LLM-aided Automatic Modeling for Security Protocol Verification

Ziyu Mao[1], Jingyi Wang[1*], Jun Sun[2], Shengchao Qin[3], Jiawen Xiong[4]
[1] Zhejiang University, Hangzhou, China
[2] Singapore Management University, Singapore
[3] Xidian University, China
[4] East China Normal University, China

***Abstract*—Symbolic protocol analysis serves as a pivotal technique for protocol design, security analysis, and the safeguarding of information assets. Several modern tools such as TAMARIN and PROVERIF have been proven successful in modeling and verifying real-world protocols, including complex protocols like TLS 1.3 and 5G AKA. However, developing formal models for protocol verification is a non-trivial task, which hinders the wide adoption of these powerful tools in practical protocol analysis.**

In this work, we aim to bridge the gap by developing an automatic method for generating symbolic protocol models using Large Language Models (LLMs) from protocol descriptions in natural language document. Although LLMs are powerful in various code generation tasks, it is shown to be ineffective in generating symbolic models (according to our empirical study). Therefore, rather than applying LLMs naively, we carefully decompose the symbolic protocol modeling task into several stages so that a series of formal models are incrementally developed towards generating the final *correct* symbolic model. Specifically, we apply LLMs for semantic parsing, enable lightweight manual interaction for disambiguation, and develop algorithms to transform the intermediate models for final symbolic model generation. To ensure the correctness of the generated symbolic model, each stage is designed based on a formal execution model and the model transformations are proven sound. To the best of our knowledge, this is the first work aiming to generate symbolic models for protocol verification from natural language documents. We also introduce a benchmark for symbolic protocol model generation, with 18 real-world security protocol's text description and their corresponding symbolic models. We then demonstrate the potential of our tool, which successfully generated correct models of moderate scale in 10 out of 18 cases. Our tool is released at [1].

***Index Terms*—Automatic modeling, Symbolic analysis, LLMs**

I. INTRODUCTION

It is notoriously hard to design and implement security protocols. Among the variety of methods for analyzing security protocols, symbolic methods play an important role in security protocol verification. Many protocol verifiers, such as TAMARIN [2] and PROVERIF [3], have been developed to formally analyze the correctness and security of a protocol, based on a symbolic model of the protocol. The effectiveness and usefulness of these tools are evidenced by the verification of large-scale real-world protocols such as TLS 1.3 [4], 5G AKA [5], and EMV payment [6], which have uncovered critical and subtle security vulnerabilities.

* Corresponding author.

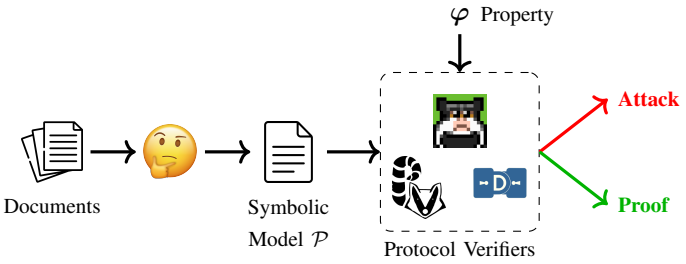


Fig. 1: Overall workflow for symbolic protocol analysis.

However, applying these tools in practice can be costly, as a user must first develop a formal symbolic model of the protocol, which requires not only expert knowledge about the protocol design, but also deep insights (as well as modeling skills) of the corresponding verifier. Take TAMARIN as an example. The overall workflow of protocol analysis is shown in Figure 1. To use TAMARIN, one must first develop a model of the security protocol in its input language (multiset rewriting rules). To do that, a user must first build, in the mind, an abstract model from the protocol document (e.g., IETF RFCs), and then encode the model using multiset rewriting rules which characterize a labeled transition system. Furthermore, the security properties φ to be verified must be encoded in first-order logic (FOL). The validity of verification results depends on whether the multiset rewriting rules and properties are modeled correctly, as well as, subtle choices made during the modeling, e.g., an untyped message in a rule may cause TAMARIN to treat their outgoing messages as input, leading to the non-termination of backwards reasoning [7]. To be fair, TAMARIN and other similar tools certainly made significant effort to make their modeling language accessible, and yet the difficulty in developing the symbolic model limits the application of these tools to ordinary users.

