MASTA: A Feedback-Scheduled Multi-Agent System for End-to-End Tamarin Modeling and Analysis

Yongkang Xiao Qiyi Deng Jing Chen* Min Shi* Ju Ma Ruiying Du
School of Cyber Science and Engineering, Wuhan University
{xiaoyongkang,qiyideng,chenjing,itachi,juma_526,duraying}@whu.edu.cn

Abstract

Tamarin-based formal verification of security protocols requires expert-authored multiset rewriting rules and temporal-logic lemmas. Translating a natural-language protocol description into a complete formal model with an LLM is hard because syntactic, reachability, and property-level errors compound across the multi-step modeling process, and a single agent that both authors and self-evaluates cannot reliably localize them.

We present MASTA, a multi-agent system that addresses these difficulties through three innovations. First, an *operational–evaluative* role decomposition assigns rule modeling to an Architect, property modeling to an Auditor, and pure dispatch to a dedicated Arbiter that holds no authoring duties. Second, the Arbiter implements a Tamarin-anchored scheduler that dispatches the next agent based on the latest verifier verdict and routes problem-tagged fixes through a modification list, with a six-class problem taxonomy and an LLM deduplication check preventing routing deadlock. Third, we construct a benchmark of 10 protocols paired with 139 mutation variants and five fine-grained metrics that jointly assess model correctness (M_1–M_3) and completeness (M_4–M_5).

On this benchmark, MASTA on Qwen 3.5-plus improves over OpenCode on the same backbone by up to $+0.27$ on individual metrics and ranks first on both the correctness and completeness averages, surpassing Codex (GPT-5.5) and Claude Code (Opus 4.7) despite using a weaker LLM. Ablations on five protocols confirm that removing the Arbiter collapses property and verification scores to zero.

1 Introduction

Security protocols form the backbone of modern networked systems, and flaws at the design level can have practical impact on deployed systems. For example, flaws in 5G authentication can enable subscriber tracking [3], while weaknesses in EMV contactless payment protocols can allow transactions to bypass cardholder verification [4]. Formal verification provides a rigorous method for analyzing protocol security by exhaustively exploring all possible protocol executions under a specified adversary model. The Tamarin prover [21] is a widely used tool for formally verifying security protocols under the Dolev–Yao adversary model [11]. It has been used to verify protocols such as TLS 1.3 [8], EMV contactless payments [4], and 5G AKA [3], and has uncovered attacks in these deployed protocols. However, effective use of Tamarin requires substantial domain expertise: users must translate an informal protocol description into multiset rewriting rules, capture the intended adversary model and cryptographic abstractions, and formalize security requirements as lemmas. This expertise barrier limits Tamarin's practical reach to a small community of specialists, reducing the extent to which formal verification is applied to deployed protocols.

*Corresponding authors.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

Recent advances in large language models (LLMs) have demonstrated strong code-generation capabilities [6, 17, 16] and shown emerging potential in formal reasoning tasks, such as theorem proving [26, 12]. A natural question is *whether LLMs can automate Tamarin-based protocol verification*. Prompting an LLM to produce a complete Tamarin model faces two challenges. First, the generated Tamarin model must simultaneously satisfy executability, syntactic legality, and lemma-level semantic correctness, because Tamarin's domain-specific language imposes structural constraints, including well-sortedness of terms, correct use of persistent versus linear facts, and proper placement of action facts, that differ fundamentally from general-purpose programming. Empirical evaluations of LLMs on formal-specification inference tasks [15] report that the conjunction of such constraints degrades model quality. Second, protocol modeling is an incremental multi-step process in which each added rule may break executable reachability and each added lemma may expose a defect in earlier rules, so end-to-end generation followed by a single final verification concentrates errors at the last stage and obscures their origin. A naive remedy that lets a single agent generate each step and self-evaluate it fuses the two activities within one context, exhibiting the same generator/evaluator coupling found in single-agent self-reflection frameworks [22, 19]. Process-level supervision and a structurally independent evaluator are therefore both required.

We address these challenges with MASTA (Multi-Agent System for Tamarin-based Analysis). In MASTA, an Architect is responsible for modeling the protocol and the adversary as Tamarin multiset rewriting rules, verifying that each rule preserves executability before proceeding. An Auditor is responsible for extracting security requirements from the protocol description and encoding them as formal lemmas. An Arbiter monitors the Tamarin verification results after each authoring step, decides which agent should act next, and routes corrective tasks back to the responsible agent whenever a problem is detected. This design structurally separates generation from evaluation: the Arbiter's scheduling decisions are grounded in Tamarin's verdicts rather than the generating agents' own assessment, eliminating the self-evaluation coupling that undermines single-agent approaches.

Our contributions are summarized as follows.

- We introduce an end-to-end LLM-based multi-agent framework for Tamarin-based security protocol verification. The framework transforms a natural-language protocol description into a complete Tamarin-compatible formal model under tool-anchored process supervision. The generated model comprises multiset rewriting rules that capture both the protocol behavior and the Dolev–Yao adversary capabilities, together with first-order temporal-logic lemmas that formally specify the protocol's security requirements.
- We propose a role decomposition in which formal-model authoring is distributed across an Architect and an Auditor, while a dedicated Arbiter handles workflow scheduling exclusively. The Arbiter dispatches the next agent based on the latest Tamarin verdict and routes corrective tasks back to the responsible agent. This design separates generation from evaluation, eliminating the self-evaluation coupling of single-agent approaches by construction.
- We construct a benchmark of 10 security protocols, together with their operational and vulnerability mutations, yielding 149 modeling tasks in total. We also design five metrics that jointly assess modeling accuracy (whether the generated rules and lemmas agree with expert references) and security completeness (whether the model correctly reflects protocol modifications and detects introduced vulnerabilities).

The full source code, the benchmark, and the experimental logs are released via a public repository [1] to support reproducibility.

The remainder of this paper is organized as follows. Section 2 reviews related work on automated protocol formalization, LLM-based formal methods, and LLM agents for protocol formalization. Section 3 introduces the necessary Tamarin background. Section 4 presents the MASTA architecture and the three-agent collaboration mechanism. Section 5 defines the benchmark and the five evaluation metrics. Section 6 reports the main comparison and ablation results, and Section 7 concludes.

2 Related Work

Automated protocol formalization tools. A line of non-LLM work narrows the gap between informal protocol descriptions and the inputs of symbolic protocol verifiers [21, 9] by introducing higher-level surface languages. SAPIC [14] compiles applied-pi-calculus processes into Tamarin's

multiset rewriting rules, relieving users of authoring rules directly. Verifpal [13] provides a readable surface syntax that compiles to ProVerif and supports passive/active attacker models. Alice-and-Bob-style specification languages [2] let users describe protocols at the message-exchange level, with automated translation to Tamarin or other backends. These tools all reduce the symbolic-modeling burden, but they require users to supply structured semi-formal specifications.

LLM-based formal methods and tool-augmented generation. LLM-assisted formal methods include theorem proving over pre-existing formal artifacts [26, 12, 24] and evaluation of specification synthesis under controlled mutations [15]; all of these either assume the formal model already exists or evaluate generation in a one-shot setting without verifier feedback. A complementary line augments LLM agents with external tool signals: Self-Debug [7] iteratively patches code using compiler and test output, and SWE-agent [25] designs agent-computer interfaces that let an LLM manipulate files and run tests to fix real-world software issues. Self-correction frameworks such as Reflexion [22] and Self-Refine [19] use verbal critiques but keep the critic and generator within the same agent.

LLM agents for protocol formalization. CryptoFormalEval [10] benchmarks LLMs on vulnerability detection using Alice-and-Bob notation as input rather than natural language. AutoSM [20] generates symbolic models for Tamarin and ProVerif with formal soundness guarantees, but requires human-in-the-loop interaction and is not fully automated. MASTA is fully end-to-end: it takes natural-language protocol descriptions as input and automatically produces complete formal models without human intervention.

