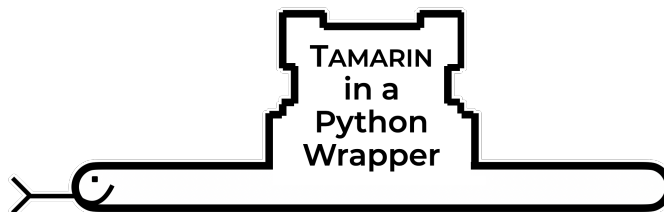


Internship Report

batch-tamarin: The Snake Tames the Monkey – A Python Wrapper for Automated Analysis of Security Protocols in the Symbolic Model of Cryptography

Luca Mandrelli

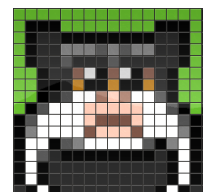
Academic Year 2024–2025



Second year internship done in partnership with the CISPA Helmholtz Center for Information Security

Internship supervisor: Prof. Dr. Cas Cremers

Academic supervisor: Isabelle Heudiard



Déclaration sur l'honneur de non-plagiat

Je soussigné(e),

Nom, prénom : Mandrelli, Luca

Élève-ingénieur(e) régulièrement inscrit(e) en 2^e année à TELECOM Nancy

Numéro de carte de l'étudiant(e) : 32009662

Année universitaire : 2024–2025

Auteur(e) du document, mémoire, rapport ou code informatique intitulé :

batch-tamarin: The Snake Tames the Monkey –
A Python Wrapper for Automated Analysis of Security Protocols
in the Symbolic Model of Cryptography

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Pont-à-Mousson, le 25 août 2025

Signature :



Internship Report

batch-tamarin: The Snake Tames the Monkey – A Python Wrapper for Automated Analysis of Security Protocols in the Symbolic Model of Cryptography

Luca Mandrelli

Academic Year 2024–2025

Second year internship done in partnership with the CISPA Helmholtz Center for Information Security

Luca Mandrelli
83, rue Maurice Barrès
54000, Pont-à-Mousson
07 81 54 33 86
luca.mandrelli@telecomnancy.eu

TELECOM Nancy
193 avenue Paul Muller,
CS 90172, VILLERS-LÈS-NANCY
+33 (0)3 83 68 26 00
contact@telecomnancy.eu

CISPA Helmholtz Center for Information Security
Stuhlsatzenhaus 5, Saarland Informatics Campus
66123 Saarbrücken, Germany
+49 681 / 87083 1001
info@cispa.de



Internship supervisor: Prof. Dr. Cas Cremers

Academic supervisor: Isabelle Heudiard

Acknowledgements

“I would like to thank Cas Cremers for welcoming me into his team, making this internship possible, and providing excellent supervision and guidance throughout this experience.

I am grateful to Maiwenn Racouchot for introducing this opportunity and for her support during the project.

I thank Aleksi Peltonen for his assistance and constructive feedback.

I also thank Esra Günsay, Erik Pallas, Niklas Medinger, Aurora Naska, and Alexander Dax for their warm welcome and helpful development ideas.

I thank Isabelle Heudiard for overseeing the smooth running of this internship.

Finally, I thank the staff at CISPA and at Telecom Nancy for making this internship possible.”

– Luca Mandrelli

Foreword

This report presents the outcome of an eleven-week internship at the CISPA Helmholtz Center for Information Security in Saarbrücken, Germany (June 1–August 15, 2025). The internship was completed as part of the second year of the engineering program at Telecom Nancy.

During my second year at Telecom Nancy, I attended a presentation by Maiwenn Racouchot, a former student of Telecom Nancy, about her work as a postdoctoral researcher at CISPA. This talk sparked my interest in research and motivated me to apply for an internship at CISPA.

The internship took place in a collaborative environment with a team of talented researchers working on the formal analysis of security protocols. It provided hands-on experience in building and maintaining a growing software project from scratch, while offering insight into the role of a research engineer.

This report focuses on the `batch-tamarin` project, a wrapper around the Tamarin Prover allowing batch processing of multiple verification tasks. It covers its context, objectives, design choices, implementation, and outcomes, as well as contributions to the Tamarin Prover community that made this project possible.

Contents

Acknowledgements	v
Foreword	vii
Contents	ix
1 Introduction	1
2 The CISPA Helmholtz Center for Information Security	3
2.1 Context	3
2.2 Mission and Values	3
2.3 Research and Key Figures	4
3 Project Overview and Methods	5
3.1 Project Context	5
3.2 Project Environment	5
3.3 Internship Supervision	6
3.4 Project Goals	6
3.5 Methodology	6
4 Related Work and Tooling Landscape	8
4.1 State-of-the-Art: Existing Tools	8
4.1.1 UT-Tamarin	8
4.1.2 ANSSI parser and server (<i>tamarin-parser</i>)	9
4.1.3 Team prototype: a Python wrapper	9
4.2 Design Choices	10
4.2.1 Python	10
4.2.2 Command-line interface (CLI)	11
4.2.3 Nix	11
4.2.4 Docker	12

4.2.5	Summary	12
5	Implementation and Extensions	13
5.1	Hands-on Exploration Phase	13
5.1.1	Understanding Tamarin Prover	13
5.1.2	First Python Experiments	13
5.2	Core system (MVP: Minimum Viable Product)	14
5.2.1	The Config File	14
5.2.2	The Core of the Execution Pipeline	16
5.3	Lemma Parser & Model Introspection	17
5.3.1	Context	17
5.3.2	Implementation	17
5.4	Preflight Validation	18
5.4.1	Motivation	18
5.4.2	Implementation	18
5.5	Interactive Recipe Generator	19
5.5.1	Motivation	19
5.5.2	Implementation	19
5.5.3	Results	20
5.6	Caching & Attack-Trace Export	20
5.6.1	Problem and Motivations	20
5.6.2	Implementation	20
5.7	Reporting & Rerun Recipes	21
5.7.1	Motivation	21
5.7.2	The challenges	21
5.7.3	Design Choice	22
5.7.4	Implementation	22
5.8	Scheduling Policies	23
5.8.1	The problem	23
5.8.2	The solution	23
6	Results and Outlook	25
6.1	Results	25
6.2	Future Work	25
6.3	Personal Reflection	26

7 Concluding Remarks	27
Bibliography / Webography	29
List of Figures	31
Listings	32
Glossary	33
 Appendices	 36
A Appendix A	36
B Appendix B	38
Résumé	43
Abstract	43

1 Introduction

Security protocols form the foundation that enables secure communication between machines over the Internet. By default, communication occurs over an untrusted network, meaning we cannot assume that peers—or even the network infrastructure—are honest. Security protocols, therefore, aim to guarantee properties that enable secure communication. Core guarantees include confidentiality (ensuring only the intended recipient learns the content), integrity (ensuring messages are not modified without detection), and authentication (ensuring each party knows who it is communicating with). Two additional protections are especially important in practice: *replay protection*, which prevents an attacker from reusing old but valid messages to trick a party into repeating an action (for example, resubmitting a payment), and *forward secrecy*, which ensures that even if a long-term secret key is compromised later, past session keys—and thus past communications—remain confidential.

Symbolic model verification of cryptographic protocols uses mathematical methods to analyze protocol designs and prove that they satisfy their intended security properties. In this approach, cryptographic operations are treated as perfect abstract functions, and an adversary (typically modeled after Dolev–Yao model) can intercept, modify, and replay messages but cannot break cryptographic primitives through computational attacks. The verification process involves modeling the protocol as a formal system, specifying security properties as logical statements, and using automated reasoning tools to either prove these properties hold or find concrete attack scenarios that violate them.

This verification method provides mathematically rigorous evidence that protocols behave as intended under the specified threat model, helping to identify subtle flaws that might be missed through informal analysis or testing alone. Widely used tools using this verification method include *Tamarin Prover* [13, 1] and *ProVerif* [3].

Tamarin Prover is the tool we focus on in this report. It provides a powerful framework for modeling and analyzing protocols, and it has been applied to real-world designs such as *TLS 1.3* [10], *Signal* [11], *5G-AKA* [2], *SPDM* [9], and *WPA2* [12]. *Tamarin* models protocols as state-transition (multiset-rewrite) rules under a strong attacker who controls the network. We model cryptography with algebra-like rules and write the security goals as properties of the protocol’s possible runs (its *traces*). *Tamarin* then either constructs a proof or returns a step-by-step attack trace that witnesses a violation.

Despite its strengths, *Tamarin* is not designed to orchestrate independent proofs, compare results, or aggregate outcomes into experiment-ready tables and logs. Research workflows often require exactly these capabilities: running large batches with different models, options, or prover versions; handling failures robustly; and extracting consistent metrics. Traditionally, this has been accomplished on a case-by-case basis for each research experiment. The goal of my internship was to develop *batch-tamarin*, a single Python-based interface that (i) lets researchers declare an experiment in a compact configuration, (ii) schedules and parallelizes runs, (iii) man-

ages multiple Tamarin versions, (iv) captures and normalizes outputs and errors, and (v) produces reproducible summaries for downstream analysis. The aim is to save time, reduce manual scripting, make protocol studies easier to reproduce, and, most importantly, have an official and unified way to handle these features.

This report presents the context of the project, a state-of-the-art analysis of existing tools for automating Tamarin workflows, a description of my design and contributions to `batch-tamarin`, the challenges encountered, and finally a brief review of the results.

2 The CISP Helmholz Center for Information Security

2.1 Context

Profile and location. The CISP Helmholz Center for Information Security is a German national research center within the *Helmholtz Association*. Its headquarters are located on the Saarland Informatics Campus in Saarbrücken (main building: Stuhlsatzenhaus 5, 66123 Saarbrücken), with additional sites currently operating in St. Ingbert; further CISP research groups are based in Hannover and Dortmund. In the coming years, CISP plans to relocate to a purpose-built research campus in St. Ingbert. [4]

History. CISP was founded in October 2011 as the BMBF-funded Center for IT-Security, Privacy and Accountability at Saarland University. After years of growth, CISP evolved into an independent institution and joined the Helmholtz Association in 2017; in 2019, it was inducted as a full member, becoming the 19th Helmholtz Center in Germany. In the most recent international evaluation of Helmholtz Centers (June 2025), CISP received the highest possible rating, “*Outstanding*”, across all evaluation criteria, recognizing its global leadership, groundbreaking research with broad impact, and strong transfer and training capabilities.[5]

Governance. CISP operates as a non-profit limited liability company (*gGmbH*¹). The Scientific Director and CEO is Prof. Dr. Dr. h. c. Michael Backes, the Chief Operating Officer, is Dr. Kevin Streit.

2.2 Mission and Values

CISP’s mission is to advance the science and practice of information security and trustworthy artificial intelligence across its full breadth, combining foundational research with application-oriented work. Its institutional values emphasize:

- addressing the grand challenges of cybersecurity and privacy in a comprehensive and holistic manner;
- conducting cutting-edge foundational and applied research to the highest international academic standards;

¹German legal form: “gemeinnützige Gesellschaft mit beschränkter Haftung” = non-profit limited liability company.

- maintaining a world-class research environment that attracts and supports top scientific talent from around the globe;
- educating and mentoring the next generation of experts and scientific leaders in cybersecurity and privacy;
- achieving societal impact through technology transfer, startup support, and sustained public engagement.