In this work, we aim to address this practical challenge by developing a method to generate symbolic models required by tools such as TAMARIN automatically. Note that there were previously some attempts with similar goals [8], [9], [10]. What makes this work different is that we integrate the capability of large language models (LLMs) into a multi-step approach and adopt a much more powerful intermediate modeling language. Note that although LLMs have shown remarkable capabilities in multiple 'less-formal' tasks like

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00197

text comprehension and code generation, we observe (with empirical evidence based on our preliminary study) that LLMs cannot be naively adopted to solve the problem directly (i.e., generating symbolic models of security protocols based on their natural language documents). Compared with tasks such as generating natural language text or Python code, generating symbolic protocol models are considerably more challenging as (1) there are limited samples that LLMs can learn from, and (2) unlike natural language texts, such symbolic models must be precise, as they are used for formal analysis.

To address the aforementioned challenges, we design and implement a multi-step method for symbolic protocol model generation with the help of LLMs. First, given the natural language document of a security protocol, an LLM is adopted to work as a parser to generate an initial draft of the model in the form of a dedicated lambda calculus (inspired by [11]), which is intended to precisely capture all of the key ingredients of the protocol. This initial draft may be broken due to several reasons including the inevitable ambiguity of natural language documents (since the designer would often omit certain assumed 'common' knowledge) and hallucination of LLMs. We thus introduce a second step that involves lightweight user interaction combined with automatic program analysis to 'repair' the generated model. Afterward, we transform the repaired model into a SAPIC$^+$ specification [12], which can then be compiled into input models for mainstream protocol verifiers including TAMARIN [2], PROVERIF [3], and DEEPSEC [13] soundly. Notably, to ensure the correctness of the generated model, each stage of our method is designed based on a formal execution model and we further establish trace inclusion relations between the models developed at different stages (following a method developed in [14]), which provides overall guarantee on the correctness of the verification result on the generated model.

We evaluated our method on real-world complex protocols and introduce the first new benchmark for such non-trivial automatic symbolic modeling task, which includes 18 popular authentication and key agreement security protocols of different scales. Applying our method, the experimental results show that it can generate 10/18 correct symbolic models from the given documents automatically. Based on the symbolic models, we are able to automatically identify attacks or verify their correctness using TAMARIN. To make our method more useful for real-world developers, we also implement a tool with a self-contained and user-friendly frontend to allow user interaction during the modeling process.

We summarize the main contributions as follows.

- We develop a novel LLM-aided pipeline for generating correct symbolic protocol models from natural language documents, which carefully decomposed the symbolic modeling task into several stages so that a series of formal models are incrementally developed towards generating the final symbolic model.
- We introduce the first benchmark for automatic modeling tasks containing 18 real-world protocols of different scales to benchmark future research in the field.
- We implement a user-friendly tool with a self-contained frontend to allow user interaction. Our evaluation on the benchmark shows promising results, i.e., generating 10/18 correct symbolic models from the given documents automatically.

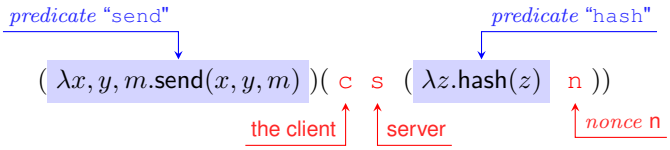
Open-source. Our benchmark, tool implementation and all the experiment data are available at [1].

II. BACKGROUND

In this section, we provide background about different symbolic models and offer an intuition into how lambda calculus can capture the semantics of natural language documents.

A. Lambda-Calculus

A lambda-term in lambda-calculus is in the form of $\lambda x.E$ (also called a function) where x is variable and E is an expression. An expression is defined recursively, which can be a *constant*, a *function*, or an *application* [15]. Here, we do not focus on the theory of lambda calculus. Instead, we introduce how lambda calculus can model the statements in security protocol documents. Taking the sentence "*the client sends a hash value of n to server*" as an example, the corresponding lambda term can be:



where send and hash are the names of the functions that model 'send' action and 'hash' function respectively, and c and s are the identifiers of client and server respectively. Function applications are evaluated by substituting the value of the arguments in the body of the function definition [15], e.g., $(\lambda z.\mathsf{h}(z))\mathtt{n} = [\mathtt{n}/z]\mathsf{h}(z) = \mathsf{h}(\mathtt{n})$. After simplifying the expression inductively, the final result is $\mathsf{send}(\mathtt{c}, \mathtt{s}, \mathsf{hash}(\mathtt{n}))$. In our approach, lambda calculus is used to construct an initial model for the protocol with the help of an LLM, which is the key component of our tool's parsing part.

