induction requires more auxiliary lemmas than cyclic induction. First, auxiliary lemmas must be conjectured by the user, and this process cannot be easily timed. Second, we included the verification time of auxiliary lemmas for cyclic induction. For example, if we only verify Secrecy with cyclic induction but all lemmas with trace induction, then the proof time for Signal 1 and 2 change only by 0.96 (2.6%) and -0.16 seconds (-0.36%) respectively.

7 Related Work

7.1 Cyclic Proof Systems and Tools

7.1.1 Proof Systems and Applications. Cyclic induction proof systems have been developed and used in diverse areas of logic and computer science. These include first-order logic with inductive predicates [13, 17], Peano arithmetic [47], Kleene algebras [28, 43], modal and first-order μ-calculi [46, 49, 2, 1], higher-order fixed point arithmetic [33], equational reasoning about functional programs [31], reasoning about process languages [46, 27], program logics [14, 44, 51], and program synthesis [30].

In many cases, cyclic proof systems are at least as powerful as systems based on explicit induction rules. An interesting theoretical question is whether they are equivalent. This has been settled in the positive for first-order μ-calculus [49] and Peano arithmetic [47] and in the negative for first-order logic with inductive predicates [6].

7.1.2 Tools for Cyclic Proofs. The Erlang Verification Tool was an early implementation of cyclic proofs for proving first-order μ-calculus properties of Erlang programs. It supports induction, co-induction, and their combination via alternating fixed points. Brotherston et al. [15] implemented a proof system for entailment proofs in separation logic using a deep embedding in HOL Light. Cyclist [16] is a generic stand-alone tool for cyclic proofs, which can be instantiated to different logics, e.g., for program termination in separation logic [44] and temporal properties of pointer programs [51]. CycleQ [31] is a tool for equational reasoning about functional programs. Cypress [30] uses cyclic reasoning for the deductive synthesis of heap-manipulating programs from separation logic specifications. Our work is the first use of cyclic proof systems for security protocol verification. This required addressing several challenges, as discussed in the introduction.

7.2 Symbolic Protocol Verification

Two other, modern symbolic protocol verifiers are ProVerif [10] and DY* [7]. Both tools were used to analyze protocols comparable to our case studies. ProVerif was used to analyze Signal [32] and its post-quantum secure key agreement protocol PQXDH [9], but both analyses are quite limited in scope. The former does not consider Signal's symmetric ratchet, and only considers a finite number of protocol sessions. The latter analyzes PQXDH in isolation and thus does not consider any of Signal's looping behavior. Recently, ProVerif was extended to support trace induction [11] comparable to Tamarin's trace induction. As in Tamarin, the lemma to be proven must be formulated as an inductive statement, and the induction variable is the length of the trace and thus fixed.

ProVerif's induction was used to verify election protocols [20] and protocols using authenticated data structures [21]. We discuss [21] in future work. The election protocol model in [20] includes two loops, one to fix the number of voters (non-deterministically counting up), and one to tally the votes (counting down). These loops resemble the "Up and down" model (see Section 6.2.1), for which we proved properties similar to those in [20].

ProVerif's induction is fairly new and there are few case studies using it. It remains to be seen whether ProVerif's induction scales to protocols like Signal, which we proved as part of our case studies with little effort. However, given that ProVerif's induction mechanism closely resembles Tamarin's trace induction, we expect that there are limitations similar to Tamarin's.

DY* was employed for a formal analysis of Signal [7], and it was later also used to analyze parts of the Message Layer Security group messaging protocol [52]. DY* builds on the program verifier F*, and thus uses dependent types. It maintains a global trace variable that is used to express security properties. DY* is very expressive and allows for intricate proof strategies. This comes at the expense of significant manual interaction, as DY* proofs require user-specified invariants on the global trace that are strong enough to imply the desired security property. The original DY* paper was motivated by looping protocols, like Signal, being out-of-scope for modern protocol verifiers, like Tamarin. The formal analysis of iMessage, which is more complex than Signal, showed that this is not the case [34].

8 Conclusion

We have introduced cyclic induction reasoning in Tamarin, proved its soundness, implemented it, and evaluated it on fourteen case studies, showing that it can be used to easily find proofs for complex protocols such as Signal. Cyclic induction exploits repeating patterns in Tamarin's constraints systems, and thereby avoids previous sources of non-termination during proof construction. Moreover, cyclic induction fundamentally changes how one constructs inductive proofs. Tamarin's previous induction scheme, trace induction, required writing inductive lemmas and could only be applied at the start of a proof. In contrast, cyclic induction enables Tamarin to automatically, and on-the-fly, discover inductive proofs. By finding repeating patterns in graphs, cyclic induction avoids the need for auxiliary lemmas in many practically relevant case studies. In contrast to writing auxiliary lemmas, finding repeating patterns in graphs is much better suited for automation.