2.3 Research and Key Figures

Research areas. CISPA’s research agenda encompasses the full spectrum of information security, structured around six closely connected areas: [8]

- **Algorithmic Foundations and Cryptography** – Foundations of cryptography and algorithms for security: design and rigorous analysis of primitives and protocols, provable security (including post-quantum²), and privacy-enhancing technologies;
- **Trustworthy Information Processing** – Methods to ensure correctness, safety, and privacy of data processing across software and hardware: formal verification, static and dynamic analysis, secure compilation, and trustworthy/robust machine learning;
- **Reliable Security Guarantees** – End-to-end models and proofs that deliver auditable security claims for complex systems and protocols, including scalable verification, specification, testing, and certification-grade assurance;
- **Threat Detection and Defenses** – Empirical, measurement-driven, and ML-assisted detection of attacks and abuse, malware and vulnerability analysis, network and web security, and proactive defense mechanisms;
- **Secure Connected and Mobile Systems** – Security and privacy of operating systems, mobile/embedded/IoT and cyber-physical systems, wireless and automotive platforms, including side-channel and hardware-assisted threats;
- **Empirical and Behavioral Security** – Human-centered security and privacy, usability, risk perception, and decision-making, large-scale measurements of online harms and interventions grounded in behavioral science.

Selected indicators (state: June 2025). CISPA reports 1,030 conference papers in total, including 528 A* publications³, and 16 ERC⁴ grants. According to CSRankings, CISPA is ranked first worldwide in the areas of Computer Security and Cryptography. [6] CISPA researchers regularly publish at and help organize premier venues; for instance, they contributed 11 papers to NDSS⁵ 2025, 13 papers to ICLR⁶ 2025, 15 papers at IEEE S&P⁷ 2025 and 27 papers at USENIX Security Symposium 2025. [7]

²Cryptographic algorithms designed to resist attacks by quantum computers.

³“A*” denotes the top-tier venues in the CORE conference ranking.

⁴European Research Council: prestigious EU funding program for frontier research.

⁵Network and Distributed System Security Symposium.

⁶International Conference on Learning Representations.

⁷IEEE Symposium on Security and Privacy (also known as “Oakland”).

3 Project Overview and Methods

3.1 Project Context

Working with the `Tamarin Prover` often requires batch processing: running a model with different parameters to study performance trade-offs, comparing different prover versions to detect regressions or evaluate improvements, and executing multiple models to reproduce experiments. `Tamarin` does not natively provide orchestration for such workloads, so researchers typically rely on custom scripts. This approach leads to code duplication, inconsistent outputs, and limited reproducibility.

The `batch-tamarin` project addresses this gap by providing a unified, Python-based interface to define experiments and run them in a reproducible manner. Key features include: (i) declare experiments, (ii) manage multiple `Tamarin` versions, (iii) execute and parallelize runs, (iv) monitor progress and resource consumption, (v) capture outputs and errors in a consistent format, and (vi) create automated reports. The objective is to improve reproducibility and reduce manual engineering effort. The primary users are researchers who model and analyze protocols with `Tamarin Prover` and artifact reviewers who need to be able to reproduce published experiments with minimal effort. A key requirement is portability—experiments should run on "any computer" without configuration changes (supporting macOS and Linux on both x86_64 and ARM architectures).

3.2 Project Environment

I worked within the *Reliable Security Guarantees* research area under the supervision of Cas Cremers. The team consists of eight researchers; I shared an office with three of them: Aleksi, Esra, and Maiwenn.

Development was primarily conducted on macOS using my personal laptop. For Linux testing and building cross-platform dependencies and packages (for both Linux and macOS)—necessitated by the project's multi-architecture and multi-system support—I used GitHub Actions to automate builds and tests.

3.3 Internship Supervision

My internship was supervised by Cas Cremers, one of the four maintainers of the `Tamarin Prover`, whose expertise in the `Tamarin Prover` proved to be a valuable resource throughout the project. He helped me identify the optimal approach to ensure the `batch-tamarin` project's success. His suggestions regarding feature prioritization were invaluable, and his feedback on implemented features was essential for refining the tool.

Aleksi and Maiwenn also supervised my work, providing the initial insights and guidance needed to begin the project and define its scope. We met weekly to discuss progress and address challenges. These meetings were informal and facilitated open discussions, which helped refine the goals as the project was progressing.

3.4 Project Goals

The `batch-tamarin` project was designed to achieve two main objectives:

Serving as a powerful tool for benchmarking and comparing Tamarin models — enabling researchers to run multiple experiments in parallel and aggregate results. Its main strengths are its high customization options and the multiple additional features it offers. To achieve this, I designed the most flexible and extensible architecture possible while maintaining a user-friendly interface. It was also important to ensure that command customization remained straightforward and well-documented.

Providing an easy method to reproduce Tamarin experiments — by offering a simple and consistent way to define experiments with a configuration file, the tool aims to facilitate result reproduction for both researchers and reviewers. Integration of Nix and Docker further enhances reproducibility by providing consistent, reproducible environments for the `Tamarin Prover`.

An additional objective was ensuring comprehensive documentation and maintainable code. This was achieved by following software development best practices, including the use of type hints, docstrings, and consistent coding style. Comprehensive examples were also developed to illustrate the tool's usage.

3.5 Methodology

I began this project with an exploration of the `Tamarin Prover`, its existing features, and its capabilities. This initial research phase was crucial for understanding the tool's architecture and for preparing example commands that `batch-tamarin` should support.

I then evaluated several approaches to invoking the `Tamarin Prover` from Python, which helped determine the best way to handle Tamarin commands and outputs. I also explored existing tools and libraries that could facilitate the development of `batch-tamarin`.

Before beginning the implementation, I conducted a state-of-the-art survey of the existing tools available for automating `Tamarin Prover` workflows. This research helped identify the gaps and limitations of current solutions and to define the requirements for `batch-tamarin`.

Finally, I implemented `batch-tamarin` following an iterative and incremental approach. I started with a minimal viable product that included the core features (running Tamarin models, capturing outputs, and managing multiple Tamarin versions). I used a GitHub issues and pull requests workflow to track progress. Each new feature or bug was addressed through a dedicated issue, and pull requests were used to describe the implemented feature or fixed bug in detail, as I was the only developer on this project.

I chose to publish the project as a Python package on PyPI to allow easy installation. This makes the project more accessible to researchers and reviewers; however, it also makes the development process more complex than standard Python scripts.

4 Related Work and Tooling Landscape

Before implementing `batch-tamarin`, I surveyed existing tooling used by researchers to run the Tamarin Prover at scale and to manage experiments. The objective was to understand what works well in practice and where workflows still require custom scripts. I evaluate prior art along four criteria that matter for reproducible studies: (i) *scope* (single-model helpers vs. many-model orchestration), (ii) *configurability* (declarative configuration, CLI vs. UI), (iii) *reproducibility* (version pinning and environment management), and (iv) *observability* (structured outputs, logs, and aggregation).

The review below focuses on three representative approaches: a focused lemma-testing utility (UT-Tamarin), a server-based parser and UI developed by ANSSI, and a Python wrapper built for specific papers within the team. These informed the requirements and design choices of `batch-tamarin`.

4.1 State-of-the-Art: Existing Tools

4.1.1 UT-Tamarin

Summary. UT-Tamarin¹ is an open-source C++ utility that helps develop and maintain individual Tamarin models by running lemmas under different heuristics and timeouts. According to its documentation, it (i) runs Tamarin on a list of lemmas, (ii) allows lemma-specific heuristics, and (iii) can try a predefined set of heuristics to “penetrate” difficult lemmas. Configuration is provided via a JSON file that can enumerate allow/deny lists and set global or per-lemma fact priorities.

Capabilities. A notable feature is the ability to annotate facts to influence proof search (prioritize or deprioritize), implemented by adding prefixes before invoking Tamarin.² UT-Tamarin expects a system-installed ‘tamarin-prover’ available on the ‘PATH’, and is geared towards single-model, single-machine usage.

Assessment. UT-Tamarin is effective for *model development*—quickly iterating on lemmas and experimenting with heuristics—rather than for orchestrating large, multi-model experiments. It does not manage multiple Tamarin versions, nor does it provide structured, aggregated outputs for downstream analysis. These constraints make end-to-end reproducible pipelines harder to achieve when studies span many models or provers’ versions.

¹https://github.com/benjaminkiesl/ut_tamarin

²The Tamarin manual describes fact annotations that adjust heuristic priority, e.g., ‘+[+]’ / ‘[-]’

4.1.2 ANSSI parser and server (*tamarin-parser*)

Summary. ANSSI’s *tamarin-parser*³ combines (i) a parser that can recover and rewrite the AST of ‘.spthy’ models⁴ and (ii) a server that configures analyses and summarizes results through a UI. The repository bundles the models used in their study on random nonce misgeneration together with the tooling.

Capabilities. The parser is general-purpose in the sense that it can be repurposed for other analyses beyond the published study, enabling programmatic model transformations. The server architecture supports interactive use through a UI.

Assessment. This approach is well-suited to an interactive, paper-centered workflow with a bespoke transformation pipeline. However, it offers little in the way of a *declarative, reusable* configuration for large-scale batch runs. The documentation does not expose built-in multi-version management for Tamarin or an external configuration file; most behavior appears to be defined in code or via the UI, which complicates headless, reproducible pipelines. (This is our inference from the repository structure and README.)

4.1.3 Team prototype: a Python wrapper

Summary. A prototype within the team implemented a Python wrapper around Tamarin for specific papers and benchmarks. Two publicly visible examples illustrate the pattern: (i) the *Symbolic_KEM_Models*⁵ artifact, which includes shell scripts (e.g., `analysis.sh`) and a `tamarin_wrapper.py` to orchestrate case studies, and (ii) *additionalMaterialAutomationTamarin*⁶, which contains benchmarking scripts (`benchmark.py`, `tamarin_wrapper.py`) and post-processing utilities (`results.py`, plotting scripts) tied to a particular study.

Capabilities and limitations. These Python-based approaches demonstrate that lightweight orchestration is practical and extensible. In practice, however, they rely on ad hoc text files and per-paper scripts, which are difficult to read, reuse, and maintain at scale; output normalization and metadata capture are limited, and there is no built-in support for managing multiple Tamarin versions concurrently. Consequently, reproducing results across machines or extending experiments often requires manual adaptation.

Research conclusion. Across these projects, we observe complementary strengths: per-lemma heuristic exploration and JSON configuration (UT-Tamarin), powerful parsing and UI support (ANSSI), and pragmatic Python orchestration (team prototype). None, however, delivers a unified, declarative, and *reproducible* batch layer that manages multiple prover versions, schedules many models, and emits structured, comparable outputs. This gap motivates *batch-tamarin*’s design goals.

³<https://github.com/ANSSI-FR/tamarin-parser>

⁴The Tamarin models syntax is called *SPTHY*

⁵https://github.com/FormalKEM/Symbolic_KEM_Models

⁶<https://github.com/racoucholu/additionalMaterialAutomationTamarin>

4.2 Design Choices

This section explains the principal engineering choices made while designing `batch-tamarin`. For each decision, I discuss the motivating requirements, the alternatives that were considered, the trade-offs, and the practical consequences for maintainability, reproducibility, and adoption in a research group.

4.2.1 Python

Motivation and alternatives. Maintainability and team adoption were the two dominant drivers for the language choice. We considered **Rust** due to its strong static typing, performance, and ability to produce standalone executables. However, Rust presents a higher barrier to entry for non-software-developer researchers in the team (PhD students and postdocs with mixed backgrounds). This raised concerns about long-term maintainability and the willingness of team members to contribute after the internship.