B. Symbolic Protocol Models

Different protocol verifiers are equipped with different input languages. Though each of them have their own strengths and weaknesses, from the the perspective of usability they have one thing in common: the input language are hard for non-expert user, which prevents the collaboration of those verifiers. SAPIC$^+$ [12] is designed to be a general specification language for security protocol modeling, which allows users to provide a single protocol specification then use three state-of-the-art protocol verifiers: PROVERIF, TAMARIN, DEEPSEC [16] as backends for analysis.

Term algebra. In symbolic protocol analysis, the messages communicated in the protocol and agents are abstracted as terms $\mathcal{T}$, which are built over a set of names $\mathcal{N}$, variables $\mathcal{V}$, and signatures Σ (e.g., $\mathsf{aenc}(\cdot,\cdot), \mathsf{senc}(\cdot,\cdot), \mathsf{hash}(\cdot) \in \Sigma$). A substitution σ is applied to map from variables to terms

```
1  let Client(skc,pks) =
2    new k;
3    out(aenc(k,pks));
4    in(cipher);
5    if h(k) = cipher then
6      event SessKeyC(pks,k);
7      event Commit(pks,h(k))
8  let Server(sks,pkc) =
9    in(r);
10   let k=adec(r,sks) in
11   let pks=pk(sks) in
12   event Answer(pks,k);
13   event Send(pks,h(k));
14   out(h(k))
15 process:
16   !( new sks; out(pk(sks)); !( new skc; out(pk(skc));
17    ( !Client(skc, pk(sks)) | !Server(sks, pk(skc))))
```

Fig. 2: A simple protocol specification using SAPIC$^+$.

which is using postfix notation, i.e., $t\sigma$ denotes replacing every variable t in the domain of σ. If all variables in t can be replaced with ground terms $t_g \in \mathcal{T}_\Sigma(\mathcal{N})$, the substitution σ is *grounding* for t, i.e., $vars(t\sigma) = \emptyset$. Here we use Γ to denote the universe of such substitutions. The semantics of the function symbols is given by the equational theories. For example $\mathsf{dec}(\mathsf{enc}(n,k),k) = n$ illustrates the decryption of a symmetrically encrypted message.

SAPIC$^+$ specification. The symbolic models are specified in a dialect of *appiled-π calculus* in SAPIC$^+$. Figure 2 shows a simple protocol modeled in SAPIC$^+$ for illustration. There are only two message exchange involve two roles in this protocol, which can be denoted in *Alice&Bob notation* [17]:

`Client` $\rightarrow$ `Server: aenc(k, pks)`
`Server` $\rightarrow$ `Client: hash(k).`

The `Client` process (Lines 1-7 in Figure 2) and the `Server` process (Lines 8-14) represent the role behaviors within this key exchange protocol. A client initially generates a new symmetric key `k`, which is then encrypted using the server's public key and transmitted to the server. The server verifies the key's receipt by sending back the hash of the key to the client Event `SessKeyC` and `Answer` claims that the client has set up a session key and the server has received the key respectively. The top specification $\mathcal{P}_t$ which starts with the keyword `process`, initializes the role instantiations. In this work, we use $\mathcal{P}_t \cup (\bigcup_{r\in R} \mathcal{P}_r)$ to denote a complete SAPIC$^+$ specification, where r is the role name and $\mathcal{P}_r$ is the corresponding role process.

Multiset rewriting rules. The SAPIC$^+$ specification is quite similar to the input languages of two of its backends: PROVERIF and DEEPSEC. In contrast, the protocol's symbolic model is specified using *multiset rewriting (MSR) rules* in TAMARIN. In MSR rules, the protocol's states are encoded as *facts*. When a rule is applied, some facts are consumed, and new facts are generated. Naturally, a transition system can be established via MSR rules. The intuition behind the compilation from SAPIC$^+$ to MSR rules is that an execution of a statement in SAPIC$^+$ corresponds to an application of a rule in TAMARIN.

III. MOTIVATION AND OUR APPROACH

A. Motivation

Developing a symbolic model is crucial but challenging. As introduced in Section I, symbolic analysis is a highly impactful approach for protocol security. While most of existing works [4], [5] focusing on reasoning phase (shown in Figure 1), little attention has been given to the modeling phase, where the symbolic model is abstracted from natural language documents. However, as acknowledged in many studies, the modeling phase is effort-intensive. Table I gives an brief statistics of the efforts used in some of famous symbolic protocol analysis work. It is urgent to develop an automatic approach for symbolic model generation.