Comparison with SAPIC and AutoSM. Table 7 in Appendix H contrasts MASTA with SAPIC and AutoSM. SAPIC is a translator, so the modeling step we automate remains the user's responsibility. AutoSM is the closer comparison, and the key difference is autonomy: its Repairer stage requires user interaction, whereas MASTA runs end to end using only Tamarin verdicts. The evaluations also differ in what they can detect. AutoSM reuses the reference model's lemmas and checks whether they still verify on the generated model, which corresponds to one of our five metrics (M_3, verification consistency). Because MASTA generates the lemmas itself, we also evaluate their structural equivalence to the reference (M_2) and the fidelity of the modeled message flow (M_1). In addition, because our benchmark includes 139 mutations, we can assess whether subtle protocol variations are modeled faithfully (M_4, M_5), which a benchmark consisting of one model per protocol cannot capture.

Summary. Across all three lines of related work, a common gap remains: no existing system takes a natural-language protocol description and produces a Tamarin-verifiable formal model in a fully automated, process-supervised pipeline. Automated formalization tools require structured semi-formal input rather than prose. LLM-based formal methods either assume a pre-existing formal artifact or generate in a one-shot setting without verifier feedback. Multi-agent and self-correction frameworks do not externalize the evaluative role to an independent agent, leaving generation and assessment within the same context. MASTA addresses all three gaps simultaneously by combining natural-language input, a role-decomposed multi-agent architecture, and step-level Tamarin feedback as the grounding signal for cross-agent scheduling and fix-job dispatch.

3 Preliminaries

Tamarin. Tamarin [21] specifies a protocol as a set of multiset rewriting rules of the form

$$[L] \xrightarrow{A} [R],$$

where L and R are multisets of facts representing the rule's premises and conclusions, and A is a multiset of action facts recorded on the execution trace. Linear facts in L are consumed on application; persistent facts are retained. Security properties are expressed as lemmas: first-order temporal-logic formulas over the sequence of action facts. For example, a secrecy lemma takes the form

$$\forall\, x, i.\ \texttt{Secret}(x) \mathbin{@} i \implies \neg\exists j.\ \texttt{K}(x) \mathbin{@} j,$$

asserting that any value marked `Secret` at trace position i is never learned by the adversary at any position j. For each lemma, Tamarin returns one of two verdicts: `proved` or `falsified`.

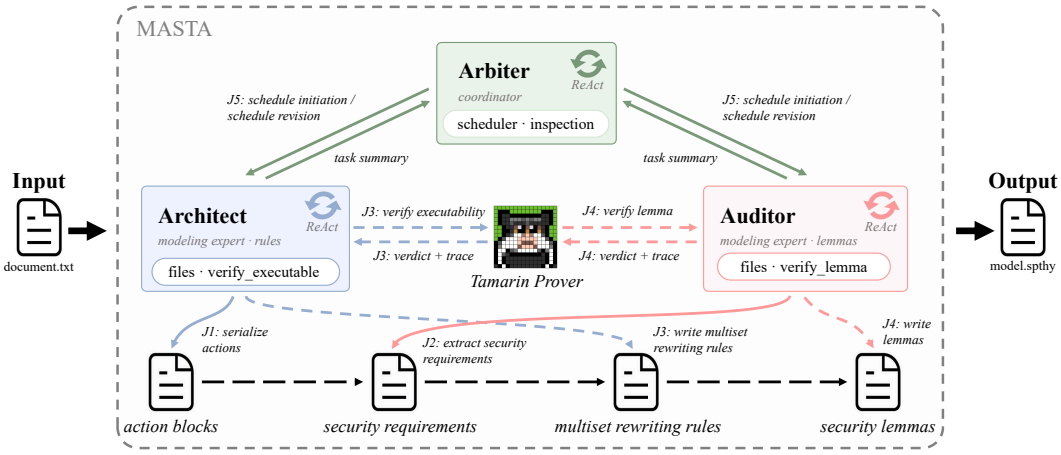


Figure 1: Architectural overview of MASTA: three agents (Architect, Auditor, Arbiter) sharing file I/O and Tamarin invocation; four operational jobs (J1–J4) and one continuous scheduling job (J5); the Arbiter dispatches the next agent based on the latest Tamarin verdict, including fix-job routing, under a one-hour per-protocol budget.

For evaluation purposes, we additionally use the label `timeout` when Tamarin fails to return either verdict within the wall-clock budget, covering both the explicit cutoff and cases where its proof-search heuristics fail to terminate.

Adversary model. Tamarin natively embeds the Dolev–Yao attacker [11], the standard network adversary who controls the communication channel and can intercept, replay, drop, and compose messages from observed terms. This baseline is extended on a per-protocol basis to model additional adversary capabilities such as long-term key compromise, which are expressed as extra Tamarin rules that the Architect adds to the formal model.

Action blocks. We introduce *action blocks* as a structured intermediate representation that bridges natural-language protocol descriptions and Tamarin rules. An action block is attributed to a single participant role and contains an ordered sequence of five operation types: RECEIVE, GENERATENONCES, COMPUTE, ASSERTION, and SEND. The operations within a block are totally ordered in this fixed sequence, and an action-block sequence $A = (b_1, b_2, \ldots, b_n)$ captures the full message-exchange skeleton of the protocol. Each block corresponds to one Tamarin multiset rewriting rule. The formal definition of operation types and the block equality predicate used by M_1 are given in Appendix A.

4 Method

4.1 Overview

Figure 1 illustrates the overall architecture of MASTA. The system processes one target protocol per execution and instantiates three agents: Architect, Auditor, and Arbiter. Each agent operates on its own system prompt, and all agents share two capabilities: file read/write and Tamarin invocation. Additionally, only the Arbiter possesses a workflow-scheduling capability. The system performs five distinct jobs (Table 1): four operational jobs (J1–J4) that generate partial formal models, and one continuous scheduling job (J5) that coordinates agents to accomplish the first four jobs. Input to the agent system is provided as a text file, while the output of each run is a Tamarin formal model consisting of multiset rewriting rules and first-order temporal-logic lemmas.

4.2 Job Specifications

Two intermediate artifacts support the authoring agents. A *set of action blocks* captures the message-exchange skeleton of the protocol as a sequence of role-attributed receive, fresh-nonce, compute,

Table 1: The five jobs of MASTA. J1–J4 are operational jobs that produce formal models; J5 is a continuous scheduling job whose only output is a dispatch instruction.

Job	Owner	Input	Output
J1 Action serialization	Architect	Protocol document	Structured action blocks
J2 Security-requirement extraction	Auditor	Protocol document	Structured security requirements
J3 Rule modeling	Architect	Action blocks	Tamarin rules + exec. verdicts
J4 Property modeling	Auditor	Security requirements	Tamarin lemmas + lemma verdicts
J5 Workflow scheduling	Arbiter	State + verdicts	Dispatch instruction

and send operations, with each action block corresponding to one Tamarin rule template, without committing to Tamarin syntax. A *structured security requirement* specifies the security assumptions together with the targeted secrecy [11] and authentication [18] properties between protocol participants. The four operational jobs are organized so that authoring is interleaved with Tamarin checkpoints, localizing faults to the step that introduced them.

J1 (Architect): action serialization. The Architect parses the protocol document into a sequence of action blocks, fixing the message-exchange order before any Tamarin syntax is written.

J2 (Auditor): security-requirement extraction. The Auditor parses the same protocol document into a structured security requirements document. J2 depends only on the protocol document, so it could run in parallel with J1; the default sequence runs it after J1 so that jobs never overlap.

J3 (Architect): rule modeling. The Architect translates the action blocks into Tamarin multiset rewriting rules. After each rule is added, Tamarin is invoked on an exists-trace "executable" lemma to confirm that an honest run remains reachable; if the verdict is `falsified`, the rule is revised before the next rule is added. This per-rule check ties reachability faults to the rule that introduced them.

J4 (Auditor): property modeling. The Auditor writes Tamarin lemmas from the structured security requirements and then issues a single batched Tamarin call once all lemmas are written. Batching the verification, rather than calling Tamarin per lemma, allows shared restrictions and action-fact conventions to be checked once and amortizes the per-call startup cost.