Why Python. Python was chosen as the best trade-off. It is widely known in research communities, has an extensive ecosystem (libraries for resource monitoring, Docker control, templating, `glscache`, etc.), and accelerates development. The main practical trade-offs are:

- *Distribution:* Python projects do not naturally produce single-file, self-contained binaries. A pip-installable package (and a published PyPI artifact) was therefore the most pragmatic distribution option. This can limit use on locked-down servers where installing packages is restricted; we mitigate this with Docker-based options (see below).
- *Type safety:* Python’s dynamic typing is looser than Rust’s guarantees. To reduce the risk of runtime errors due to missing or malformed data, I used static typing⁷, explicit use of `Optional`⁸, targeted runtime validation, and unit tests.
- *Ecosystem benefits:* Mature libraries simplified implementation: for resource/process monitoring, we used `psutil`; for Docker control, the official Python Docker SDK; for templated reports `Jinja2`; for readable terminal logs and progress `rich`; and for caching `diskcache`. This shortened the development time significantly.
- *Build and CI/CD:* A GitHub Actions pipeline runs tests on each push and pull request, reports status on pull requests, and automates build and publication for tagged releases.⁹ This provides continuous feedback on project health and simplifies releases.

Conclusion. Given the project goals (low barrier to contribution, fast iteration, and integration with existing research tooling) Python was the most suitable choice despite its weaker static guarantees. The combination of type hints and tests aimed to balance agility and robustness. Python’s packaging ecosystem also facilitates distribution and collaboration.

⁷This ensures the value has the correct type

⁸This ensures we handle the absence of a value

⁹See [15] for the CI/CD documentation.

4.2.2 Command-line interface (CLI)

Design goals. The CLI had to be flexible for both interactive use and automation in pipelines. The interface was therefore designed around small, composable subcommands (e.g., `batch-tamarin run`, `batch-tamarin check`, `batch-tamarin report`, `batch-tamarin init`) rather than a single monolithic command.

TUI vs. rich terminal output. Initially a full TUI similar to `lazygit`¹⁰ or `lazydocker`¹¹ was considered to ease interactive configuration. In practice, simple text-based commands augmented with rich terminal output (via the `rich` library) gave the best compromise: users can script commands, and interactive runs remain readable thanks to colorized logs, progress bars, and structured summaries. Implementing a TUI would have increased complexity and maintenance difficulty, so it was abandoned in favour of a text experience. A TUI remains possible in the future because the command structure and executor abstraction are modular.

Modularity and discoverability. Dividing functionality into subcommands improves discoverability and testing: each command can be documented, exercised in CI, and replaced independently. This structure also makes it straightforward to add new front-ends (for example, a thin web UI) that reuse the same core APIs.

4.2.3 Nix

Role of Nix. Nix was adopted as a development and build tool to maximize reproducibility. In simple terms, *reproducibility* means that another person can rerun the same experiment later and obtain the same results because the tools and their versions are identical. A single command (entering a Nix shell) prepares the development environment with pinned versions of Python, build tools, and native dependencies [16]. This avoids the usual "works on my machine" problems and simplifies the onboarding of new contributors.

Interaction with non-Nix users. Although Nix is the preferred developer environment, a classical Python development workflow (`virtualenv`, `pip`, requirements files) is supported for contributors who do not use Nix. This dual support minimizes friction for the widest possible set of users.

Building images and cross-platform artifacts. Nix proved particularly useful to build Docker images with predefined Tamarin versions and their native dependencies. Using Nix for image construction reduced surprises caused by system-level differences and ensured that images used in experiments contain the exact toolchain expected [16]. This was a challenging aspect, as the Tamarin Prover has many dependencies. The building process of Tamarin is long and pretty complex, so Nix's reproducibility is a significant advantage and was my only option for a reliable build process for old Tamarin versions. The final building solution involved getting a precise commit of `nixpkgs`.

¹⁰<https://github.com/jesseduffield/lazygit>

¹¹<https://github.com/jesseduffield/lazydocker>

4.2.4 Docker

Motivation. Docker images provide a clean and practical mechanism to distribute multiple, mutually incompatible versions of `tamarin-prover` on the same host without polluting the system installation. They also enable running experiments on machines where installing Tamarin directly is undesirable or impossible [14]. Put plainly, a Docker container is like a lightweight, portable box that carries the exact tools and versions needed for a run.

Integration challenges. Supporting both local and containerized execution required reworking the execution pipeline of the `batch-tamarin` project. Concretely, there is a need for abstraction with two implementations: a `LocalExecutor` (invokes a locally installed `tamarin-prover`) and a `DockerExecutor` (runs the prover inside a Docker `glsc` container and maps permissions). There was also a need for adapting logging, timeout handling, and resource monitoring so that both backends produce consistent outputs.

Tamarin-prover contribution. To make `batch-tamarin` capable of running any Tamarin version on any computer, I prepared *official* `tamarin-prover` container images that **cover versions 1.4.0 through 1.10.0** with **multi-architecture** support (both `linux/arm64` and `linux/amd64` a.k.a. `x86_64`). In practice, this means a single image tag selects the right binary for each machine automatically (Docker’s “manifest list” mechanism¹²), so `batch-tamarin` can run the exact prover version on any supported computer without manual installation or dependency conflicts.

Benefits and trade-offs. The Docker approach yields strong reproducibility and easy multi-version management at the cost of additional complexity when debugging (containers add an extra layer) and slightly higher resource overhead. Nonetheless, for reproducible research and for reviewers reproducing experiments, this trade-off is strongly favorable [14].

4.2.5 Summary

Overall, the design choices aimed to maximize the utility of `batch-tamarin` for researchers: favouring a language and ecosystem that lowers the barrier to contribution (Python), providing a scriptable and discoverable CLI, and using Nix and Docker to ensure reproducible environments and multi-version execution. Where the chosen technologies introduced risks (dynamic typing, dual execution paths), mitigations such as static typing, tests, and a well-defined execution architecture were put in place to reduce maintenance difficulty.

¹²A multi-architecture container image index that allows a single image tag to automatically resolve to different platform-specific images (e.g., ARM64, x86_64)

5 Implementation and Extensions

This chapter details the implementation in phases that mirror the chronological evolution of the project. For each step, I describe the requirements, the design decisions, the implementation details, and the eventual challenges encountered, following feature releases.

5.1 Hands-on Exploration Phase

5.1.1 Understanding Tamarin Prover

I began by studying the official Tamarin Prover material (user manual and Book draft) to understand how protocols are modeled and how proofs are driven from the command line. Concretely, I surveyed the structure of `.spthy` models (SPTHY) and the inputs and outputs of the command line interface. From an operational standpoint, I catalogued the most relevant commands for automation: `tamarin-prover --version`, `tamarin-prover test` (sanity checks and environment verification), and the classic `tamarin-prover` command with the combinations of options that we can set when we are working with a model. I paid particular attention to the shape and stability of standard outputs and exit codes, as these govern how a wrapper can distinguish (i) a successful proof, (ii) a counterexample with an attack trace, (iii) an unterminated proof search, (iv) well-formedness errors, or (v) an unexpected error during the model proving. This initial survey provided the baseline for robust process management and future output normalization.

5.1.2 First Python Experiments

My first prototype explored how to call the Tamarin Prover from Python in a way that would be portable and observable. I built a small application using a terminal toolkit (akin to a TUI) to (i) auto-detect available `tamarin-prover` binaries on the system, (ii) run `tamarin-prover test` to validate the installation, and (iii) extract the self-reported version via `tamarin-prover --version`. These first code samples were later useful and reusable in the entire project.

To execute the previous commands reliably, I implemented a minimal *Process Manager* responsible for: spawning child processes, retrieving the standard terminal outputs (`stdout/stderr`), enforcing timeouts, and sampling resource usage (real-time process duration and peak RAM usage). This early component has been reused extensively throughout the project; only minor adjustments were needed to support the different extensions to the core functionalities.

This exploration process has been a crucial foundation, as it helped me to fully understand the Tamarin Prover’s capabilities and informed the design of the prover calling from Python.

5.2 Core system (MVP: Minimum Viable Product)

This section will describe the core features of `batch-tamarin`, which were the minimal requirements for a functional prototype. The result of this phase was presented as my first prototype to the team. It has also been the first published pre-release version of the tool.

5.2.1 The Config File

The main user interface is a configuration file called a *glsrecipe*, written in JSON format. This configuration has three main parts:

1. **Global configuration (`config`)**: output directory, default timeout in seconds, and global resource limits (number of glscpu cores and glsram budget). Resources accept "max" or explicit integer values (e.g., 4 means 4 cores or 4 GB of RAM). Memory also supports percentage-based values: the wrapper computes the RAM as that percentage of the detected maximum memory, rounds the result down to a whole number of gigabytes, and clamps it to at least 1 GB and at most the detected maximum.

The following example illustrates a typical global configuration:

```
1 {  
2   "output_directory": "results/exp-01",  
3   "default_timeout": 3600, // in seconds  
4   "global_max_cores": "max", // or <int> cores  
5   "global_max_memory": "80%" // or "max" or <int> GB  
6 }
```

Listing 5.1: Example of a global configuration.

2. **Tamarin Prover inventory (`tamarin_versions`)**: maps short, human-readable aliases to specific `tamarin-prover` executables. Each entry may reference an absolute or relative file path, or a command resolvable on the system PATH (e.g., `tamarin-prover` for the system-installed prover).

The next example will illustrate a classic Tamarin Prover inventory:

```
1 "tamarin_versions": {  
2   "default": {  
3     "path": "tamarin-prover", // system-installed Tamarin Prover  
4   },  
5   "stable": {  
6     "path": "tamarin-1.10/bin/tamarin-prover" // relative path  
7   },  
8   "legacy": {  
9     "path": "/opt/homebrew/bin/tamarin-prover" // absolute path  
10  }  
11 }
```

Listing 5.2: Example of a Tamarin Prover inventory.

3. **Tasks (tasks):** declares the analyses to execute. Each task specifies the input `.spthy` model, selects one or more prover aliases, and defines an output prefix. Tasks may optionally add CLI options (e.g., `--bound=5`), preprocessor `-D` flags, and resource limits (CPU cores, RAM in GB, and specific timeout). Tasks can also use a prefix filter that translates into the `--prove=lemma_name` option from the Tamarin Prover.

This example will show a complete configuration of tasks with all optional parameters:

```

1 "tasks": {
2   "simple": { // a simple config with all mandatory fields
3     "theory_file": "example/protocol.spthy",
4     "output_file_prefix": "simple",
5     "tamarin_versions": ["default", "stable"]
6   },
7   "task_level_options": { // config with task-level options
8     "theory_file": "example/protocol.spthy",
9     "output_file_prefix": "simple",
10    "tamarin_versions": ["default", "stable"],
11    "tamarin_options": ["--heuristic=S", "-v"],
12    "preprocess_flags": ["KEYWORD1", "KEYWORD2"],
13    "resources": {
14      "max_cores": 3, // overwrite the default : 4 cores
15      "max_memory": 12, // in GB, overwrite the default: 16GB
16      "timeout": 600 // in seconds, overwrite default from global config
17    },
18    "lemmas": [ // just a list of prefixes to filter lemmas
19      {
20        "name": "lemma1"
21      },
22      {
23        "name": "lemma2"
24      }
25    ]
26  },
27  "lemma_level_options": { // config with lemma-level options
28    "theory_file": "example/protocol.spthy",
29    "output_file_prefix": "simple",
30    "tamarin_versions": ["default", "stable"],
31    "tamarin_options": ["--heuristic=S", "-v"],
32    "preprocess_flags": ["KEYWORD1", "KEYWORD2"],
33    "resources": {
34      "max_cores": 3,
35      "max_memory": 12, // in GB
36    },
37    "lemmas": [
38      { // This lemma overwrites the task-level config
39        "name": "lemma1",
40        "tamarin_versions": ["legacy"], // overwrite used provers
41        "tamarin_options": ["--bound=5"], // overwrite task-level options
42        // take the preprocess_flags from task level
43      },
44      { // This lemma uses resource inheritance
45        "name": "lemma2",
46        "resources": {
47          "max_cores": 2, // overwrite the task level
48        } // 12GB RAM limit is inherited from the task level
49      } // Timeout is inherited from the global level (default timeout)
50    ]
51  }

```

Listing 5.3: Example of tasks configurations.