TABLE I: Estimated results for 3 impactful symbolic analysis works. The third column shows the symbolic model size.

Protocol	Document	Symbolic Models
TLS 1.3	Draft 21 [19], 143 pages	1 model, 2000+ LoC
5G AKA	3GPP TS [20], 722 pages	7 models, 4000+ LoC
EMV	[21], [22], 2000+ pages	40 models, 2000+ LoC

Intuitive code generation is infeasible and untrustworthy. LLMs have demonstrated their impressive capabilities in code generation. It may seem intuitive to treat the modeling task as a form of code generation. However, unlike common programming languages such as Python, *LLMs are unfamiliar with formal specification languages* (due to the missing of dedicated datasets[1]). On the other hand, it is vital to ensure the correctness of the symbolic model as the model is intended for formal verification.

B. Overview of our approach

Motivated by the aforementioned challenges and observations, we design a multi-step approach that divides the automatic task into several stages. Our insight is to only use LLMs for extracting necessary ingredients for the final model while relying on a series of formal models to establish layered correctness for the final symbolic model. Specifically, our approach, illustrated in Figure 3, consists of four stages:

(1) LCCG: a LLM-powered CCG parser[2], which takes a protocol document as input, parses it into lambda calculus expressions (that are defined specifically for modeling security protocols).
(2) L-REPAIR: which repairs the broken specifications with static analysis techniques and user interaction to make it well-formed.
(3) Rewriter: which transforms the lambda expressions into a SAPIC$^+$ specification $\mathcal{P}$.
(4) Compiler: which takes the well-formed SAPIC$^+$ process $\mathcal{P}$ as input and compiles it into models $\mathcal{R}$ accepted by verifiers: TAMARIN, DEEPSEC, and PROVERIF directly.

IV. A DEDICATED LAMBDA CALCULUS

In this section, we introduce a dedicated lambda calculus (λ-DSL) to specify security protocols in an intuitive manner, so that it is easier to be generated by LLMs.

[1]Although many relevant papers have released their symbolic models [18], aligning protocol documents to these symbolic models remains challenging.

[2]CCG: combinatory categorial grammar [23], which is a rule-based system coupling syntax and semantics. It can take the natural language sentence as input and output a lambda expression. Here we use the LLM to serve as a CCG parser.

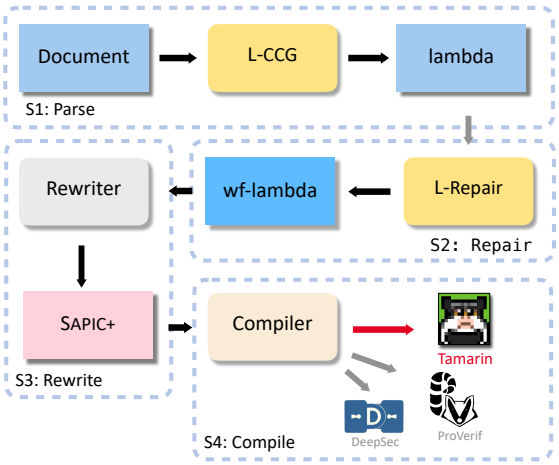


Fig. 3: The high-level workflow of our approach.

A. Notation

We begin by presenting an overview of the notation employed in this work to ease the reader's comprehension.

E	event, corresponds to basic statement in DSL
σ	substitution, maps variables to terms
Σ, Σ^*	set of primitives, $\Sigma^* = \Sigma \cup \{\mathsf{binds}\}$
$\mathsf{aenc}, \mathsf{adec}$	asymmetric encryption and decryption primitives
$\mathcal{N}, \mathcal{R}$	set of nonces and roles respectively
A^{m}	multisets, where elements can appear more than once
$\subseteq^{\mathsf{m}}, \cup^{\mathsf{m}}, /^{\mathsf{m}}$	multiset operations: subset, union, and subtraction
$\mathcal{I}$	multiset of protocol instances
$M \vdash t$	deduction relation, t can be inferred from set M
$c \xrightarrow{l} c'$	state transition from c to c' labeled by l