J5 (Arbiter): workflow scheduling. The Arbiter is the sole authority over which agent acts next. On each iteration, it inspects the system state for issues exposed by the latest Tamarin verdict, selects either the next normal-sequence job or a fix-job for the agent responsible for the affected artifact, and tags the dispatch with a problem-type label drawn from a fixed six-class taxonomy. Before issuing the dispatch, the Arbiter runs an LLM deduplication check against the modification list of past dispatches to prevent reissuing the same direction under different wording; on rejection, a deterministic AdjustDirection procedure cycles through the remaining labels and admissible target artifacts. A run terminates either with a complete formal model, with a *timeout* when the one-hour budget is exhausted, or with a *stuck state* when AdjustDirection finds no new direction. The full pseudocode is given as Algorithm 1 in Appendix C.

4.3 Agent Prompt and Context Design

All three agents share a six-layer prompt template, assembled in a fixed order on every turn:

1. **Output Protocol**: the JSON schema for each LLM response.
2. **Role Description**: the agent's identity.
3. **Capability Description**: the list of callable capabilities.
4. **Responsibilities Description**: the agent's per-phase workflow and output formats.
5. **Context**: the history of prior `(action, result)` pairs in the run.
6. **Current Task**: the immediate goal set by the Arbiter at each dispatch.

Layers 1–4 are fixed per agent type, while layers 5–6 evolve across turns. The full template is reproduced in Appendix D.

5 Benchmark and Metrics

To evaluate MASTA, we need both a measurement scheme that decomposes model quality into observable, comparable quantities and a benchmark whose inputs and references support that scheme. This section introduces five metrics (M_1–M_5) that jointly quantify the correctness and the completeness of a generated Tamarin model (Section 5.1), and describes how the 10-protocol benchmark and its 139 mutation variants were constructed to exercise each of the five metrics (Section 5.2).

5.1 Metrics

Correctness metrics (M_1–M_3). The first three metrics assess whether the generated model is *correct* with respect to the reference.

M_1 (process fidelity) captures how faithfully the agent reproduces the protocol's message-exchange structure. A protocol run is modeled as a sequence of action blocks, each tied to one participant role and containing an ordered list of receive, generate, compute, and send operations. M_1 counts the minimum number of insertions, deletions, and substitutions needed to transform the agent-produced sequence A into the reference sequence B, and then normalizes by the length of the longer sequence. Two action blocks are equal when they share the same role and their operation lists are identical under alpha-equivalence, treating function symbols as constants and allowing only bound variables to be renamed.

$$M_1 = 1 - \frac{\mathrm{EditDistance}(A,\, B)}{\max(|A|,\, |B|)} \in [0, 1].$$

M_2 (property consistency) captures whether the agent writes security lemmas that are structurally equivalent to the reference lemmas. Directly checking logical equivalence between two Tamarin lemma formulas is undecidable in general, so we instead provide the agent with a canonical lemma template that fixes quantifier structure, action-fact naming, variable naming, and clause order. Because the agent is required to follow this template, every generated lemma can be compared to the corresponding reference lemma by direct abstract-syntax-tree matching. Let the reference contain N lemmas. Each generated lemma is compared against its reference counterpart and receives a score of 1 if the two are alpha-equivalent as syntax trees and 0 otherwise. M_2 is the mean score over all N reference lemmas.

$$M_2 = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}[r_i \text{ is alpha-equivalent to its generated counterpart}] \in [0, 1].$$

M_3 (verification consistency) captures whether running Tamarin on the generated model yields the same verdicts as on the reference model. A naive binary agreement score is insufficient for `proved` lemmas, because a vacuously true model in which the protocol is never actually executed would still receive `proved` from Tamarin. To penalize such false positives, we use the number of proof steps as a proxy for protocol execution depth. Let $\hat{v}(l)$ and $v^*(l)$ denote the agent and reference verdicts for lemma l, and let $\hat{n}(l)$ and $n^*(l)$ denote the corresponding proof-step counts. A verdict mismatch receives score 0. A matching `proved` verdict receives $\min(1, \hat{n}(l)/n^*(l))$, which is less than 1 when the agent proof uses fewer steps than the reference, indicating a potentially vacuous proof. All other matching verdicts receive score 1. M_3 is the mean score over all K lemmas.

$$s(l) = \begin{cases} 0 & \hat{v}(l) \neq v^*(l) \\ \min\left(1,\, \dfrac{\hat{n}(l)}{n^*(l)}\right) & \hat{v}(l) = v^*(l) = \texttt{proved} \\ 1 & \text{otherwise} \end{cases}$$

$$M_3 = \frac{1}{K} \sum_{l=1}^{K} s(l) \in [0, 1].$$

Completeness metrics (M_4–M_5). Correctness metrics evaluate the standard-model output but do not test whether the system faithfully captures *variant* protocol behaviors. Inspired by the mutation-based evaluation of FormalBench [15], we design two additional metrics to assess model completeness under protocol variation.

M_4 (operational-mutation detection) tests whether the agent correctly reflects deliberate but security-preserving changes to the protocol. Each operational-mutation variant o carries a set of mutation points $\mathcal{P}(o)$, where each point is a subsequence of action blocks that must appear in the generated sequence. A mutation point $p \in \mathcal{P}(o)$ is considered detected if every action block in p appears in the agent-produced sequence under the M_1 equality predicate. Let $\mathcal{O}$ be the set of all operational-mutation variants of a protocol. M_4 is the fraction of mutation points detected across all variants.

$$M_4 = \frac{\sum_{o \in \mathcal{O}} \sum_{p \in \mathcal{P}(o)} \mathbf{1}[p \text{ is detected}]}{\sum_{o \in \mathcal{O}} |\mathcal{P}(o)|} \in [0, 1].$$

M_5 (vulnerability-mutation detection) tests whether the agent correctly captures introduced security flaws. Each vulnerability-mutation variant u is associated with one or more security lemmas whose expected verdict is `falsified`. A variant u is considered detected if Tamarin returns `falsified` on at least one of those lemmas when run on the generated model. Both a Tamarin `timeout` and a modeling error that prevents verification count as undetected. Let $\mathcal{U}$ be the set of all vulnerability-mutation variants of a protocol. M_5 is the fraction of variants correctly detected.

$$M_5 = \frac{1}{|\mathcal{U}|} \sum_{u \in \mathcal{U}} \mathbf{1}[u \text{ is detected}] \in [0, 1].$$

5.2 Dataset

Protocol collection. We collected Tamarin formal models from the official Tamarin prover repository [23] and through broader GitHub search, retaining only models for which Tamarin verification terminates automatically within ten minutes. For each retained model, we manually authored a natural-language protocol description in the style of an RFC document, so that the system input is a standalone prose document with no formal content.

Mutation construction. For operational mutations, we modify the protocol description to change the message-exchange structure while preserving security, and provide the corresponding reference Tamarin model and action-block sequence. For vulnerability mutations, we introduce a security flaw into the description and provide the reference model in which at least one security lemma returns `falsified`. All variant descriptions and reference files were produced with LLM assistance and verified by a human expert. The benchmark contains 10 standard protocols together with 101 operational and 38 vulnerability mutation variants, totaling 149 input documents.

6 Experiments

6.1 Setup

We compare MASTA against three general-purpose coding agents: Codex (GPT-5.5), Claude Code (Opus 4.7), and OpenCode (Qwen 3.5-plus). MASTA also uses Qwen 3.5-plus, making the OpenCode–MASTA pair a controlled comparison where only the agent architecture varies. All four systems share the same skill set wrapping the four operational jobs (J1–J4) and the same one-hour per-protocol time budget. Each baseline follows the default sequence J1→J2→J3→J4 and self-evaluates after every job, revising its own output on failure without dispatching to another agent. Tamarin version, hardware, and decoding hyperparameters are listed in Appendix D. The full source code, the 10-protocol benchmark with all 149 modeling tasks, the per-run experimental logs, and the evaluation scripts that produce the M_1–M_5 tables below are released in a public repository [1].