By default, the following resource limits are applied by the wrapper if no resources are specified at the task or lemma level: CPU cores are limited to 4, RAM is limited to 16 GB, and the default timeout from the global config is applied.

When specifying a lemma, there is the possibility to fine-tune the different parameters for this specific lemma. There is a specific inheritance mechanism to make the configuration as concise as possible. Anything specified in the lemma will overwrite the task-level configuration, and anything not specified will be inherited from the task level.

You can find a complete example of configuration in appendix A.

This verbose configuration may appear complex initially, but it provides a flexible framework for fine-tuning that has proven robust in practice. The configuration’s stability has been validated through extensive development, as no structural modifications were required while introducing numerous new features to the tool.

5.2.2 The Core of the Execution Pipeline

The core development focused on a clear execution architecture with single-purpose components and typed data structures. The pipeline is orchestrated by a *runner* that coordinates five managers: configuration, resources, tasks, processes, and outputs.

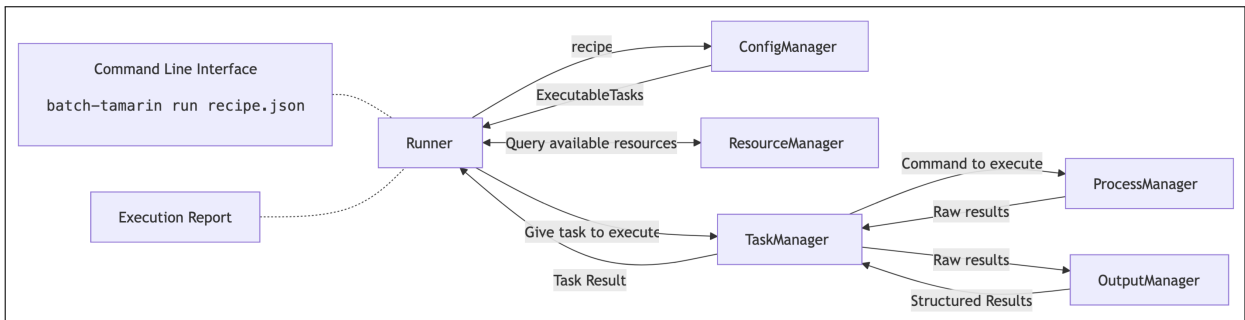


Figure 5.1: High-level execution pipeline: from recipe to normalized results.

Execution flow. Given a recipe, the **Config Manager** validates and materializes it into a list of *Executable Tasks*—concrete invocations of **tamarin-prover** with resolved options and resources. The **Resource Manager** implements admission control based on available CPU/RAM quotas and the chosen scheduling policy (see Section 5.8). For each allocated task, the **Task Manager** delegates to the **Process Manager**, which executes the command, enforces timeouts, and records telemetry. Raw outputs and exit codes are then parsed by the **Output Manager** and serialized to structured JSON ready for aggregation. You can find this architecture in Figure 5.1.

The runner provides live, human-readable progress using rich terminal formatting and concludes with a reproducible summary: global configuration, prover versions used, per-task and per-lemma outcomes, and a compact error report suitable for debugging.

This strong execution basis enables efficient resource utilization and robust error handling, setting the stage for easy future enhancements.

5.3 Lemma Parser & Model Introspection

5.3.1 Context

Initially, the wrapper relied on the `--prove` argument, which proves all lemmas at once, while using `--prove={lemma_name}` to filter lemmas based on the given prefix, leaving Tamarin to iterate over all matching lemmas internally. This limited our ability to (i) report fine-grained progress, (ii) attribute resources to individual lemmas, and (iii) recover gracefully from partial failures. Initially, one would only know that a specific batch of lemmas had been executed by Tamarin, with no details about individual lemmas in the batch. Moreover, precise reporting would have been impossible, as one would only know whether the Tamarin Prover ended its execution with or without an error, but would not have been able to precisely report the results (validated lemma, falsified lemma, and untermiated lemma) for the executed batch.

Parsing `.spthy` files directly to enumerate lemmas is a principled alternative: it enables the wrapper to launch one prover instance per lemma (or per filtered subset), to schedule them independently, and to label outputs unambiguously. Another approach would be to use regex either on Tamarin's terminal output or directly on models, but this is brittle. Although easier to implement, we rejected it due to instability.

In Python, there are two practical ways to obtain an SPTHY parser: (i) reimplement a parser from scratch, or (ii) reuse the Tree-sitter grammar maintained by the Tamarin community. The first option would be long and complex and thus out of scope for this internship. The second option poses integration challenges because the reference Tree-sitter implementation is in C and JavaScript, which complicates direct use from Python.

5.3.2 Implementation

Just before the `Config Manager` creates *Executable Tasks*, it calls the new `Lemma Parser` component that uses a Tree-sitter¹ grammar for the SPTHY language to recover the AST and extract lemma names. The Lemma Parser can take a prefix filter as an optional parameter to restrict the search to specific lemmas. Each matched lemma becomes its own `--prove={lemma_name}` executable unit, which integrates seamlessly with the existing task model. The rest of the execution pipeline remains unchanged, as it already handled per-lemma invocations naturally. You can see Figure 5.2 for a comparison of the configuration processing without and with the parser.

To address the tree-sitter integration challenge, I created a new PyPI project to provide a SPTHY language support package `-py-tree-sitter-spthy`² – which provides a ready-to-use Python binding of the compiled SPTHY language for the tree-sitter parser, like other language bindings already available on PyPI³. This package is generated just by a simple compilation pipeline on GitHub Actions, which compiles the C code of the tree-sitter parser and bundles it with the Python binding. It is based on the code of `vscode-tamarin`⁴, which provides the latest version of the SPTHY grammar for tree-sitter, based on a reverse engineering of the Tamarin Prover's own parser.

¹See glossary entry Tree-sitter.

²Available at <https://github.com/lmandrelli/py-tree-sitter-spthy>

³Here are two examples of official bindings on PyPI: <https://pypi.org/project/tree-sitter-go/>, <https://pypi.org/project/tree-sitter-rust/>

⁴Available at <https://github.com/tamarin-prover/vscode-tamarin>

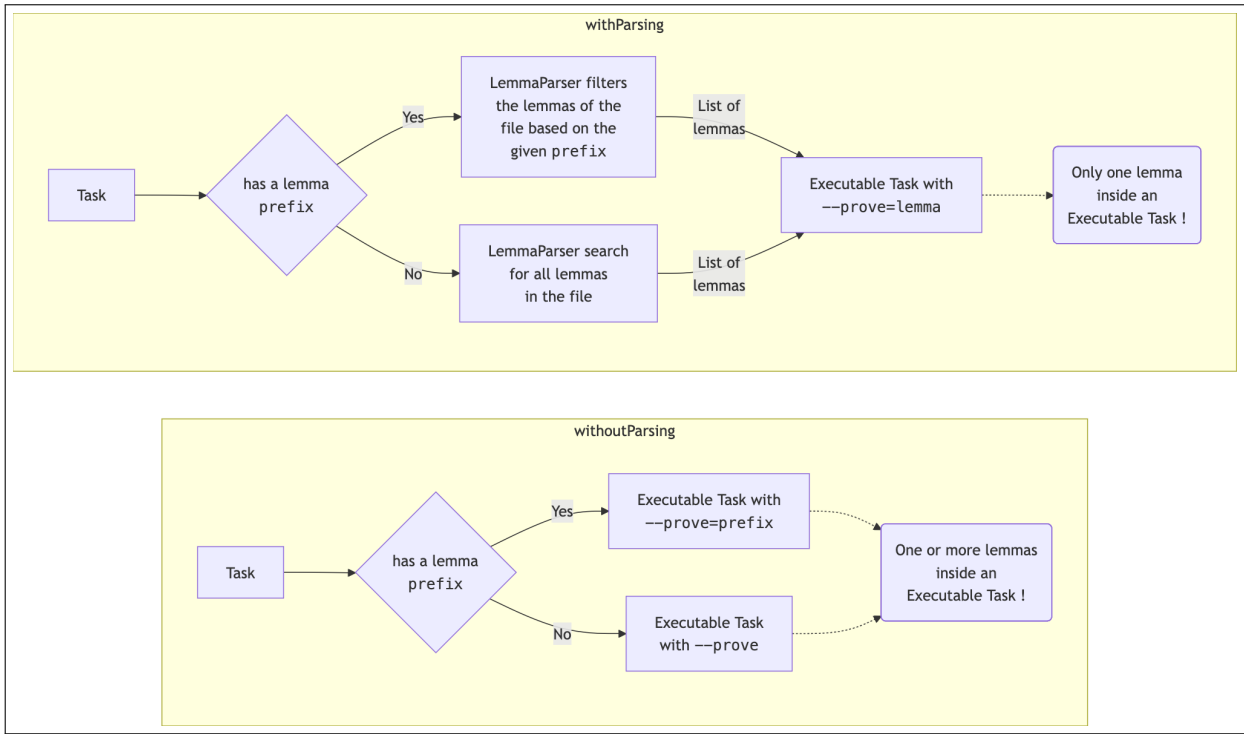


Figure 5.2: Effect of the parser on configuration processing

The parser is a core component as it is used in the further extensions of the project. It could also be extended in the future to support more advanced parsing features, like lemma type recognition, model modifications, or even error recovery.

5.4 Preflight Validation

5.4.1 Motivation

During model development and configuration authoring, fast feedback is crucial. Users need to validate that models are well-formed and that the recipe expands into the intended set of lemmas before launching long runs. A dedicated **check** command provides a quick, safe dry-run to catch misconfigurations early. This also helps to ensure that every specified Tamarin Prover is working as intended. A paper reviewer will also be able to use this command to quickly verify that the provided files and configuration are complete and valid before running the full experiment.

5.4.2 Implementation

The **check** command reuses the same configuration loading as the **run** pipeline; it directly reuses the **ConfigManager**, this naturally reuses the parser and JSON validation from this component, but a modification was necessary to skip the directory creation done during the configuration reading. It invokes the Tamarin Prover once per input model without the **--prove** parameter to detect well-formedness errors reported by Tamarin. You can see the Figure 5.3 for a complete overview of this command.

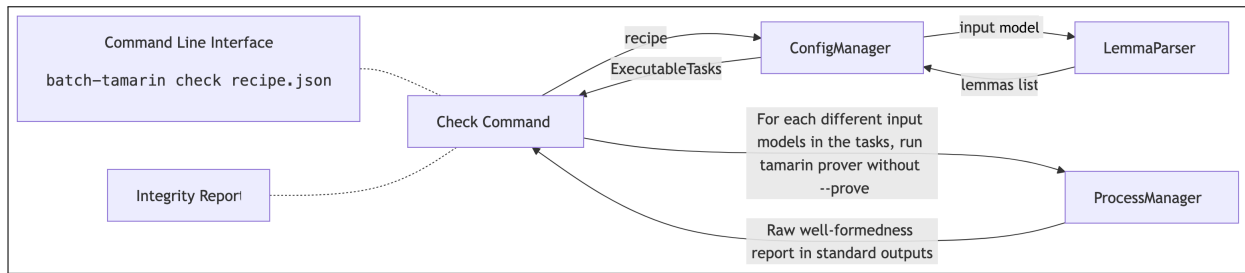


Figure 5.3: Architecture of the `check` command

The output is a concise table summarizing: global settings; resolved prover versions with integrity reports and self-reported versions; the list of tasks and lemmas that would run with all effective configuration details; and any detected model well-formedness issues. Given that we parse Tamarin’s standard output, we can reliably detect the presence of errors but not always their precise cause. Users should still invoke Tamarin directly to diagnose model errors in detail.