B. Syntax

From the perspective of lambda calculus, we observed that a sentence in a protocol's document is often an *application*, i.e., instantiating arguments of the key predicate with ground terms (e.g., role name, explicit message). Intuitively, a lambda expression describes that *a role performs some operations at a time-point* from a global perspective. Below we present the syntax of the above λ-DSL, for simplicity we abbreviate application $(\lambda x, \overline{y}.\ e(x, \overline{y})\ a\ \overline{t})$ as $e(a, \overline{t})$, which we call an *event* E. The abstract syntax of our λ-DSL is shown below,

$$
\begin{aligned}
P &::= E; P \mid \mathsf{cond}\ \alpha\ P\ \mathsf{else}\ Q \\
E &::= e(a, \overline{t}) \mid \ \bot,\ a \in \mathcal{A},\ t \in \mathcal{T} \\
e &::= \mathsf{gen} \mid \mathsf{send} \mid \mathsf{recv} \mid \mathsf{know} \mid \mathsf{op}
\end{aligned}
$$

where e is the name of an event E, f and a are the name of functions and agents respectively, t is a term in $\mathcal{T}$, and $\bot$ denotes the terminating statement. We notate lists with overline, i.e., $\overline{t}$ is a sequence of terms. The substitution can be extended from terms to event E with notation $e(a, \overline{t})\sigma$, which is an abbreviation for $e(a\sigma, \overline{t\sigma})$. An event E is *grounding iff* all arguments are grounding.

Following [24], we define a supersort `msg` for terms, i.e., for all term $t \in \mathcal{T}$, t is of sort `msg`. Further, we define two subsorts of `msg` for term t: terms representing agent names are classified as `agent`, denoted $t : \mathtt{agent}$, and terms representing fresh nonces are classified as `nonce`, denoted $t : \mathtt{nonce}$. The grounding events set is denoted as $Ev_{\mathsf{g}} = E(\mathcal{A}, \mathcal{T}_{\Sigma}(\mathcal{N}))$ where $\mathcal{A} = \{a \in \mathcal{N} \mid a : \mathtt{agent}\}$ is the set of agent names.

In this work, we consider the symbol set e presented in the above syntax, which can be extended to other event symbols. The event $\mathsf{gen}(a, n)$ denotes that the agent a generates a fresh nonce n. Interaction events $\mathsf{send}(a, m)$ and $\mathsf{recv}(a, m)$ denote that agent a sends or receives a message m via a public channel. Similar to its backend, TAMARIN, asynchronous communication is the default message exchange mode supported by λ-DSL. Specifically, messages m within a send event are placed on a public channel, where they can be intercepted or manipulated by an adversary and may be received by other roles at any later time. The event $\mathsf{know}(a, \overline{t})$ states that role a knows terms $\overline{t}$ initially, i.e., terms $\overline{t}$ are in the initial knowledge set of a. The event $\mathsf{op}(a, \overline{t})$ is a special kind of event, which states that agent a performs some local operation.

Definition 1 (Event op). *Event* op *is a second-order lambda calculus expression which takes a function f as an argument, i.e., $\lambda a, f, t.\ \mathsf{op}(a, f, t)$.*

Event op can be used to specify some statements in protocol's documents including term binding and security property claiming. We extend the set from Σ to Σ^* with a new function binds, which denotes a term binding for some variable. For explicit binding, the event is in the form of $\mathsf{op}(a, \mathsf{binds}, v, t)$. If the argument f is instantiated with other cryptographic primitives $f \in \Sigma$ like aenc, the event is specifying an implicit binding, e.g., $\mathsf{op}(a, \mathsf{aenc}, m, k)$ stating that agent a encrypts message m with k and then binds the encrypted cipher to a variable implicitly. If $f \notin \Sigma^*$, we say that it is a *signal* [24] event, which models the labeling behavior of an agent, e.g., $\mathsf{op}(a, \mathsf{accept}, m))$ representing that agent a has already accepted a message m and stored it. To make it conform to the syntax, we reform the event $\mathsf{op}(a, f, \overline{t})$ as $\mathsf{op}(a, f(\overline{t}))$.

Using its neat syntax, the λ-DSL enables specification of both stateless and conversational interactions within a protocol. For stateless protocols, send and recv events represent independent message exchanges. Additionally, λ-DSL utilizes SAPIC$^+$ as its backend, automatically storing all accessible variables and constants in state memory. Its pattern-matching functionality enabled by the op statement supports stateful session features such as nonce verification.