Future Work. We believe that cyclic induction can benefit from further optimized heuristics and algorithms for constructing cyclic proofs automatically. There are many promising options here. For example, there is an enormous body of work on the problem of *subgraph matching* (e.g., [53, 12, 50]). Applying results from this field could further improve our implementation of backlink search. Additionally, one could explore better search strategies. For example, backtracking when negated branches of a cut do not lead to a contradiction, automated weakening of formulas that are not required to derive contradictions in base cases, and much more. Another interesting line of work is exploring whether cyclic induction can simplify proving properties of protocols using authenticated data structures such as Merkle Hash Trees in the symbolic model, which was initially suggested by [21]. Finally, our cyclic induction framework does not yet apply to Tamarin's mode for proving observational equivalence [4]. Investigating whether it does also remains as future work.

Acknowledgments

We thank the Werner Siemens-Stiftung (WSS) for their generous support of this project.

We also thank Nadia Polikarpova for her explanations of cyclic proofs for "Synthesis of Safe Pointer-Manipulating Programs," which inspired us to apply them to the Tamarin prover.

References

[1] Bahareh Afshari, Sebastian Enqvist, and Graham E Leigh. 2024. Cyclic proofs for the first-order μ-calculus. *Logic Journal of the IGPL*, 32, 1, (Jan. 25, 2024). DOI: 10.1093/jigpal/jzac053.

[2] Bahareh Afshari and Graham E. Leigh. 2017. Cut-free completeness for modal mu-calculus. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. (June 2017). DOI: 10.1109/LICS.2017.8005088.

[3] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. 2018. A Formal Analysis of 5G Authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS. (Oct. 15, 2018). DOI: 10.1145/3243734.3243846.

[4] David Basin, Jannik Dreier, and Ralf Sasse. 2015. Automated Symbolic Proofs of Observational Equivalence. In *22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS. (Oct. 2015). DOI: 10.1145/2810103.2813662.

[5] David Basin, Ralf Sasse, and Jorge Toro-Pozo. 2021. The EMV Standard: Break, Fix, Verify. In *2021 IEEE Symposium on Security and Privacy (S&P)*. (May 2021). DOI: 10.1109/SP40001.2021.00037.

[6] Stefano Berardi and Makoto Tatsuta. 2019. Classical System of Martin-Lof's Inductive Definitions is not Equivalent to Cyclic Proofs. *Logical Methods in Computer Science*, Volume 15, Issue 3, (Aug. 1, 2019). DOI: 10.23638/LMCS-15(3:10)2019.

[7] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. 2021. DY* : A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code. In 6th IEEE European Symposium on Security and Privacy (EuroS&P). (Sept. 6, 2021). DOI: 10.1109/EuroSP51992.2021.00042.

[8] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. 2017. Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate. In *2017 IEEE Symposium on Security and Privacy (S&P)*. (May 2017). DOI: 10.1109/SP.2017.26.

[9] Karthikeyan Bhargavan, Charlie Jacomme, Franziskus Kiefer, and Rolfe Schmidt. 2024. Formal verification of the PQXDH Post-Quantum key agreement protocol for end-to-end secure messaging. In 33rd USENIX Security Symposium. https://www.usenix.org/conference/usenixsecurity24/presentation/bhargavan.

[10] Bruno Blanchet. 2016. Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif. *Foundations and Trends® in Privacy and Security*, 1, 1–2, (Oct. 30, 2016). DOI: 10.1561/3300000004.

[11] Bruno Blanchet, Vincent Cheval, and Véronique Cortier. 2022. ProVerif with Lemmas, Induction, Fast Subsumption, and Much More. In *2022 IEEE Symposium on Security and Privacy (S&P)*. (May 2022). DOI: 10.1109/SP46214.2022.9833653.

[12] Sarra Bouhenni, Saïd Yahiaoui, Nadia Nouali-Taboudjemat, and Hamamache Kheddouci. 2021. A Survey on Distributed Graph Pattern Matching in Massive Graphs. *ACM Comput. Surv.*, 54, 2, (Feb. 9, 2021). DOI: 10.1145/3439724.

[13] James Brotherston. 2005. Cyclic Proofs for First-Order Logic with Inductive Definitions. In *Automated Reasoning with Analytic Tableaux and Related Methods*. DOI: 10.1007/11554554_8.

[14] James Brotherston, Richard Bornat, and Cristiano Calcagno. 2008. Cyclic proofs of program termination in separation logic. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL. (Jan. 7, 2008). DOI: 10.1145/1328438.1328453.