The goal of this command is to make it straightforward to iterate on a recipe until it is consistent, while localizing errors to their source in a structured way.

5.5 Interactive Recipe Generator

5.5.1 Motivation

As noted above, the configuration file is verbose and can be intimidating for new users. Small mistakes are reported during configuration loading, but writing a correct recipe from scratch can still be frustrating. Moreover, because there are numerous options, documentation is essential. The `init` command addresses these issues: instead of starting from an empty file, users interactively create a valid configuration by answering a series of prompts in the terminal (after providing the input models as arguments).

5.5.2 Implementation

`init` is implemented as a thin interactive layer over the Pydantic models used by the *Config Manager*. It constructs a recipe object in memory, validates it with the same schema as `run` and `check` commands, and serializes it to JSON. When the user wants to specify a lemma for a task, it leverages the Lemma Parser to immediately test a lemma prefix against the provided input model and show the user which lemmas match the prefix filter.

Because the Pydantic model needs to be valid and is shared through the different commands, any changes to the model done for a command need to be reflected in the `init` command as well. This implies maintenance over the questions that create the user prompts, ensuring the questions reflect the current state of the model.

5.5.3 Results

The research team was highly satisfied with this command, finding it to be a significant improvement over hand-writing configuration files. The initial experience with `batch-tamarin` without this command required learning the configuration structure and available options, whereas using the command is intuitive and straightforward. Using the `init` command, it is possible to quickly generate a valid configuration without copying and pasting from example configurations. My personal experience with this command has also been very positive, as I was able to create additional example configurations easily with this tool instead of copying previous ones.

5.6 Caching & Attack-Trace Export

5.6.1 Problem and Motivations

Traces. From version 1.10 onward, Tamarin can emit attack traces on demand. These traces are valuable for later analysis; without automated support, one would have to re-run long batches with additional options to obtain them. An alternative solution would be to write the option explicitly in the configuration file. Adding the option is trivial, but compatibility is not: the feature is unavailable on older versions. To keep `batch-tamarin` compatible with 1.4, we needed a mechanism to prevent incompatible parameters from being used with older versions. **Caching.** A second impactful enhancement to the wrapper is the caching system. Up to here, repeated executions of identical tasks are wasteful: Tamarin proofs can be slow, up to several hours for complex lemmas. Consider two common scenarios: a lemma fails in a batch, and we want to rerun it while modifying only this particular lemma, or we previously stopped the execution of a batch and now want to resume it. In both cases, without caching, we would have to run all tasks from scratch, wasting time and resources. Caching addresses these two problems by allowing us to rerun only the failed and modified lemma, or by skipping execution of previously completed tasks and resuming from where we left off.

5.6.2 Implementation

Traces. I extended command construction with the Tamarin Prover parameter `---output-json` and `---output-dot` to write traces to a file. To resolve the compatibility problem, I added a new utility module, called at the end of the command generation done by the `Config Manager`, that automatically removes all the unsupported parameters of a command, given the prover version. This module contains a mapping of unsupported parameters per Tamarin version, which is easy to extend in the future if needed. This way, users can also specify any parameters in the configuration, without thinking about the compatibility issues.

Caching. Caching is managed by a new module, the `Cache Manager`. This module is invoked by the `Task Manager` to avoid modifying the existing architecture and to integrate seamlessly into the workflow. The `Task Manager` always checks the cache before running an Executable Task to retrieve previously cached results. The cache uses the `diskcache`⁵ Python library and stores results via a dictionary-like interface in the standard local cache directory. It includes all previously executed tasks and their results; it works with a global scope and does not limit cache searches based on the input configuration file. The cache key hashes only inputs that semantically affect results: (i) the input model file content, (ii) the selected prover executable path, (iii) the normalized CLI options, (iv) the preprocessor flags, and (v) the effective resource limits (timeout, CPU, RAM). Interrupted runs are distinguished from failures and do not poison the cache. File artifacts (proofs, logs, traces) are restored from cache, keeping the output directory layout stable. The main challenge was ensuring that all information was cached and restored correctly, and choosing the most appropriate inputs to define the cache key.

5.7 Reporting & Rerun Recipes

5.7.1 Motivation

The raw JSON output is not optimal for the user to read and interpret. While it is machine-readable and a programmer can easily extract information from it, the goal is to provide a user-friendly and human-readable output. The wrapper originally produced a report in the terminal at the end of execution, but this was a limited summary, and the terminal output is not intended to display a lot of information. The goal of this extension is to produce a complete report, portions of which are suitable for inclusion in research artifacts or papers to facilitate documentation and result sharing. A requirement for the reports was to maintain access to the raw data produced by the wrapper with the `run` command, while the report generation would be handled by a new `report` command that would process the previously generated raw data to produce the desired report. This approach allows users to run lengthy experiments and generate reports later without rerunning the entire experiment, or to produce reports in different formats. To be able to be used in different contexts, the report should be available in multiple formats: HTML, Markdown, LaTeX, and Typst.

An additional expected feature was to be able to easily re-run failed lemmas or lemmas that exhausted their resources. The report should include a recipe enabling users to re-run only the failed lemmas with the same configuration as the original run, while allowing selective parameter changes (e.g., increasing timeout or RAM limits for specific lemmas).

5.7.2 The challenges

The initial data flow transformed objects across modules, shedding details needed at the end of the execution to generate the reports. To get these details back, we needed to reassemble the information from various sources, which was non-trivial. This also required a lot of time and effort to implement a new data structure that would contain all the information needed to generate the report. This new data structure had to be designed carefully to ensure that it was both comprehensive and wouldn't break any previously made components. It would sometimes

⁵Available at <https://grantjenks.com/docs/diskcache/>

involve conversion between different data formats or structures.

A second challenge was LaTeX reporting. Jinja2 templating is not the easiest tool for generating LaTeX documents. The main difficulty was table generation: LaTeX requires specific syntax and packages, and NiceTabular doesn't support multi-page tables. Multi-page-capable alternatives often lose the formatting that makes tables readable. Producing a good LaTeX template took roughly a week, and the final template still lacks some features present in the other formats.

5.7.3 Design Choice

I introduced a top-level batch data model that mirrors the entire lifecycle of a `run`: inputs, scheduling decisions, task/lemma telemetry, outputs, and errors. The model is serialized automatically at the end of execution, enabling offline report generation and easy re-runs based on previous results (e.g., increasing RAM for specific lemmas that exhausted memory). It takes back similar data structures from the execution context, ensuring all components can interact with this new model. In a general overview, only the Runner component needed a deep refactor, as other components could use converted data. This challenge required two weeks of work to completely implement and test the new run pipeline.

The rerun recipe can be easily deduced from this model, as it contains all the information needed to reconstruct the configuration of a specific lemma. The only needed computation is to filter the wanted lemmas and copy the relevant configuration.

5.7.4 Implementation

Reporting uses Jinja2 templates to render HTML, Markdown, LaTeX, and Typst variants. At the end of the execution pipeline, we now add three new outputs:

- A summary report in HTML format, a plain copy of the terminal summary, the main purpose is to have an overview of the execution outside of the terminal.
- A rerun recipe in JSON format, allowing users to easily re-execute failed lemmas. It copies all the global configuration and the lemma-specific configuration to have an easy way to modify the parameters.
- A complete JSON with all the execution details, which is the serialized batch data model. This file is the main input of the `report` command.

The `report` command takes the complete JSON file as input and generates a comprehensive report in the requested format. The report includes: (i) a summary of the global configuration and prover versions used, (ii) timelines of task and lemma executions with resource usage, (iii) per-lemma comparisons across prover versions with outcomes and telemetry, and (iv) an appendix of errors encountered. A user could then use this report to understand the execution better, share results with collaborators, and include tables or graphs from the report directly in a paper or artifact. You can find some samples of a generated example report in appendix B.

5.8 Scheduling Policies

5.8.1 The problem

The initial scheduling strategy prioritized jobs requiring the least resources (CPU and RAM). This is efficient but can lead to unexpected behavior when users expect a different policy. Rather than imposing a single policy, users should be able to choose among strategies. The most relevant strategies for this wrapper—based on the different information we have and the constraint that a launched task should not be preempted—are:

- FIFO: First-In-First-Out, tasks are executed in the order of the configuration file, if the next task in the file can be fitted within the resource limit, we can allocate it.
- SJF: Shortest Job First, tasks with the smallest resource requirements are prioritized.
- LJF: Longest Job First, tasks with the largest resource requirements are prioritized.

5.8.2 The solution

The three strategies can be chosen with a CLI parameter when launching the run pipeline (e.g., `--scheduler sjf`). The Resource Manager implements all three. FIFO simply iterates over pending tasks in order, whereas SJF and LJF sort pending tasks by resource requirements. I kept a list-based allocation: allocate the next task in the list if it fits the resource limits; pre-sort the list during Resource Manager initialization based on the selected strategy.

The figure 5.4, 5.5, 5.6 illustrates the different policies through the timelines generated by the report command.