Example 1. *Continuing the simple protocol example introduced in Section II-B, we use below expressions to show how our λ-DSL can specify a protocol, and how it can be related to natural language description intuitively.*

```
/*The client generates a key k, encrypts it with the public
 key pkS and sends it to S. The server confirms the receipt
 by sending the hash of the key back to the client.*/
1 @gen(C, k);
2 @op(C,
3  @bind(msg,
4   @aenc(k, @pk(S))));
5 @send(C, msg);
6 @recv(S, msg;
7 @send(S, @hash(k));
8 @recv(C, @hash(k))
```

[GEN]

$$\frac{inst = \{\mathsf{gen}(a,n); P\} \subseteq^{\mathsf{m}} \mathcal{I},\ n \in \mathcal{N} \quad \mathcal{E}' = \mathcal{E} \cup \{n'\},\ \mathcal{I}' = \mathcal{I} \backslash^{\mathsf{m}} inst \cup^{\mathsf{m}} \{P\{n'/n\}\}}{(\mathcal{E},\mathcal{I},\sigma) \to (\mathcal{E}',\mathcal{I}',\sigma)}$$

[ADV]

$$\frac{\nu\tilde{n}.\sigma \vdash M}{(\mathcal{E},\mathcal{I},\sigma) \xrightarrow{\mathsf{K}(M)} (\mathcal{E},\mathcal{I},\sigma)}$$

[SIGNAL]

$$\frac{inst = \{\mathsf{op}(a, f(\bar{t})); P\} \subseteq^{\mathsf{m}} \mathcal{I},\ \mathcal{I}' = \mathcal{I} \backslash^{\mathsf{m}} inst \cup^{\mathsf{m}} \{P\}}{(\mathcal{E},\mathcal{I},\sigma) \xrightarrow{\text{event } f(t)} (\mathcal{E}',\mathcal{I}',\sigma)}$$

[SEND]

$$\frac{inst = \{\mathsf{send}(a,m); P\} \subseteq^{\mathsf{m}} \mathcal{I} \quad \sigma' = \sigma \cup \{m/x\},\ \mathcal{I}' = \mathcal{I} \backslash^{\mathsf{m}} inst \cup^{\mathsf{m}} \{P\}}{(\mathcal{E},\mathcal{I},\sigma) \to (\mathcal{E},\mathcal{I}',\sigma)}$$

[RECV]

$$\frac{inst = \{\mathsf{recv}(a,m); P\} \subseteq^{\mathsf{m}} \mathcal{I},\ \exists\tau \in \Gamma.\nu\tilde{n}.\sigma \vdash m\tau,\ \mathcal{I}' = \mathcal{I} \backslash^{\mathsf{m}} inst \cup^{\mathsf{m}} \{P\tau\}}{(\mathcal{E},\mathcal{I},\sigma) \xrightarrow{\mathsf{K}(m\tau)} (\mathcal{E},\mathcal{I}',\sigma)}$$

Fig. 4: Selection of the transition relations for events

Definition 2 (Protocol specification). *A protocol specification Λ is defined as a tuple $(P, R, \mathcal{T})$ where R is a set of roles, P are lambda expressions built over $\mathcal{T}$. For any statement (event) E in P, it is executed by a role $\pi_{\mathsf{a}}(E) \in R$.*

A protocol is executed by agents who can play any role of the protocol arbitrary times [25], e.g., multiple authenticated clients can establish an unbounded number of sessions with the server. We refer to an execution of an agent in an executing protocol as a *thread*. For any thread, we suppose that there is an unique identifier id attached. We define the *protocol instance* as follows, which is an instantiation of our lambda calculus specification P.

Definition 3 (Protocol Instance). *Let $\Lambda = (P, R, \mathcal{T})$, a protocol instance for Λ is defined as a partial execution of P denoted as P_g with a set of involved threads, i.e., $P_g = P_s\tau$ where τ is a grounding substitution and $P = P'; P_s$.*

A protocol instance *inst* composed of multiple threads. Although the smallest execution unit is the same in both the semantics of our specification Λ and the symbolic model, i.e., a thread, the manner in which they are executed differs. In the operational semantics of the actual symbolic model, each execution step selects a thread for execution (Figure 6 in Section VI gives an example to present this distinction). In contrast, our specification chooses a thread from a specific protocol instance for each step. This approach aligns with how protocols are typically described in natural language texts.