6.2 Main Comparison

Table 2 reports the per-metric M_1–M_5 scores for every system on the 10-protocol evaluation set. Because correctness and completeness are not directly additive, we report Avg(Corr) = mean(M_1, M_2, M_3) and Avg(Comp) = mean(M_4, M_5) instead of an unweighted mean over M_1–M_5.

Table 2: Main comparison on the 10-protocol evaluation set ($n = 10$ protocols). Avg(Corr) and Avg(Comp) are the means of M_1–M_3 and M_4–M_5, respectively. Bold entries indicate the best value per column.

System	LLM	M_1	M_2	M_3	Avg(Corr)	M_4	M_5	Avg(Comp)
Codex	GPT-5.5	0.615	0.586	0.534	0.578	0.757	0.961	0.859
Claude Code	Opus 4.7	0.724	0.643	**0.594**	0.654	**0.758**	0.753	0.756
OpenCode	Qwen 3.5-plus	0.762	0.430	0.526	0.573	0.668	0.866	0.767
MASTA (Ours)	Qwen 3.5-plus	**0.829**	**0.699**	0.567	**0.698**	0.745	**1.000**	**0.873**

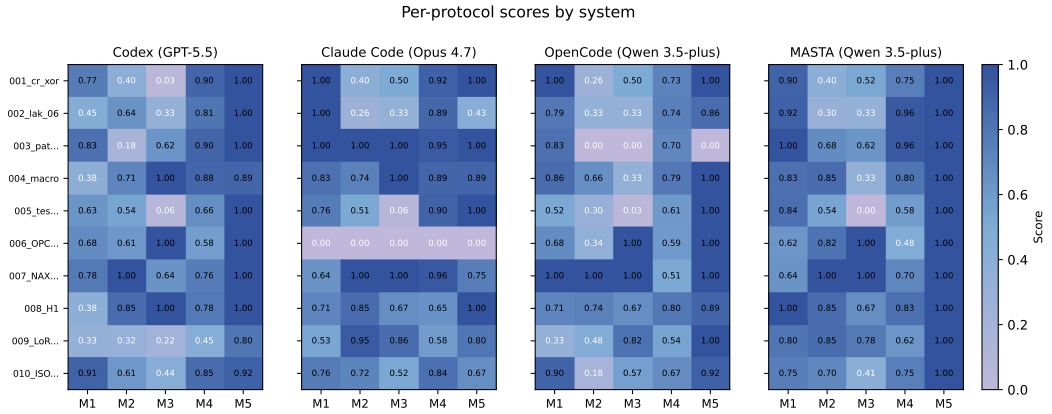


Figure 2: Per-protocol M_1–M_5 heatmap for the four systems of Table 2. Rows are the 10 benchmark protocols (sorted by id), columns are the five metrics, and cells give per-protocol scores.

MASTA ranks first on both sub-averages, with an Avg(Corr) of 0.698 and an Avg(Comp) of 0.873, outperforming every baseline, including those backed by substantially stronger LLMs. On M_5 (vulnerability-mutation detection), it attains a perfect score of 1.000, meaning every injected security flaw is correctly exposed as a `falsified` lemma by Tamarin, whereas the best baseline (Codex on GPT-5.5) reaches only 0.961. The most controlled comparison is OpenCode versus MASTA: both use Qwen 3.5-plus as the underlying LLM, so any performance gap is attributable to the agent architecture rather than model capacity. MASTA improves over OpenCode by $+0.067$ on M_1, $+0.269$ on M_2, $+0.041$ on M_3, $+0.077$ on M_4, and $+0.134$ on M_5, which raises correctness by 0.125 and completeness by 0.106, demonstrating that the three-agent role decomposition and Arbiter-driven scheduling contribute across all evaluation dimensions. Notably, despite running on Qwen 3.5-plus, MASTA surpasses Claude Code (Opus 4.7) on M_1, M_2, and M_5, and surpasses Codex (GPT-5.5) on M_1, M_2, M_3, and both sub-averages, showing that architectural design can compensate for a weaker underlying model.

Figure 2 breaks these means down by protocol, making the architectural advantage of MASTA more concrete. On M_5, MASTA achieves a perfect score on every single protocol, confirming that it detects all injected vulnerabilities across the entire benchmark without exception. On M_3, each baseline scores below 0.1 on two of the 10 protocols, whereas MASTA does so only on protocol 005, where it scores 0.00; since all four systems score below 0.1 on that protocol, this entry reflects the difficulty of the protocol rather than a systematic failure of the architecture.

6.3 Ablation

To investigate the contribution of the two core architectural choices in MASTA, namely the Architect–Auditor operational role split and the dedicated Arbiter, we design two ablation variants. *w/o Role Specialization* merges the Architect and Auditor into a single operational agent that handles all four jobs, while keeping the Arbiter and its cross-agent scheduling intact. *w/o Arbiter* retains the Architect–Auditor split but removes the Arbiter entirely, so each agent self-evaluates after every job and revises its own output without any cross-agent dispatch. Both variants use the same Qwen 3.5-plus backbone

Table 3: Ablation results on $n = 5$ protocols. Differences should be read as descriptive rather than confirmatory given the small scope.

Variant	M_1	M_2	M_3	Avg(Corr)	M_4	M_5	Avg(Comp)
MASTA (Ours)	**0.898**	**0.553**	**0.363**	**0.605**	**0.811**	**1.000**	**0.905**
w/o Role Specialization	0.580	0.339	0.167	0.362	0.511	0.571	0.541
w/o Arbiter	0.358	0.000	0.000	0.119	0.321	0.000	0.161

Table 4: Cost of each system (Tamarin calls, wall-clock time, and API tokens from session logs).

System	Tamarin calls	Wall-clock time	API tokens
MASTA	1,567	52.3 h	111.1 M
Codex	863	8.7 h	81.5 M
Claude Code	966	55.4 h	184.0 M
OpenCode	3,728	39.6 h	424.5 M

and one-hour per-protocol budget as MASTA; the full configuration of each variant is reproduced in Appendix F.

Table 3 reports the results on five protocols. Removing the Arbiter collapses the system: correctness drops by 0.486 and completeness by 0.744, with M_2, M_3, and M_5 reduced to zero. This shows that the Arbiter's cross-agent scheduling and feedback-driven fix-job dispatch are essential for the system to produce a verifiable formal model at all. Without the Arbiter, each agent's self-evaluation fails to detect and recover from accumulated modeling errors. Removing the Architect–Auditor split reduces correctness by 0.243 and completeness by 0.364. The gap on M_4 and M_5 indicates that separating model-side and property-side expertise helps the system handle variant protocols more robustly.

6.4 Cost and Robustness to Verifier Feedback

Verification calls and wall-clock time. Across the 149 tasks (10 protocols and 139 mutations), MASTA averages 10.5 Tamarin calls and 21 minutes of wall-clock time per task, for a total of 52.3 hours. The feedback loop terminates on the Arbiter's own decision; the external constraints are a 3,600 s wall-clock budget per task and a 300 s budget per Tamarin call. No task was terminated due to budget exhaustion: the median wall-clock consumption was 1,058 s (29% of the budget), the longest task took 3,374 s (93.7%) and still completed on its own, and only 5 of the 149 tasks consumed 80% or more of the budget.

Cost against baselines. Table 4 compares the cost of the four systems. OpenCode, which shares MASTA's backbone, makes 2.4 times as many Tamarin calls (3,728 versus 1,567) and spends 3.8 times as many tokens, yet scores lower on both sub-averages; more frequent prover invocation is therefore not what drives the improvement. Claude Code spends the most wall-clock time and 1.7 times MASTA's tokens, and it also scores lower on both sub-averages. Codex is the cheapest, using 73% of MASTA's tokens and one sixth of its wall-clock time, but its correctness is 0.120 below MASTA's (0.578 versus 0.698). MASTA is the only system that leads on both sub-averages.