[15] James Brotherston, Dino Distefano, and Rasmus Lerchedahl Petersen. 2011. Automated Cyclic Entailment Proofs in Separation Logic. In *Automated Deduction – CADE-23*. DOI: 10.1007/978-3-642-22438-6_12.

[16] James Brotherston, Nikos Gorogiannis, and Rasmus L. Petersen. 2012. A Generic Cyclic Theorem Prover. In *Programming Languages and Systems*. DOI: 10.1007/978-3-642-35182-2_25.

[17] James Brotherston and Alex Simpson. 2011. Sequent calculi for induction and infinite descent. *Journal of Logic and Computation*, 21, 6, (Dec. 1, 2011). DOI: 10.1093/logcom/exq052.

[18] Alan Bundy. 2001. Chapter 13 - The Automation of Proof by Mathematical Induction. In *Handbook of Automated Reasoning*. (Jan. 1, 2001). DOI: 10.1016/B978-044450813-3/50015-1.

[19] Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. 2005. *Rippling: Meta-Level Guidance for Mathematical Reasoning*. DOI: 10.1017/CBO9780511543326.

[20] Vincent Cheval, Véronique Cortier, and Alexandre Debant. 2023. Election Verifiability with ProVerif. In *36th Computer Security Foundations Symposium (CSF)*. (July 2023). DOI: 10.1109/CSF57540.2023.00032.

[21] Vincent Cheval, José Moreira, and Mark Ryan. 2023. Automatic Verification of Transparency Protocols. In *8th European Symposium on Security and Privacy (EuroS&P)*. (July 2023). DOI: 10.1109/EuroSP57164.2023.00016.

[22] Stephen A. Cook. 1971. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. New York, NY, USA, (May 3, 1971). DOI: 10.1145/800157.805047.

[23] Cas Cremers, Jaiden Fairoze, Benjamin Kiesl, and Aurora Naska. 2020. Clone Detection in Secure Messaging: Improving Post-Compromise Security in Practice. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS. (Nov. 2, 2020). DOI: 10.1145/3372297.3423354.

[24] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. 2017. A Comprehensive Symbolic Analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS. (Oct. 30, 2017). DOI: 10.1145/3133956.3134063.

[25] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. 2016. Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication. In *2016 IEEE Symposium on Security and Privacy (S&P)*. (May 2016). DOI: 10.1109/SP.2016.35.

[26] Cas Cremers, Charlie Jacomme, and Philip Lukert. 2023. Subterm-Based Proof Techniques for Improving the Automation and Scope of Security Protocol Analysis. In *36th Computer Security Foundations Symposium (CSF)*. (July 2023). DOI: 10.1109/CSF57540.2023.00001.

[27] Mads Dam and Dilian Gurov. 2002. μ-calculus with explicit points and approximations. *Journal of Logic and Computation*, 12, 2, (Apr. 1, 2002). DOI: 10.1093/logcom/12.2.255.

[28] Anupam Das and Damien Pous. 2017. A Cut-Free Cyclic Proof System for Kleene Algebra. In *Automated Reasoning with Analytic Tableaux and Related Methods*. DOI: 10.1007/978-3-319-66902-1_16.

[29] Guillaume Girol, Lucca Hirschi, Ralf Sasse, Dennis Jackson, Cas Cremers, and David Basin. 2020. A Spectral Analysis of Noise: A Comprehensive, Automated, Formal Analysis of Diffie-Hellman Protocols. In 29th USENIX Security Symposium. https://www.usenix.org/conference/usenixsecurity20/presentation/girol.

[30] Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. 2021. Cyclic program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* (June 18, 2021). DOI: 10.1145/3453483.3454087.

[31] Eddie Jones, C.-H. Luke Ong, and Steven Ramsay. 2022. CycleQ: an efficient basis for cyclic equational reasoning. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* (June 9, 2022). DOI: 10.1145/3519939.3523731.

[32] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. 2017. Automated Verification for Secure Messaging Protocols and Their Implementations: A Symbolic and Computational Approach. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P).* (Apr. 2017). DOI: 10.1109/EuroSP.2017.38.

[33] Mayuko Kori, Takeshi Tsukada, and Naoki Kobayashi. 2021. A cyclic proof system for hfl_ℕ. In *29th EACSL Annual Conference on Computer Science Logic (CSL 2021).* Vol. 183. DOI: 10.4230/LIPIcs.CSL.2021.29.