C. Semantics

Following [26], the operational semantics is defined via transition rules[3] between configurations in Figure 4. We define a *configuration* c in our lambda calculus specification as a 3-tuple $(\mathcal{E},\mathcal{I},\sigma)$, where $\mathcal{E}$ is a set of fresh nonces generated during protocol execution, $\mathcal{I}$ is a multiset representing the protocol instances executing in parallel, and σ is a grounding substitution used to model the output message. A basic state transition specifies how the protocol state evolves through the execution of an event, denoted as $c \xrightarrow{l} c'$, where l represents the *trace label* of the transition. The trace label corresponds to the execution of the event E. To formalize this, we introduce a projection function π_{lbl}, which maps the execution of a grounding event E to its trace label, i.e., $\pi_{\mathsf{lbl}}(E) = l$. The traces of our specification Λ is formally defined as:

Definition 4 (Traces). *Given Λ, the set of traces is defined as $traces(\Lambda) = \{l_1 ... l_n \mid c_0 \overset{l_1}{\Longrightarrow} c_1 ... \overset{l_n}{\Longrightarrow} c_n\}$ where relation $\overset{l}{\Rightarrow}$ is defined as $\to_* \xrightarrow{l} \to_*$ [26], which excludes empty labels.*

Intuitively, a trace is a sequence of *observable* (non-empty) events against protocol's execution. The security properties are established with first-order logic formulas over traces.

Algorithm 1: LCCG parser

Input: protocol document nl, length of context N
Output: lambda calculus expressions L

```
1  Let L ← []
2  Segment nl into a list of chunks sequentially: chks ← Segment(nl)
3  M ← len(chks)    ▷ chks = [k_1 k_2 ··· k_M]
4  for i in range(1, M) do
5      Initialize context ctx as an empty list: ctx ← []
6      for j in range(1, N) do
7          blk ← null
8          if i − N + j ≥ 1 then
9              blk ← k_{i−N+j} + L_{i−N+j}
10         Add blk into its context: ctx.append(blk)
11     Prompt LLM with k_i and ctx: λ ← Parse(ctx, k_i)
12     L.append(λ)
13 return L
```

V. LCCG, L-REPAIR, AND REWRITER

A. LCCG: LLM as Parser

Reading protocol documents and understanding protocols' behavior are challenging for large LLMs. One reason is the complexity of protocol behavior, which can be confusing for LLMs lacking domain-specific knowledge. Another reason is that input documents tend to be lengthy (hundreds or even thousands of tokens), affecting the accuracy of LLMs' outputs. Recent research [27] shows that "*LLMs may get lost in the middle*", where they tend to pay less attention to relevant information in the middle portion of the input, leading to a non-robust use of the provided context.

We present an algorithm which harnesses LLM to parse entire protocol documents shown in Algorithm 1. Given a protocol document, we initially divide it into a list of *chunks* (Line 2), and the parsing is performed sequentially (Lines 4-12). Conceptually, a chunk is a block of text that represents a meaningful unit within the protocol, typically corresponding to a distinct message, action, or structural element. Both the chunk size *chk* and context length *N* are configurable parameters that can be adjusted for different needs.

[3]The comprehensive elaboration of the rules is provided in the extended version in [1].

For the parsing of a single chunk, we maintain a context ctx, containing the adjacent N chunks and their associated lambda expressions (Lines 6-10). Utilizing few-shot in-context learning (Line 11), the LLM processes each chunk, captures the semantics of the input with the lambda expressions, and adds them to the memory list L. This chunk-by-chunk approach allows the LLM to concentrate on the immediate context, thereby minimizing the risk of missing intermediate details, as previously discussed. We implement the Parse function (Line 11) with in-context few shots learning. Following [28], [29], we design prompts to instruct the LLM in writing lambda expressions, all of which are provided in [1].

B. L-REPAIR: *Repair the draft*

Before being translated into a SAPIC$^+$ specification, the model generated by LCCG must be made *well-formed* and consistent with natural language documents. However, the initial model is frequently found to be broken for various reasons. In the following, we first give the formal definition of well-formedness of Λ. Then we present common problems of the generated initial model, followed by presenting the corresponding remedy.