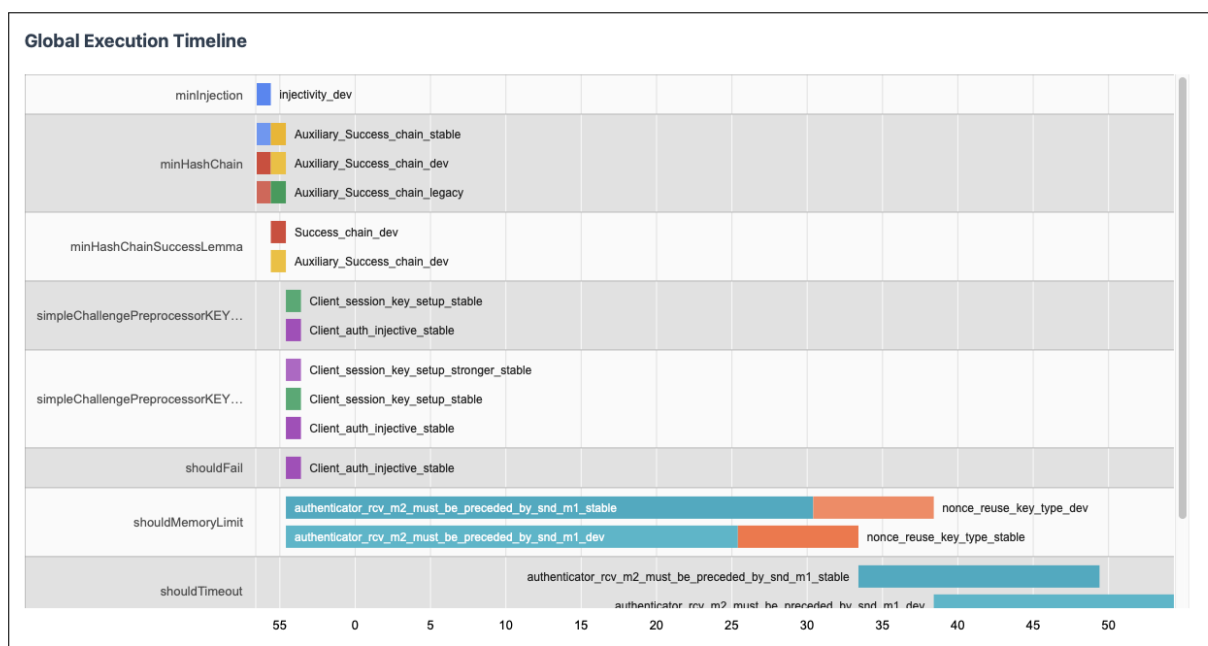


Figure 5.4: FIFO scheduling timeline

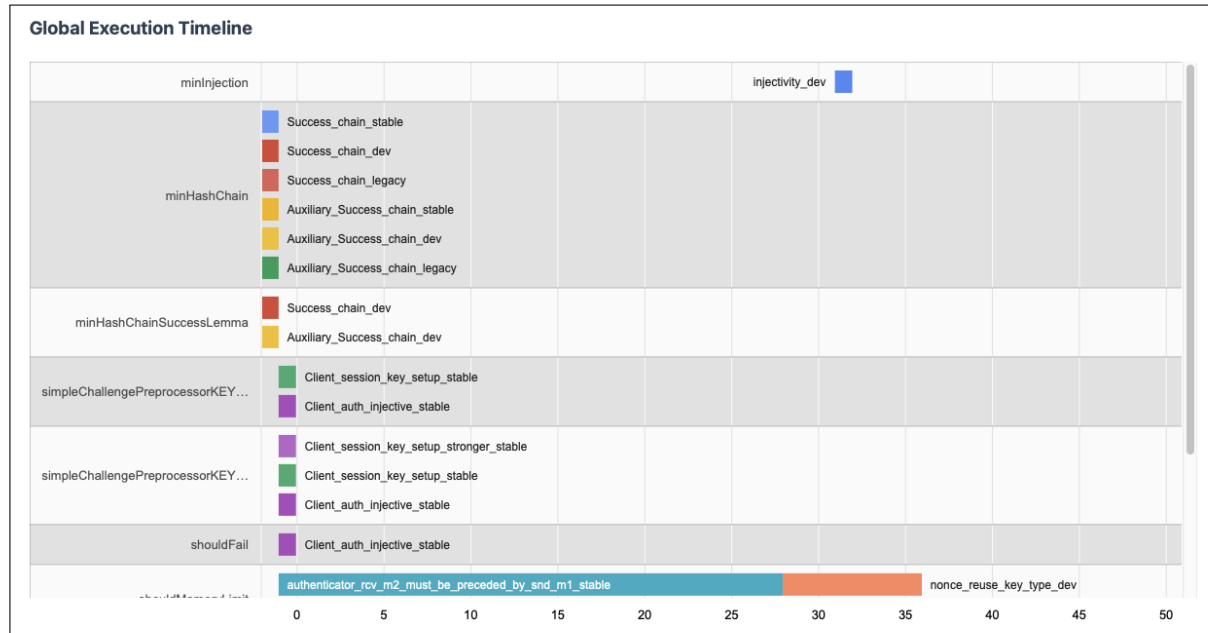


Figure 5.5: SJF scheduling timeline

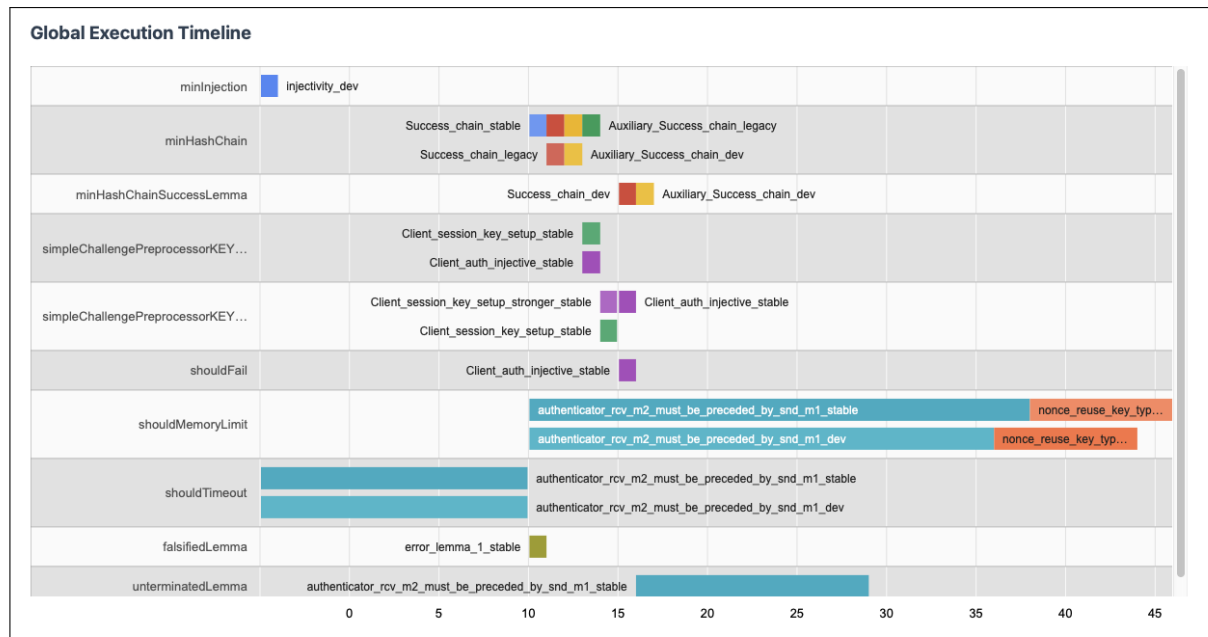


Figure 5.6: LJF scheduling timeline

6 Results and Outlook

6.1 Results

The tool has already supported a paper under submission by the research team. Using only a few commands, we reproduced a full experiment on a modest model entirely via containerized execution, ensuring clean environments and eliminating “it works on my machine” issues. The Markdown report was reused verbatim in the artifact repository to present outcomes.

The `batch-tamarin` project has been accepted as an official repository within the Tamarin Prover community, which simplifies discovery and fosters contributions.

6.2 Future Work

Several improvements are planned. Beyond incremental usability refinements, the most impactful items are:

- **Parser-based modifications:** Modify the model with a richer configuration to be able to disable lemmas or modify them like the ANSSI wrapper.
- **Enhanced reporting:** Improve the reporting capabilities to include more detailed statistics, and improve the LaTeX template to be on par with the other formats.
- **CLI improvements:** Enhance the CLI to report progress in a more user-friendly manner.
- **Containerized execution:** The project integrates cleanly with Docker via provided Dockerfiles for multiple Tamarin versions. A possible improvement is to let `batch-tamarin` pull and run a Docker image on demand, eliminating any local installation. This is non-trivial and left as work in progress. The intended reviewer workflow must remain simple; adding an extra orchestration layer may not yield clear time savings unless carefully scoped.

6.3 Personal Reflection

This internship offered a complete view of building a research tool from first principles and aligning it with a community's needs. I practiced scoping, prioritizing features, and estimating development effort under real-world constraints. Technically, I deepened my understanding of protocol verification and automated reasoning while honing software engineering skills—typing, testing, reproducible builds, and packaging. Most importantly, I learned how close collaboration with researchers shapes pragmatic design choices that measurably improve reproducibility.

7 Concluding Remarks

Summary. I have presented `batch-tamarin`, a Python-based orchestration layer for the Tamarin Prover designed for reproducibility, scalability, and usability. Starting from exploratory prototypes, the work converged on a modular pipeline driven by a declarative recipe and incrementally added: a lemma parser, a configuration checker, an interactive initializer, a version-aware trace exporter, a principled cache, a reporting engine, and a configurable scheduler.

Contributions. The main contributions are: (i) a unified execution architecture with typed data models; (ii) reproducible multi-version execution via Docker images and Nix-pinned builds; (iii) per-lemma scheduling and telemetry enabled by parsing SPTHY; (iv) end-to-end reporting and re-run recipes that streamline large experiments; and (v) community artifacts, notably precompiled SPTHY Tree-sitter bindings.

Limitations. The current design targets a single host and relies on OS scheduling for fairness across processes. While adequate for many labs, very large experiments would benefit from distributed back-ends. Debugging inside containers, although feasible, remains less convenient than fully local execution.

Outlook. Future directions include remote executors, richer provenance capture (e.g., automatic embedding of exact prover commit hashes), and lightweight analytics for proof-search behavior. With an active community and established release processes, the project is well-positioned to support reproducible protocol verification at scale.

Bibliography / Webography

- [1] D. Basin, Cas J. F. Cremers, Jannik Dreier, and R. Sasse. Tamarin: Verification of large-scale, real-world, cryptographic protocols. *IEEE Security & Privacy*, 20:24–32, 2022. 1
- [2] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5g authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’18, page 1383–1396, New York, NY, USA, 2018. Association for Computing Machinery. 1
- [3] Bruno Blanchet. Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif. In *Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif*, volume 1 of *Foundations and Trends® in Privacy and Security*, pages 1 – 135. Now Foundations and Trends, October 2016. 1
- [4] CISA Helmholtz Center for Information Security. About cisa. Website, 2025. Available at: <https://cisa.de/en/about> (accessed 2025-08-11). 3
- [5] CISA Helmholtz Center for Information Security. Cisa receives top rating in international helmholtz evaluation 2025. Press release, June 2025. Available at: <https://cisa.de/en/evaluation> (accessed 2025-08-11). 3
- [6] CISA Helmholtz Center for Information Security. Cisa world leader in computer security. News, 2025. Available at: <https://cisa.de/en/cisa-world-leader> (accessed 2025-08-11). 4
- [7] CISA Helmholtz Center for Information Security. Cisa’s latest conference contributions (ndss 2025 and iclr 2025). Website, 2025. Available at: <https://cisa.de/> (accessed 2025-08-11). 4
- [8] CISA Helmholtz Center for Information Security. Research areas. Website, 2025. Available at: <https://cisa.de/en/research/research-areas> (accessed 2025-08-11). 4
- [9] Cas Cremers, Alexander Dax, and Aurora Naska. Formal analysis of SPDML: Security protocol and data model version 1.2. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6611–6628, Anaheim, CA, August 2023. USENIX Association. 1
- [10] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of tls 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’17, page 1773–1788, New York, NY, USA, 2017. Association for Computing Machinery. 1
- [11] Cas Cremers, Charlie Jacomme, and Aurora Naska. Formal analysis of Session-Handling in secure messaging: Lifting security from sessions to conversations. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1235–1252, Anaheim, CA, August 2023. USENIX Association. 1

- [12] Cas Cremers, Benjamin Kiesel, and Niklas Medinger. A formal analysis of IEEE 802.11's WPA2: Countering the cracks caused by cracking the counters. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1–17. USENIX Association, August 2020. 1
- [13] Cas J. F. Cremers. Symbolic security analysis using the tamarin prover. *2017 Formal Methods in Computer Aided Design (FMCAD)*, pages 5–5, 2017. 1
- [14] Docker, Inc. Docker: Get started overview. Documentation, 2025. Available at: <https://docs.docker.com/get-started/> (accessed 2025-08-13). 12
- [15] GitHub, Inc. Github actions documentation. Documentation, 2025. Available at: <https://docs.github.com/en/actions> (accessed 2025-08-13). 10
- [16] NixOS Foundation. Nix reference manual (stable). Manual, 2025. Available at: <https://nixos.org/manual/nix/stable/> (accessed 2025-08-13). 11

List of Figures

5.1	High-level execution pipeline: from recipe to normalized results.	16
5.2	Effect of the parser on configuration processing	18
5.3	Architecture of the <code>check</code> command	19
5.4	FIFO scheduling timeline	23
5.5	SJF scheduling timeline	24
5.6	LJF scheduling timeline	24
B.1	Configuration overview	38
B.2	Execution summary	39
B.3	Execution summary graphics	39
B.4	Global result table	40
B.5	Execution timeline	40
B.6	Task detail	41
B.7	Rendered traces	42
B.8	Error detail	42