[34] Felix Linker, Ralf Sasse, and David Basin. 2025. A Formal Analysis of Apple's iMessage PQ3 Protocol. In 34th USENIX Security Symposium. Seattle, WA, USA, (Aug. 13–15, 2025). https://www.usenix.org/conference/usenixsecurity25/presentation/linker.

[35] [SW] Felix Linker, Christoph Sprenger, Cas Cremers, and David Basin, Looping for Good: Cyclic Proofs for Security Protocols Apr. 11, 2025. DOI: 10.5281/zenodo.16992323.

[36] Felix Linker, Christoph Sprenger, Cas Cremers, and David Basin. Looping for Good: Cyclic Proofs for Security Protocols (Full Version). (Oct. 2025). DOI: 10.3929/ethz-c-000783356.

[37] Moxie Marlinspike and Trevor Perrin. 2016. The X3DH Key Agreement Protocol. Revision 1. (Nov. 4, 2016). https://signal.org/docs/specifications/x3dh/x3dh.pdf.

[38] Simon Meier. 2013. *Advancing Automated Security Protocol Verification.* Doctoral Thesis. ETH Zurich. DOI: 10.3929/ethz-a-009790675.

[39] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. 2013. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification (CAV).* DOI: 10.1007/978-3-642-39799-8_48.

[40] Aleksi Peltonen, Ralf Sasse, and David Basin. 2021. A comprehensive formal analysis of 5G handover. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks.* (June 28, 2021). DOI: 10.1145/3448300.3467823.

[41] Trevor Perrin and Moxie Marlinspike. 2016. The Double Ratchet Algorithm. Revision 1. (Nov. 20, 2016). https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf.

[42] 2024. Proof by infinite descent. In *Wikipedia.* (Dec. 24, 2024). Retrieved Apr. 8, 2025 from https://en.wikipedia.org/w/index.php?title=Proof_by_infinite_descent&oldid=1264957154.

[43] Jan Rooduijn, Dexter Kozen, and Alexandra Silva. 2024. A Cyclic Proof System for Guarded Kleene Algebra with Tests. In *Automated Reasoning.* DOI: 10.1007/978-3-031-63501-4_14.

[44] Reuben N. S. Rowe and James Brotherston. 2017. Automatic cyclic termination proofs for recursive procedures in separation logic. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs.* (Jan. 16, 2017). DOI: 10.1145/3018610.3018623.

[45] Ulrich Schöpp. 2001. *Formal Verification of Processes.* Master's thesis. University of Edinburgh. Retrieved Apr. 8, 2025 from https://ulrichschoepp.de/Docs/msc.pdf.

[46] Ulrich Schöpp and Alex Simpson. 2002. Verifying Temporal Properties Using Explicit Approximants: Completeness for Context-free Processes. In *Foundations of Software Science and Computation Structures.* DOI: 10.1007/3-540-45931-6_26.

[47] Alex Simpson. 2017. Cyclic Arithmetic Is Equivalent to Peano Arithmetic. In *Foundations of Software Science and Computation Structures.* DOI: 10.1007/978-3-662-54458-7_17.

[48] Christoph Sprenger and Mads Dam. 2003. On global induction mechanisms in a μ-calculus with explicit approximations. *RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications*, 37, 4. DOI: 10.1051/ita:2003024.

[49] Christoph Sprenger and Mads Dam. 2003. On the structure of inductive reasoning: circular and tree-shaped proofs in the μ-calculus. In *Foundations of Software Science and Computation Structures.* DOI: 10.1007/3-540-36576-1_27.

[50] Xibo Sun, Shixuan Sun, Qiong Luo, and Bingsheng He. 2022. An in-depth study of continuous subgraph matching. *Proc. VLDB Endow.*, 15, 7, (Mar. 1, 2022). DOI: 10.14778/3523210.3523218.

[51] Gadi Tellez and James Brotherston. 2020. Automatically Verifying Temporal Properties of Pointer Programs with Cyclic Proof. *Journal of Automated Reasoning*, 64, 3, (Mar. 1, 2020). DOI: 10.1007/s10817-019-09532-0.

[52] Théophile Wallez, Jonathan Protzenko, Benjamin Beurdouche, and Karthikeyan Bhargavan. 2023. TreeSync: Authenticated Group Management for Messaging Layer Security. In 32nd USENIX Security Symposium. (Aug. 2023). https://www.usenix.org/conference/usenixsecurity23/presentation/wallez.

[53] Zhijie Zhang, Yujie Lu, Weiguo Zheng, and Xuemin Lin. 2024. A Comprehensive Survey and Experimental Study of Subgraph Matching: Trends, Unbiasedness, and Interaction. *Proc. ACM Manag. Data*, 2, 1, (Mar. 26, 2024). DOI: 10.1145/3639315.