Robustness to imperfect verifier feedback. Of the 1,567 Tamarin calls, 29 (1.85%) timed out, and 14 of the 149 tasks (9.4%) encountered at least one timeout; 93% of the timeouts were resolved by the responsible agent within its own turn, and all 14 tasks ran to completion. Appendix I details how the Arbiter handles persistent timeouts, with one representative trace of each Arbiter decision.

6.5 Second Backbone

To test whether the gains come from the architecture or from synergy with Qwen 3.5-plus, we evaluate MASTA with a second, architecturally distinct backbone, Sonnet 4.6. Because of the time and token cost of a full benchmark run, we use a reduced subset of three protocols, each with its base model plus 3 operational and 2 to 3 vulnerability mutation variants, yielding 20 modeling tasks per backbone. We rerun Qwen 3.5-plus on the identical subset so that the two runs are strictly comparable (Table 6 in Appendix G). The two backbones perform comparably: Sonnet 4.6 is slightly ahead in correctness

(0.747 versus 0.738), while Qwen 3.5-plus is slightly ahead in completeness (0.978 versus 0.920). We interpret this as evidence that the gains come from the architecture rather than from synergy with a particular backbone.

7 Conclusion

We presented MASTA, an end-to-end multi-agent system that generates executable Tamarin formal models directly from natural-language protocol descriptions. MASTA introduces a structural *operational–evaluative* role decomposition that distributes formal-model authoring across an Architect and an Auditor and isolates evaluation in a dedicated Arbiter whose only job is workflow scheduling, together with a dynamic Tamarin-anchored scheduler in which the Arbiter dispatches the next agent based on the latest Tamarin verdict and modeling state and routes problem-type-tagged fixes back to any responsible agent under a deduplicated modification list. Together, these designs meet the two requirements stated in Section 1: process-level supervision and a structurally independent evaluator.

We also constructed a benchmark of 10 security protocols paired with 101 operational and 38 vulnerability mutations, totaling 149 modeling tasks, together with five fine-grained metrics that jointly assess model correctness (M_1–M_3) and completeness (M_4–M_5). On this benchmark, MASTA achieves the best scores on M_1 (process fidelity, 0.829), M_2 (property consistency, 0.699), and M_5 (vulnerability detection, 1.000) among all evaluated systems, and ranks first on both the correctness average (0.698) and the completeness average (0.873) despite using the weaker Qwen 3.5-plus backbone. The matched-backbone OpenCode–MASTA pair isolates the contribution of the multi-agent architecture. Ablations on five protocols confirm that removing the Arbiter collapses M_2, M_3, and M_5 to zero, while removing the Architect–Auditor split reduces correctness by 0.243 and completeness by 0.364, indicating that the observed gains stem from both the role decomposition and the Arbiter-driven scheduling.

Limitations

The limitations of our work are as follows:

Benchmark scale and threat model. The inclusion criterion (Tamarin auto-termination within ten minutes) caps protocol complexity for tractable M_1–M_5 evaluation, and the threat model is limited to Dolev–Yao symbolic semantics. Larger protocols, where the one-hour budget may be insufficient, and extensions to computational threat models such as CryptoVerif [5] are left for future work.

LLM context window. The Arbiter's scheduling decision requires the full formal model, the accumulated modification list, and the latest Tamarin verdict to fit within the LLM context window, so on protocols whose accumulated state exceeds the effective context length, the scheduler may truncate state and miss issues.

Tamarin-corpus coverage in pretraining. Tamarin-specific syntax generation depends on the LLM's coverage of the Tamarin corpus and degrades on constructs underrepresented in pretraining. A related concern is data contamination: some benchmark protocols correspond to widely studied designs whose public Tamarin models may overlap with LLM pretraining corpora, which can inflate baseline scores or compress the gap between systems. We do not quantify this contamination.

Tamarin specialization. MASTA is specialized to Tamarin: prompt templates, role specifications, and the dispatch tag set are all structured around multiset rewriting rules and temporal-logic lemmas, so migrating to ProVerif or CryptoVerif requires dedicated adaptation while preserving the Architect–Auditor–Arbiter decomposition. Complementing synthetic mutations with real-world CVE reproduction would also tighten the evaluation against deployed vulnerability distributions.

Acknowledgments and Disclosure of Funding

This research was supported in part by the National Natural Science Foundation of China under grants No. 62625207, 62602479, and the China Postdoctoral Science Foundation - Hubei Joint Support Program under Grant Number 2026T077HB. The corresponding authors are Jing Chen and Min Shi.

References

[1] MASTA: Code, benchmark, and experimental logs. `https://github.com/luojiazhishu/masta`, 2026.

[2] David Basin, Michel Keller, Saša Radomirović, and Ralf Sasse. Alice and Bob meet equational theories. In *Logic, Rewriting, and Concurrency: Essays Dedicated to José Meseguer on the Occasion of His 65th Birthday*, pages 160–180. Springer International Publishing, 2015.

[3] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirović, Ralf Sasse, and Vincent Stettler. A formal analysis of 5G authentication. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1383–1396, 2018.

[4] David Basin, Ralf Sasse, and Jorge Toro-Pozo. The EMV standard: Break, fix, verify. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1766–1781, 2021.

[5] Bruno Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security and Privacy (S&P)*, pages 140–154, 2006.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[7] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations (ICLR)*, 2024.

[8] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1773–1788, 2017.

[9] Cas J. F. Cremers. The Scyther tool: Verification, falsification, and analysis of security protocols. In *Computer Aided Verification (CAV)*, pages 414–418. Springer, 2008.

[10] Cristian Curaba, Denis D'Ambrosi, Alessandro Minisini, and Natalia Pérez-Campanero Antolín. CryptoFormalEval: Integrating LLMs and formal verification for automated cryptographic protocol vulnerability detection. *arXiv preprint arXiv:2411.13627*, 2024.

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

[12] Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1229–1241, 2023.

[13] Nadim Kobeissi, Georgio Nicolas, and Mukesh Tiwari. Verifpal: Cryptographic protocol analysis for the real world. In *International Conference on Cryptology in India (INDOCRYPT)*, pages 151–202. Springer International Publishing, 2020.

[14] Steve Kremer and Robert Künnemann. Automated analysis of security protocols with global state. In *IEEE Symposium on Security and Privacy (S&P)*, pages 163–178, 2014.

[15] Thanh Le-Cong, Bach Le, and Toby Murray. Can LLMs reason about program semantics? A comprehensive evaluation of LLMs on formal specification inference. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vienna, Austria, 2025. Association for Computational Linguistics.

[16] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. StarCoder: May the source be with you! *Transactions on Machine Learning Research*, 2023.

[17] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022.

[18] Gavin Lowe. A hierarchy of authentication specifications. In *IEEE Computer Security Foundations Workshop (CSFW)*, pages 31–43, 1997.

[19] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[20] Ziyu Mao, Jingyi Wang, Jun Sun, Shengchao Qin, and Jiawen Xiong. LLM-aided automatic modeling for security protocol verification. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 642–654. IEEE, 2025.

[21] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Computer Aided Verification (CAV)*, pages 696–701. Springer, 2013.

[22] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[23] The Tamarin Team. tamarin-prover: Source code repository. `https://github.com/tamarin-prover/tamarin-prover`, 2024. Accessed: 2026-05.

[24] Haoze Wu, Clark Barrett, and Nina Narodytska. Lemur: Integrating large language models in automated program verification. In *International Conference on Learning Representations (ICLR)*, 2024.

[25] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

[26] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J. Prenger, and Animashree Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 21573–21612, 2023.

A Action Block Specification

An *action block* $b = (r,\ ops)$ consists of a participant role r and an ordered operation sequence $ops = (o_1, o_2, \ldots)$. Each operation o_i is drawn from the five types listed in Table 5. Operations within a block must appear in the fixed total order

$$\textsc{Receive} \prec \textsc{GenerateNonces} \prec \textsc{Compute} \prec \textsc{Assertion} \prec \textsc{Send},$$

and each type may appear zero or more times in that relative order. If the block corresponds to the final step of a role's participation, the SEND entry carries the special marker $\langle \textsc{End} \rangle$ to indicate that no message is transmitted; this marker is used for termination only and does not represent a real network message.