Listings

5.1	Example of a global configuration.	14
5.2	Example of a Tamarin Prover inventory.	14
5.3	Example of tasks configurations.	15
A.1	Example recipe demonstrating configuration, prover inventory, and two tasks. . .	36
B.1	Commands to reproduce the complete example.	38

Glossary

API Application Programming Interface: a set of definitions and protocols that allow different software applications to communicate with each other 11

AST Abstract Syntax Tree: a tree representation of the syntactic structure of source code, used by compilers and parsers to understand program structure 9, 17

attack trace A step-by-step sequence showing how an attacker can violate a security property; serves as a concrete counterexample when a lemma cannot be proven 1, 13

CI/CD Continuous Integration/Continuous Delivery: development practices that involve frequent code integration & automated deployment pipelines 10

CLI Command-Line Interface: a text-based user interface for interacting with computer programs through typed commands 8, 25, 43

Docker A containerization platform that packages applications & their dependencies into lightweight, portable containers 11, 12, 25, 27, 43

Dolev–Yao model A symbolic attacker model where the adversary has complete control over network communication (can eavesdrop, block, replay, & compose messages) while cryptographic primitives are assumed to be perfect 1, 43

exit code Numeric value returned by a program to indicate success or type of failure; used by batch-tamarin to distinguish different execution outcomes 1

FIFO First In, First Out: a scheduling algorithm that processes tasks in the order they were submitted to the queue 23, 43

heuristic Problem-solving strategy that guides proof search but doesn't guarantee optimal solutions; used to navigate the large search space in automated theorem proving 1

HTML HyperText Markup Language: the standard markup language used to create web pages and web applications 21, 22, 43

JSON JavaScript Object Notation: a lightweight, text-based data interchange format that is easy for humans to read and write 14, 19, 43

lemma In Tamarin, a security property or claim about a protocol that needs to be proven true or false; represents formal statements about authentication, confidentiality, or other security goals 8

LJF Longest Job First: a scheduling algorithm that prioritizes tasks with the largest resource requirements or longest expected execution time 23, 43

multiset-rewrite rules Formal specification method where system states are represented as bags of facts & transitions are modeled as rules that consume & produce facts 1, 43

Nix A package manager & build system that enables reproducible & declarative package management & system configuration 11, 27, 43

nixpkgs The comprehensive package collection for the Nix ecosystem; pinning a specific commit hash ensures reproducible & deterministic dependency versions across different machines 1, 11

Prover A software tool that automatically attempts to construct mathematical proofs or find counterexamples for given properties; in this context, refers to the Tamarin Prover *vii*

PyPI Python Package Index: the official third-party software repository for Python programming language packages 7, 10, 17

research artifact Supplementary materials (code, data, configurations) accompanying research papers to enable reproducibility & verification of results 1

scheduler Component that decides execution order of tasks based on available resources & chosen priorities; implements policies like FIFO, SJF, or LJF 1

SJF Shortest Job First: a scheduling algorithm that prioritizes tasks with the smallest resource requirements or shortest expected execution time 23, 43

SPTHY Security Protocol Theory language: the input language used by Tamarin Prover to model security protocols, facts, rules, and lemmas 13, 17, 27, 43

symbolic model Abstract representation of cryptographic protocols where cryptographic operations are treated as perfect black boxes, ignoring computational complexity 1

telemetry Automatic collection & transmission of performance & resource usage data during execution; enables detailed analysis of proof attempts 1

timeout Maximum time allowed for a computation before it's forcibly terminated; prevents infinite execution of complex proofs 1

Tree-sitter An incremental parsing framework that builds concrete syntax trees for source code; provides robust parsing capabilities with error recovery & incremental updates 1, 17

TUI Text-based User Interface: an interface that uses text, symbols, and pseudo-graphics to provide a more visual interaction than CLI while remaining text-based 11, 13

UI User Interface: the means by which users interact with and control a computer program or system 8, 9, 11

well-formedness Whether a model follows correct syntax & semantic rules; determines if a Tamarin model is valid & can be analyzed 1

Appendices

A Appendix A

Appendix A — Example Recipe.

```
1 {
2   "config": {
3     "output_directory": "results/exp-01",
4     "default_timeout": 3600, // in seconds
5     "global_max_cores": "max", // or <int> cores
6     "global_max_memory": "80%" // or "max" or <int> GB
7   },
8
9   "tamarin_versions": {
10     "default": {
11       "path": "tamarin-prover", // system-installed Tamarin Prover
12     },
13     "stable": {
14       "path": "tamarin-1.10/bin/tamarin-prover" // specific path (relative
15       from pwd)
16     },
17     "legacy": {
18       "path": "/opt/homebrew/bin/tamarin-prover" // absolute path (here a
19       fictive homebrew installation)
20     }
21   },
22   "tasks": {
23     "simple": { // a simple config with all mandatory fields
24       "theory_file": "example/protocol.spthy",
25       "output_file_prefix": "simple",
26       "tamarin_versions": ["default", "stable"]
27       // by default :
28       // no tamarin options
29       // no preprocessor flags
30       // default resources :
31       //   max cores: 4
32       //   max memory: 16GB
33       //   timeout: 3600, taken from global config
34     },
35     "task_level_options": {
36       "theory_file": "example/protocol.spthy",
37       "output_file_prefix": "simple",
38       "tamarin_versions": ["default", "stable"],
39       "tamarin_options": ["--heuristic=S", "-v"],
40       "preprocess_flags": ["KEYWORD1", "KEYWORD2"],
41       "resources": {
42         "max_cores": 4,
43         "max_memory": 16, // in GB
44         "timeout": 600 // in seconds, overwrite default
45       }
46     },
47     "lemmas": [ // this lemma list will just filter lemmas with the different
```

```

45     prefixes (if the prefix is the same, there is duplication)
46     {
47         "name": "lemma1"
48     },
49     {
50         "name": "lemma2"
51     }
52 ],
53 "lemma_level_options": {
54     "theory_file": "example/protocol.spthy",
55     "output_file_prefix": "simple",
56     "tamarin_versions": ["default", "stable"],
57     "tamarin_options": ["--heuristic=S", "-v"],
58     "preprocess_flags": ["KEYWORD1", "KEYWORD2"],
59     "resources": {
60         "max_cores": 4,
61         "max_memory": 16, // in GB
62     },
63     "lemmas": [
64         {
65             // For this first lemma, we show parameters that can be completely
66             // overwritten
67             "name": "lemma1",
68             "tamarin_versions": ["legacy"],
69             "tamarin_options": ["--bound=5"],
70             "preprocess_flags": ["KEYWORD3"],
71         },
72         {
73             // For this second lemma, the resources are the only parameters
74             // that have inheritance
75             "name": "lemma2",
76             "resources": {
77                 "max_cores": 2, // overwrite the task level
78                 // take 16GB from task-level config
79                 // As no timeout was set for this task and lemma, we use the
80                 // default timeout from the config
81             }
82         }
83     ]
84 }

```

Listing A.1: Example recipe demonstrating configuration, prover inventory, and two tasks.

B Appendix B

Appendix B — Report Samples. This appendix presents parts of a complete example report, generated by batch-tamarin. This example is reproducible by running:

```
1 batch-tamarin run examples/complete.json
2 batch-tamarin report -f tex complete-example-results/
```

Listing B.1: Commands to reproduce the complete example.

The following figures are screenshots from the generated HTML report:

This first section (B.1) begins with a reminder of the loaded configuration:

Overview				
Configuration				
Global Settings		Tamarin Versions		
Setting	Value	Alias	Path	Version
Max cores	10	stable	tamarin-binaries/tamarin-prover-1.10/1.10.0/bin/tamarin-prover	v1.10.0
Max memory	32GB	dev	tamarin-binaries/tamarin-prover-dev/1.11.0/bin/tamarin-prover	v1.11.0
Default timeout	80s	legacy	tamarin-binaries/tamarin-prover-1.8/1.8.0/bin/tamarin-prover	v1.8.0

Figure B.1: Configuration overview

It continues with a table (B.2) that provides statistics about the execution, with associated graphics (B.3):

Global Summary			
Global Statistics			
Metric	Value	Metric	Value
Total Executed Lemmas	23	Total runtime	137.6s
Verified Lemmas	16 (69.6%)	Total peak memory used	4.1GB
Falsified Lemmas	1 (4.3%)	Max peak memory used	1.0GB
Unterminated Lemmas	1 (4.3%)	Freshly executed tasks	23 (100.0%)
Failed Executions	1 (4.3%)	Cache hits	0 (0.0%)
Lemmas killed by timeout	2 (8.7%)		
Lemmas killed for memory limit	2 (8.7%)		

Figure B.2: Execution summary

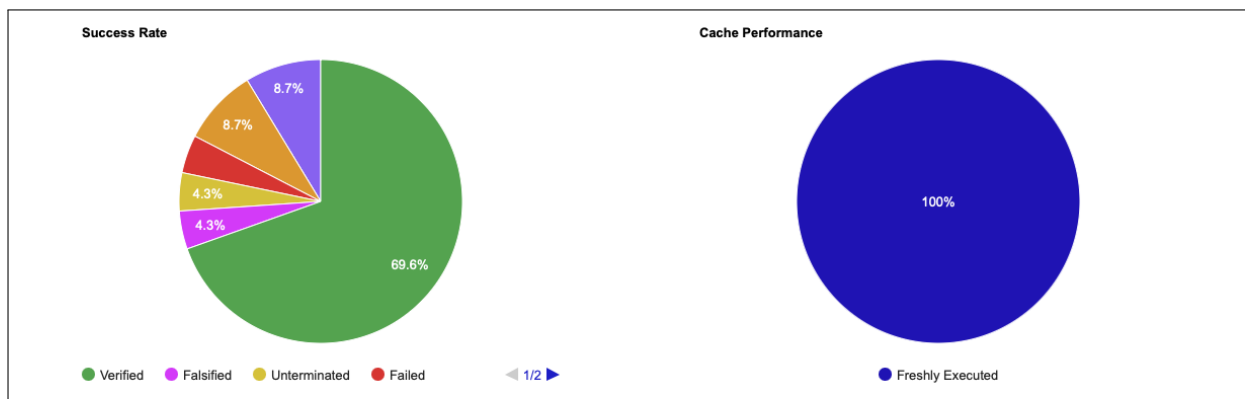


Figure B.3: Execution summary graphics

The second section is the global result table and the execution timeline (B.4, B.5):

Global Results						
Task	Extracted lemma	Tamarin Version	Status	Runtime	Peak Memory Used	Cache Hit
minInjection examples/___protocols___/MinimalInjectiveFact.spthy	injectivity	dev	✓ Verified	1.4s	31.6MB	No
minHashChain examples/___protocols___/Minimal_HashChain.spthy	Success_chain	stable	✓ Verified	1.5s	0.2MB	No
		dev	✓ Verified	1.5s	0.2MB	No
		legacy	✓ Verified	1.5s	2.9MB	No
	Auxiliary_Success_chain	stable	✓ Verified	0.4s	0.2MB	No
		dev	✓ Verified	0.4s	0.2MB	No
		legacy	✓ Verified	0.4s	12.0MB	No
minHashChainSuccessLemma examples/___protocols___/Minimal_HashChain.spthy	Success_chain	dev	✓ Verified	0.4s	30.2MB	No
	Auxiliary_Success_chain	dev	✓ Verified	0.4s	0.8MB	No
simpleChallengePreprocessorKEYWORD1 examples/___protocols___/SimpleChallengeResponse.spthy	Client_session_key_setup	stable	✓ Verified	0.4s	0.2MB	No
	Client_auth_injective	stable	✓ Verified	0.4s	0.2MB	No

Figure B.4: Global result table

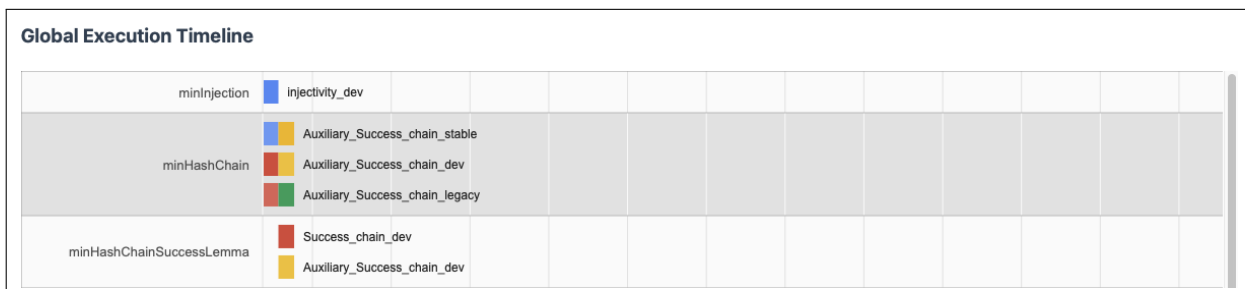


Figure B.5: Execution timeline

The next section gives task-by-task details, with per-lemma outcomes and telemetry (B.6). In case a lemma has dot graphics associated, they will be rendered with the details (B.7).