Table 5: The five operation types inside an action block.

Type	Semantics
RECEIVE	Receive a message from the network; bind its subterms to local variables.
GENERATENONCES	Sample fresh random values (nonces, session keys).
COMPUTE	Apply cryptographic functions (encryption, hash, signature, MAC, DH exponentiation) to produce new terms.
ASSERTION	Check a condition on local state (e.g., signature verification, equality test); abort the block if the check fails.
SEND	Transmit a term over the network (or $\langle \textsc{End} \rangle$ if no message is sent).

Block equality. Two action blocks $b = (r, ops)$ and $b' = (r', ops')$ are *equal* iff $r = r'$ and ops and ops' are item-wise alpha-equivalent. Alpha-equivalence treats all function symbols (encryption schemes, hash functions, etc.) as constants and permits only renaming of bound variables; in particular, changing a function symbol or reordering operations within the same type counts as a substitution under the M_1 edit distance.

Action-block sequence. An action-block sequence $A = (b_1, b_2, \ldots, b_n)$ orders the n blocks by their first occurrence in an honest protocol execution trace. Each block b_i corresponds to exactly one Tamarin multiset rewriting rule in the reference model.

B Metric Computation Details

This appendix completes the formal specification of the metrics introduced in Section 5.

M_1 **action-block equality.** Action-block equality requires identical role and item-wise alpha-equivalent internal operation sequences of the form receive→nonce→compute→send. Function symbols are treated as constants, and only bound variables may be renamed. The underlying Levenshtein edit distance charges insertions, deletions, and substitutions one unit each.

M_2 **canonical lemma template.** Direct logical-equivalence checking on Tamarin lemma formulas is undecidable in general, so we constrain the agent's output to a canonical template and reduce the equivalence check to AST alpha-equivalence. The template covers the five property categories in our benchmark—secrecy, aliveness, weak agreement, non-injective agreement, and injective agreement—and fixes for each category: (i) the quantifier structure (e.g., a fixed prefix `All A B t #i. ... ==> ...`); (ii) the action-fact vocabulary, drawn from a closed set of facts including `Create`, `Running`, `Commit`, `Secret`, `Honest`, and `LtkReveal` with predefined semantics; (iii) the variable naming convention; and (iv) the clause order in the lemma body. A reference lemma and a generated lemma are then aligned by their position in the lemma list (every protocol's reference fixes the order), parsed into ASTs, and compared under alpha-equivalence (function symbols are treated as constants, and only bound variables may be renamed). A match scores 1.0, otherwise 0.0; generated lemmas violating the template are scored 0.0 regardless of their content, and excess generated lemmas beyond the reference list do not contribute to the numerator or the denominator.

M_3 **proof-step ratio for `proved` verdicts.** A purely binary verdict-agreement score gives undue credit to agents whose formal models are vacuously true, i.e., models under which the protocol is never actually executed and Tamarin trivially returns `proved` on every safety lemma. We therefore use the Tamarin proof-step count $n(l)$ as a proxy for proof depth and replace the binary credit on matching `proved` verdicts with the ratio $\min(1,\, \hat{n}(l)/n^*(l))$, which is strictly less than 1 whenever the agent proof terminates in fewer steps than the reference proof. Verdict mismatches and matching non-`proved` verdicts are scored as in the main text: 0 on mismatch, 1 on agreement. The proof-step count is read from Tamarin's standard textual output, and runs that produce no proof object (e.g., parse errors before invocation) are treated as a verdict of `error` and contribute 0 on mismatch with any reference verdict.

Timeout scoring. Under the one-hour per-protocol timeout from Section 4, M_1 and M_2 apply the same formulas as non-timeout cases on whatever partial artifacts were produced before the cutoff, with missing action blocks charged to edit distance and missing lemmas left unmatched. M_3 assigns the agent verdict `timeout` on every lemma not yet verified; under the scoring rule above, this collapses to 0 when the reference verdict is `proved` or `falsified`, and to 1 only in the (rare) case that the reference verdict is also `timeout`. A timed-out operational-mutation variant counts in the M_4 denominator and contributes zero detected mutation points; the same convention applies to M_5, since a timeout cannot yield a `falsified` verdict from Tamarin. Protocol-level aggregation is otherwise unchanged.

No false-positive penalty for M_4 / M_5. M_4 false positives are modeling errors already penalized by M_1, M_5 false positives are constrained by Tamarin's objective verification verdicts, and any residual false positive arising from upstream modeling defects is absorbed by $M_1/M_2/M_3$, so M_4

Algorithm 1 Arbiter workflow scheduling loop.

Require: protocol document D, time budget $T = 1$ hour
Ensure: final formal model or timeout

initialize state σ from D; modification list $L \leftarrow []$
while elapsed time $< T$ **and** σ is not *Final* **do**
  $issue \leftarrow \text{DETECTISSUE}(\sigma)$ {using latest Tamarin verdicts}
  if $issue$ is NONE **then**
    $job \leftarrow \text{NEXTNORMALJOB}(\sigma)$ {J1→J2→J3→J4}
    $target \leftarrow \text{OWNER}(job)$
  else
    $job \leftarrow \text{FIXJOB}(issue)$
    $target \leftarrow \text{RESPONSIBLEAGENT}(issue)$
  end if
  $dir \leftarrow (\text{PROBLEMTYPE}(job), \text{TARGETARTIFACT}(job))$
  while $\text{LLMDEDUP}(dir, L)$ is **true do**
    $dir \leftarrow \text{ADJUSTDIRECTION}(dir)$ {deterministic: cycle problem-type label, then change target artifact}
  end while
  append $(target, dir)$ to L
  $output \leftarrow target.\text{EXECUTE}(job, \sigma)$
  if $job \in$ {J3, J4} **then**
    $verdict \leftarrow target.\text{RUNTAMARIN}(output)$
  else
    $verdict \leftarrow \emptyset$
  end if
  $\sigma.\text{UPDATE}(output, verdict)$
end while
return $\sigma.\text{EMIT}()$

and M_5 are reported as recall over their respective mutation denominators without an additional false-positive term.

C Scheduling Algorithm Details

This appendix reproduces the full pseudocode and the auxiliary deduplication machinery summarized in Section 4.2.

Modification-list direction format. $direction_text$ is restricted to a "problem-type label + target-artifact identifier" granularity and may not contain a concrete modification operation: for instance, "the security requirement disagrees with the protocol description" is allowed, whereas "rename K to `ltk`(A, B) in the secrecy lemma" is not.

Six-class problem-type taxonomy. The problem-type label is drawn from a fixed six-way taxonomy: rule unreachable, property disagrees with protocol, security-requirement omission, lemma syntax error, executable-verification failure, and other. Open subtypes are carried by the target-artifact identifier, so the LLM deduplication check runs over a stable label set.

ADJUSTDIRECTION procedure and termination. Before issuing a new dispatch, the Arbiter calls $\text{LLMDEDUP}(dir, L)$, which returns true iff the candidate direction is judged equivalent to some entry already in L. On a true verdict, the Arbiter applies ADJUSTDIRECTION, a deterministic procedure that first cycles through the remaining five problem-type labels in a fixed order, and only after all six labels have been tried at the current target artifact does it switch the target artifact within the job's admissible artifact set. The two-level cycle terminates because the direction space is finite, and the resulting dispatch is guaranteed not to collide with any historical entry in L, which prevents routing deadlock.

D Prompt and Agent Configurations

This appendix details the six-layer prompt template summarized in Section 4.3. Each agent's full prompt is assembled by concatenating the six layers below in the listed order, each rendered under a markdown `##` section header. Empty layers are filled with the literal placeholder `(empty)` so that the LLM never sees a missing section.

Layer 1: Output Protocol. A ReAct-style behavioral contract that constrains the output of every LLM call to a single JSON object of the form

```
{
  "action":  { "capability": "<name>", "args": { ... } },
  "summary": { "did": "...", "last_verification": "...",
               "outstanding": "...", "concern": "..." }
}
```

Exactly one `action` is permitted per turn (no arrays). The four `summary` fields are mandatory: `did` is a one-sentence description of the action and its effect, `last_verification` carries the Tamarin verdict and key error information when the action invoked the verifier (otherwise empty), `outstanding` reports what the current phase still needs (e.g., "the secrecy lemma for the session key is still missing"), and `concern` flags any self-detected issue (format ambiguity, repeated failures, missing premises). The host parses this summary on every turn to decide whether to continue, dispatch a fix-job, or advance to the next phase.

Layer 2: Role Description. A natural-language statement of the agent's identity and mission, fixed per agent type. The Architect is positioned as a Tamarin modeling specialist responsible for action blocks and rules; the Auditor as a cryptographic property specialist responsible for the security requirements and lemmas; the Arbiter as a workflow coordinator responsible for diagnosing issues and dispatching fixes.

Layer 3: Capability Description. The capability list is rendered automatically from each agent's capability dictionary, listing every callable capability together with its signature and calling convention. The Architect and Auditor share `files.{read,write,exists}` and the Tamarin capabilities `tamarin.run_executable` and `tamarin.verify_lemma`, plus an `end` capability that signals task termination. The Arbiter additionally holds the scheduler capabilities `scheduler.schedule_initial`, `scheduler.schedule_revision`, and `scheduler.advance`. Generating this layer from the live capability dictionary keeps the prompt synchronized with the dispatcher and avoids prompt–code drift.

Layer 4: Responsibilities Description. A detailed description of the agent's per-phase workflow, expected output artifacts, and failure-handling strategy. For example, the Architect's J1 phase reads `document.txt`, writes `action_blocks_draft.yaml`, and terminates; its J3 phase advances rule by rule, verifying executable reachability after each rule and deciding whether to continue or to revise the most recent rule. This layer also fixes the canonical output formats (YAML for action blocks, Tamarin for rules and lemmas, markdown for the security requirements) so that the agent's free-form decisions still produce parseable artifacts.

Layer 5: Context. A serialized history of all prior turns in the current run, formatted as a sequence of `(action, result)` pairs separated by horizontal rules. Each entry records the agent's chosen capability call together with the value returned by the host (file contents, Tamarin verdicts, scheduler responses). This layer functions as the agent's working memory and ensures that multi-turn reasoning has access to the full execution trace rather than only the most recent step. A compaction routine bounds context length: the most recent eight entries are kept verbatim, while earlier entries are replaced by an LLM-generated summary that preserves verdicts, decisions, and outstanding concerns.

Layer 6: Current Task. A short, phase-specific instruction that the host (typically the Arbiter) sets at every dispatch. Examples include "generate the action blocks from the protocol document and write `action_blocks_draft.yaml`", "write a Tamarin lemma for each entry in the security requirements and verify executability", and "inspect the current rule set and choose a target for

revision". This layer lets the same agent be repurposed across phases without rewriting the upper layers.

Step lifecycle. Each invocation of `agent.step()` runs the following pipeline: (1) assemble the six-layer prompt; (2) call the LLM and capture both the thinking-channel output and the JSON answer; (3) parse the response, validate the `action` structure and the `summary` fields, and tolerate stray markdown fences or extra text; (4) dispatch the action to the corresponding capability handler and append the resulting `(action, result)` pair to the context; (5) trigger context compaction if the next prompt would exceed the token budget; (6) hand the four summary fields to the host, which decides whether to continue, fix, or advance the phase.

Execution environment. All experiments use Tamarin prover 1.10.0 with Maude 3.4 and run on a workstation with an AMD Ryzen 9 7950X3D CPU and 192 GB of memory, executing 16 protocol instances in parallel. All LLMs are queried with their default API decoding settings.

E Reproducibility Materials

All materials listed below are released via a public repository [1]. They were frozen before the first experimental run:

1. Dataset: the 10 protocol source documents, the standard expert-authored Tamarin reference model of each protocol and the derived reference models for each mutation variant (produced with LLM assistance and verified by a human expert), together with the operational-mutation list, the vulnerability-mutation list, and the position annotations for every protocol.
2. Benchmark annotations: the reference action-block sequence in JSON and the canonical-format reference lemmas for every protocol and every mutation model.
3. Interfaces and prompts: the skill signatures, I/O formats, and system prompts of the three agent roles.
4. Run scripts: the deployment script, the ablation-variant switch configuration, and the evaluation scripts (M_1–M_5 computation and protocol-level aggregation).

F Ablation Configuration Details

This appendix reproduces the configuration of the two ablation variants summarized in Section 6.3.

w/o Role Specialization. A single operational expert handles every job previously assigned to the Architect or the Auditor. The system prompt of this single expert merges the Architect's and the Auditor's role specifications, and per-job prompt instructions are routed by job type, so that the model-side jobs (J1 action serialization and J3 rule modeling) reuse the main system's Architect instruction body, while the property-side jobs (J2 security-requirement extraction and J4 property modeling) reuse the main system's Auditor instruction body. The Arbiter role, the workflow scheduling, and the cross-agent dispatch are all preserved, except that the dispatch target collapses to the single operational expert; the one-hour per-protocol timeout and the modification-list deduplication check are unchanged from the main system. Any drop relative to MASTA attributable to this variant isolates the contribution of splitting operational work between two specialized experts.

w/o Arbiter. The Architect and the Auditor operate without an Arbiter and follow the default job sequence J1→J2→J3→J4 with the same job-to-role assignment as the main system. Each agent runs Tamarin within its own jobs (executable verification during J3 for the Architect, batched lemma verification at the end of J4 for the Auditor). After every job, the current agent self-evaluates using the Arbiter's review prompt rewritten with a "self-evaluation" role label, with the Tamarin verdict (when applicable) and an LLM confirmation of outstanding issues serving as decision rules. If self-evaluation passes, control is passed to the next job's designated agent; if it fails, the current agent revises its own output without dispatching to any other agent. The self-evaluation and revision loop shares the main system's one-hour per-protocol timeout. Upon J4 completion with the Auditor's self-evaluation passing, the variant emits its final formal model, which, like the main-system output

defined in Section 4.1, consists of Tamarin rules and first-order temporal-logic lemmas; the sole difference is that the dispatch authority is replaced by the agents' own self-evaluation. Because the job sequence and the job-to-role assignment are identical to Section 4.1, any performance gap is jointly attributable to the dedicated-evaluator side of the role decomposition and to the dynamic Tamarin-anchored scheduling.

G Second-Backbone Results

Table 6 reports the per-metric scores of the second-backbone study in Section 6.5.

Table 6: MASTA with two backbones on the same 20-task subset.

Backbone	M_1	M_2	M_3	Avg(Corr)	M_4	M_5	Avg(Comp)
Sonnet 4.6	0.816	0.703	0.723	0.747	0.839	1.000	0.920
Qwen 3.5-plus	0.967	0.643	0.604	0.738	0.956	1.000	0.978

H Comparison with SAPIC and AutoSM

Table 7 gives the qualitative comparison discussed in Section 2.

Table 7: Qualitative comparison of MASTA with SAPIC and AutoSM.

	SAPIC [14]	AutoSM [20]	MASTA
Input	applied-pi process written by the user	natural-language document	natural-language document
Modeling	done by the user; SAPIC only translates it	generated, with user interaction in the Repairer stage	generated, no human involvement
Repair signal	not applicable	static analysis and user interaction	Tamarin verdicts
Security properties	supplied by the user	taken from the reference model	generated, then verified
Evaluation	not applicable	18 protocols; a model is correct when the properties verified on the reference still hold on it	149 tasks; 5 metrics covering process structure, properties, verdicts, and mutation completeness

I Robustness to Imperfect Verifier Feedback

This appendix details the timeout analysis summarized in Section 6.4. A single Tamarin call has a 300 s budget, and each task has a 3,600 s budget. If a call exceeds its budget, it is retried within the modeling loop; only exhausting the task budget leaves the task unfinished. Table 8 reports both measures across all 1,567 calls in our benchmark runs.

Table 8: Tamarin timeouts across all benchmark runs of MASTA.