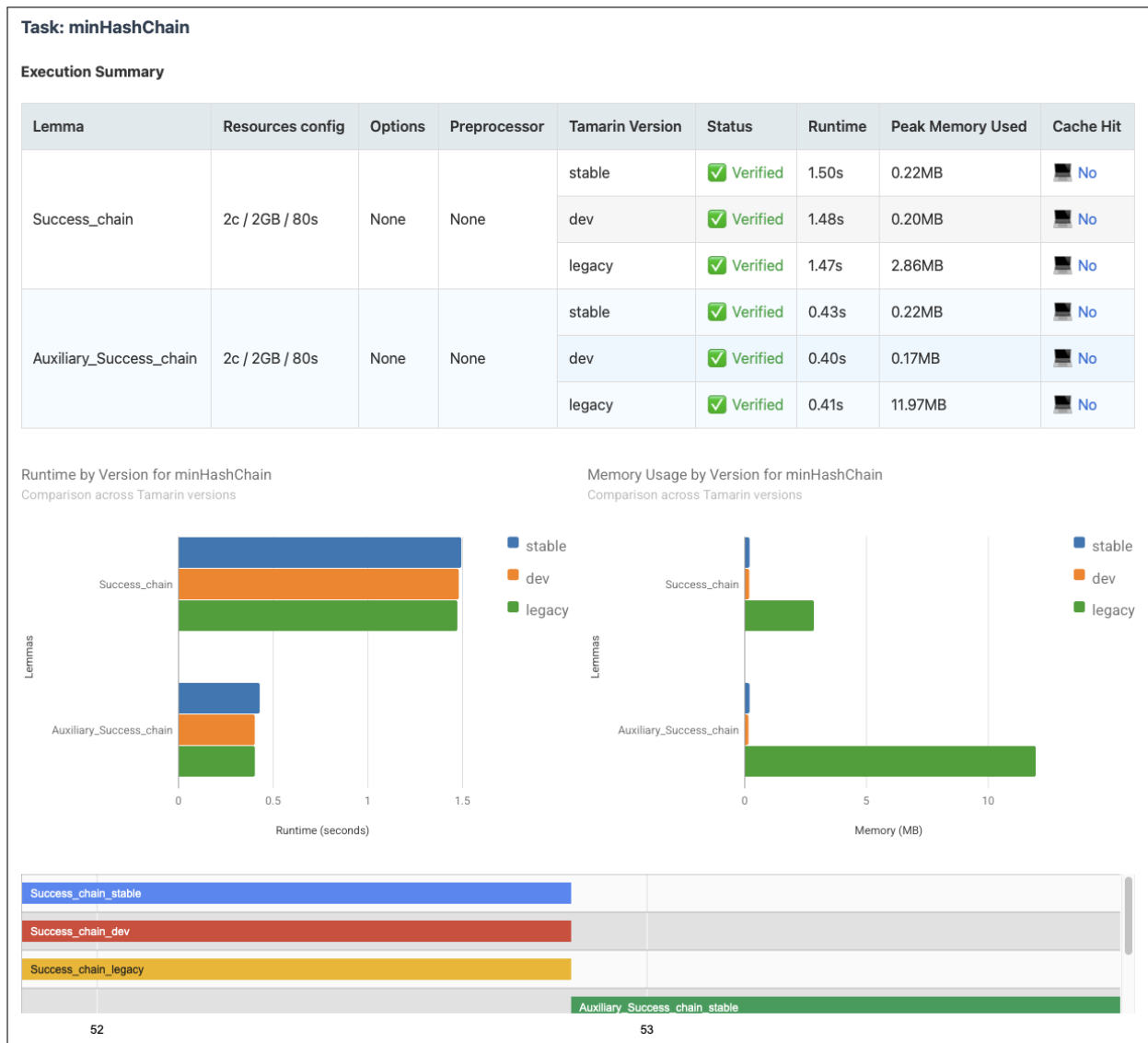


Figure B.6: Task detail

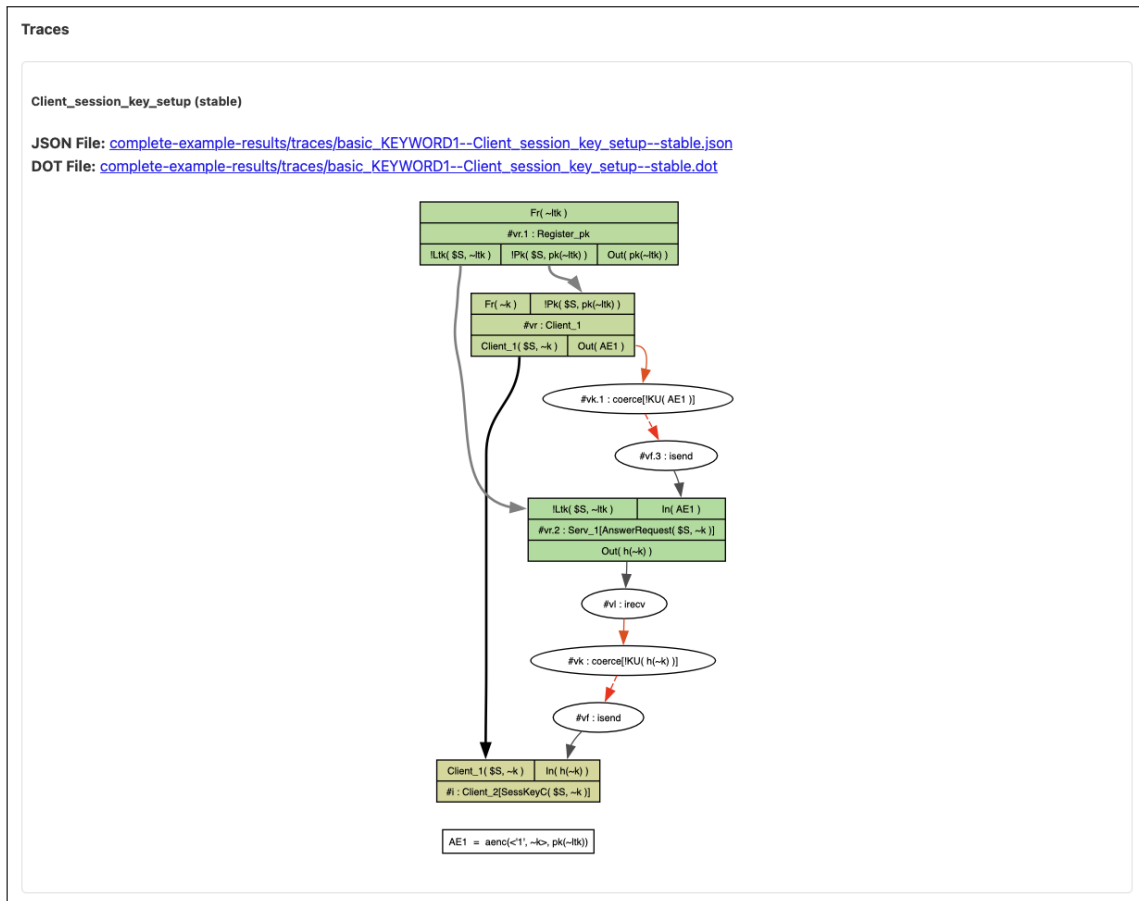


Figure B.7: Rendered traces

Finally, the last section is an appendix of errors encountered during the execution (B.8).

Detailed Error Information

TAMARIN ERROR : simpleChallengePreprocessorKEYWORD1

On lemma : Client_auth_injective, with tamarin-prover : stable

Task failed with return code 1

```
maude tool: 'maude'
checking version: 3.4. OK.
checking installation: OK.
"examples/__protocols__/SimpleChallengeResponse.spthy" (line 68, column 1):
unexpected "#"
expecting "heuristic", "tactic", "builtins", "options", "functions", "function", "equations", "macros", "restriction", "axiom", "test", "lemma",
```

Figure B.8: Error detail

Résumé

Ce rapport présente `batch-tamarin`, un outil Python destiné à l'orchestration reproductible de lots d'analyses avec Tamarin Prover. Tamarin Prover est un outil de vérification formelle des protocoles de sécurité dans le modèle symbolique (de type Dolev–Yao model), fondé sur des règles de multiset-rewrite rules ; il permet d'établir des preuves de propriétés ou, à défaut, de produire des traces d'attaque concrètes. Après une phase d'exploration, nous décrivons une architecture modulaire pilotée par une « recette » déclarative en JSON. Les fonctionnalités ajoutées au fil du projet comprennent : un parseur SPTHY pour l'exécution et la télémétrie au niveau des lemmes ; une commande `check` pour la validation rapide ; un générateur interactif `init` ; l'export de traces d'attaque avec gestion des compatibilités de version ; un cache sémantique ; un moteur de rapports multi-formats (HTML, Markdown, LaTeX, Typst) ; et un ordonnanceur configurable (FIFO, SJF, LJF). Nous discutons les choix d'ingénierie (Python, Nix, Docker), les compromis, et présentons des résultats sur un cas réel. L'outil réduit le recours à des scripts ad hoc, améliore la comparabilité des expériences et renforce la reproductibilité au sein de la communauté Tamarin.

Mots-clés : vérification formelle, Tamarin Prover, orchestration, reproductibilité, Nix, CLI.

Abstract

This report presents `batch-tamarin`, a Python tool for reproducible orchestration of large-scale analyses with the Tamarin Prover. The Tamarin Prover is a formal verification tool for security protocols in the symbolic (Dolev–Yao) model, based on multiset-rewrite rules; it can either establish proofs of properties or produce concrete attack traces. We describe a modular execution pipeline driven by a declarative JSON “recipe,” then detail successive extensions: a SPTHY lemma parser enabling per-lemma scheduling and telemetry, a fast `check` command, an interactive `init` generator, version-aware attack-trace export, a semantic cache, a multi-format reporting engine (HTML, Markdown, LaTeX, Typst), and a configurable scheduler (FIFO, SJF, LJF). We discuss engineering choices (Python, Nix, Docker), trade-offs, and results on real models. The tool reduces ad hoc scripting, improves comparability across experiments, and strengthens reproducibility for the Tamarin community.

Keywords : formal verification, Tamarin Prover, orchestration, reproducibility, Nix, CLI.