Measure	Value
Total Tamarin calls	1,567 (`verify_executable` 744 / `verify_lemma` 795 / `well_formed` 28)
Timeouts	29 (1.85%)
Tasks with at least one timeout	14 / 149 (9.4%)
Tasks terminated by the 3,600 s budget	0 / 149

An isolated timeout is resolved by the agent itself: it revises `model.spthy` and reruns the check within the same turn. This covers most cases. Persistent timeouts or failures trigger the Arbiter, which reviews the history and determines whether the current agent is improving in the right direction. If so,

that agent gets additional attempts. If the fault lies with another role, the Arbiter reassigns the repair to that role and describes what is wrong. All 14 tasks that encountered timeouts ran to completion.

Arbiter recovery actions. The Arbiter recovers through two distinct actions and selects between them based on the fault location: `scheduler.advance` grants the current specialist another turn while preserving its context, leaving the assignment unchanged; `scheduler.schedule_revision` cancels the agent while it is still running and re-dispatches the work to the role that owns the damaged artifact, with a typed problem label and a brief description of the fault. We provide one trace of each case below, taken directly from our logs. In both workflows, every lemma receives a definite verdict.

Trace A: the responsible agent is correct, so it is granted more attempts. Task `007_NAXOS_04_simple_excl3/run_1/action_variants/01`. The failure occurs in the Architect's rule-modeling stage, the earliest point in the pipeline, and manifests as two consecutive 300 s timeouts.

```
architect  verify_executable -> falsified, timeout (300.12s)
architect  well_formed       -> falsified, timeout (300.25s)
arbiter    concern: "both timed out after 300s, indicating
           structural issues in the theory"
arbiter    -> scheduler.advance      (same Architect continues, context kept)
arbiter    -> scheduler.advance
architect  rewrites model.spthy 3 times (consulting the rule specification)
architect  well_formed       -> verified (253.83s)
architect  verify_executable -> verified (128.76s)
architect  END: "all rules verified"
arbiter    -> schedule_initial(model_properties -> Auditor)
           all 5 security lemmas then obtain clean verdicts
```

After these two consecutive timeouts, the Arbiter did not terminate the task, nor did it discard the model and reassign the work to another role. Judging from the verdict history, the Architect was moving in the right direction; therefore, the Arbiter issued `scheduler.advance` twice and allowed it to continue. The timing supports this decision: `well_formed` decreased from a 300 s timeout to 253.83 s, and `verify_executable` decreased from a 300 s timeout to 128.76 s. Each rewrite was a genuine improvement, not a blind retry. The error also remained contained: the subsequent Auditor stage modeled 5 security lemmas, each of which received a definite verdict.

Trace B: the responsible agent is wrong, so the work is taken away from it. Task `010_ISO9798_4_4_979844_reuse_once_1_1/run_1/base`. Here, the Architect had already completed its work correctly, and the error was introduced later by the Auditor.

```
architect  verify_executable -> verified (0.48s)
architect  END: "all rules verified"
arbiter    -> schedule_initial(model_properties -> Auditor)

auditor    rewrites model.spthy while adding lemmas, but REPLACES the
           Test1()-Test4() action facts instead of appending to them,
           silently breaking the executable lemma that had just passed
auditor    verify_lemma secrecy_key -> falsified (0.44s)
auditor    concern: "violated contract by replacing Test1()-Test4() action
           facts instead of adding to them; executable lemma will fail"

arbiter    -> scheduler.schedule_revision
                target_role     = architect
                target_artifact = model.spthy
                problem_type    = executable-verification failure
           [auditor] cancelled after 5 turns, reason='superseded by new schedule'
architect  start (task_label: revise)
architect  "The Arbiter has flagged an executable-verification failure.
            The brief states that the Auditor replaced Test1()-Test4()..."
architect  16 x verify_executable (0.44-0.60s, no timeouts)
architect  END: "model.spthy now contains Test1()-Test4() alongside
            Running/Commit/Honest/Secret; verify_executable returns verified"

arbiter    -> schedule_initial(model_properties -> Auditor)
auditor    10 x verify_lemma, all clean verdicts (0.51-1.84s)
auditor    END: "all 5 lemmas modeled and verified"
arbiter    END: "workflow complete"
```

The failure appeared in the Auditor's stage, but the defect lay in the rules, which are the responsibility of the Architect. Following the six-class problem taxonomy (Appendix C), the Arbiter mapped this executable-verification failure to the Architect and `model.spthy`. It intervened on the first falsified verdict and cancelled the Auditor immediately: because the Auditor may not edit rules, further lemma-level revisions could not have fixed the defect. The Architect's first turn restates the brief, naming both the problem type and the specific contract violation, which confirms that `schedule_revision` transmits actionable guidance rather than merely performing a hand-off.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction state MASTA's three innovations (operational–evaluative role decomposition, Tamarin-anchored scheduling, and a benchmark with five fine-grained metrics) and the experimental claims, all of which are supported by the results in Section 6.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The Limitations section in the conclusion discusses four limitations: benchmark scale and threat model, LLM context window, Tamarin-corpus coverage in pretraining (including data contamination), and Tamarin specialization.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: The paper does not contain theoretical results. The metric definitions (M_1–M_5) in Section 5.1 and the scheduling algorithm in Appendix C are operational definitions rather than theorems.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: The method (Section 4), the benchmark construction (Section 5.2), and the experimental setup (Section 6.1) are described in detail. The full prompts, scheduling pseudocode, dataset, and run scripts are released in the public repository [1].

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: All source code, the 10-protocol benchmark with 139 mutation variants, run scripts, and per-run logs are publicly released in the repository [1].

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 6.1 describes the four compared systems, the shared skill set, and the per-protocol time budget. Tamarin version, hardware, and decoding hyperparameters are listed in Appendix D.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: The per-protocol distribution behind each cell of Table 2 is visualized in the heatmap of Figure 2, and the raw per-task scores are released in the public repository [1], so any per-protocol standard deviation or bootstrap confidence interval can be recomputed.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Experiments were conducted on a workstation with an AMD Ryzen 9 7950X3D CPU and 192 GB of memory, running 16 protocol instances in parallel. The total wall-clock time across all systems was approximately two days, and the LLM API cost exceeded 800 USD.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research conforms with the NeurIPS Code of Ethics. No human-subject data is involved, and the released artifacts (a benchmark of standard published protocols and code for protocol formalization) carry no identified misuse risk beyond the dual-use considerations noted in the broader-impacts question.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.

- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: MASTA lowers the expertise barrier for symbolic protocol verification, enabling broader adoption of formal methods and earlier discovery of design-level flaws in deployed protocols. The same automation may in principle be used to discover unknown vulnerabilities in third-party protocol designs; we view this dual-use risk as comparable to existing manual symbolic-analysis tools, since both produce the same class of attack traces.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The released artifacts are a benchmark of standard published protocols and a multi-agent framework around the open-source Tamarin prover, and do not constitute a high-misuse-risk asset such as a generative model or a scraped dataset.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: The Tamarin prover (GPL-3.0) is used in accordance with its license, and its original publication is cited. Qwen 3.5-plus, Sonnet 4.6, Codex, Claude Code, and OpenCode are accessed through their public APIs and CLIs and used under their respective vendor terms of service. Benchmark protocols are derived from formal models in the official Tamarin prover repository [23] (GPL-3.0) and from public GitHub search.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: We release a benchmark of 10 protocols with 139 mutation variants, the reference Tamarin formal models and action-block sequences for every variant, the MASTA source code, and the evaluation scripts. Per-artifact documentation is included in the public repository [1].

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.

- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: LLMs are an essential component of the core agent system: every MASTA agent (Architect, Auditor, Arbiter) is driven by Qwen 3.5-plus, and the three baselines are themselves LLM-based coding agents driven by GPT-5.5, Opus 4.7, and Qwen 3.5-plus, respectively. The second-backbone study (Section 6.5) additionally runs MASTA on Sonnet 4.6. All LLM identifiers and roles are declared in Section 6.1.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.