Definition 5 (Well-formedness [25]). *Given a specification $\Lambda = (P, R, \mathcal{T})$, we say it is well-formed if (1) any variable in P is bound, (2) any received message is readable for the corresponding role r, and (3) for any statement (event) E in P, its executor $\pi_{\mathsf{a}}(E)$ is a role in R.*

We say that a variable is *bound*, if it is a fresh nonce generated by the role or it occurs in a received message. With the definition above, we summarize three kinds of problems that mainly occur in the initial generated model:

Inconsistency. Due to hallucination and the stochastic nature of LLMs, the initial model may deviate from the natural language description. For example, in a scenario where the server should respond to a client's request with a hash of the secret key, $\mathsf{hash}(\mathsf{k})$, the LLM might erroneously construct a different message, such as $\mathsf{aenc}(\mathsf{hash}(\mathsf{k}), \mathsf{pks})$. Similarly, a message intended to be sent by role A may be incorrectly attributed to role B by the LLM

Ambiguity. As ambiguity is natural in natural language document, it is crucial to employ methods to reduce errors and ensure clarity. According to our observation, the LLM tends to map unclear natural language description to a broken specification with *unbounded* variables, i.e., variables that are used but not declared. We will give more details and explanation in Section VII-A.

Unreadability. The natural language specifies both the protocol's behavior and the message in a global view. However, messages within a 'receive' event should be presented with a local view, detailing the specific contents that the receiver should expect. As outlined in [25], the form of an incoming message in a protocol specification should be *readable* for the specific role, i.e., variables should be accessible. In protocol verifier, pattern matching is based on the message form. An incoming message with inaccessible variables within it can still be accepted by the verifier, but the unintended pattern matching may destroy the semantics.

Algorithm 2: Repair parsing result

```
Input: protocol document nl, parsing result P
1  Prompt LLM to validate the parsing result: P ← Validate(P, nl)
2  LLM revise received message to be readable: P ← View(P)
3  while true do
4      ▷ Repair semi-automatically
5      Get unbounded variables: V ← Analysis(P)
6      if V = ∅ then
7          return P
8      Extract a msc to aid user-interaction: msc ← Graphviz(P)
9      User-interaction P ← Interact(msc, V, 🧍)
10 return P
```

To address the aforementioned problems, we present a repair algorithm described in Algorithm 2. Besides the broken lambda expressions P, the inputs also include the corresponding protocol description nl. To handle *inconsistency* caused by hallucination and stochasticity of LLM, we introduce a self-validation method (Line 1) following [30]. Without introducing extra information, we instruct a LLM to review the specification alongside the given protocol description, finding and correcting the mistakes generated during the parsing stage. For *unreadability* of the term in Λ, we encode a set of examples in the diff [31] format to teach the LLM how to transform a message into a correct form in the role specification step-by-step (Line 2). The function Analysis (Line 5) is implemented using the Lark parser [32]. Given a BNF syntax, the abstract syntax tree (AST) of the specification can be readily obtained and analyzed. The return value of the function Analysis is a set V of unbounded variables arising from the *ambiguity* of natural language or previous black-box steps of the LLM.

When implementing our framework as a self-contained tool, we use a semi-automatic repair method (Lines 8-9). In this case, the lambda expressions P can be directly corrected by the user. The hints for the user's corrections include a

Example 2 (Repairing lambda expressions). *We use the diff format shown below to demonstrate how lambda calculus expressions can be corrected for further rewriting. In this example, two errors are fixed: the first, on Line 5, is caused by the ambiguity of 'it' in Line 2; the second, on Line 9, results from the unreadability of the received message.*

```
1   /*The client generates a key k, encrypts it with public
2   key of server, then sends it to server S */
3   @gen(C,k)
4   @op(C, @aenc(k, pkS))
5 - @send(C,k)                  // Ambiguity of 'it'
6 + @send(@aenc(k, pkS))
7   /*The server S confirms the receipt by sending the hash
8   key of server, then sends it to server S */
9 - @recv(@aenc(k, pkS))    // The message is not readable
10 + @recv(S, req)
11 + @op(S, @binds(k, @adec(req, skS)))
12  @send(S, @hash(k))
13  @recv(C, @hash(k))
```

$$\mathbf{T}(P) \triangleq \bigcup_{r \in R} \mathbf{T}(p, r, \emptyset) \quad \textbf{(T1)}$$

$$\mathbf{T}(\mathsf{knows}(r, \bar{t}); p, r, M) \triangleq \mathsf{let}\ r(\bar{t}) = \mathbf{T}(p, r, M \cup \{\bar{t}\}) \quad \textbf{(T2)}$$

$$\mathbf{T}(\mathsf{Cond}\ \alpha\ p, r, M) \triangleq \mathsf{if}\ \alpha\ \mathsf{then}\ \mathbf{T}(p, r, M) \quad \textbf{(T3)}$$

$$\mathbf{T}(\mathsf{gen}(r, n); p, r, M) \triangleq \mathsf{new}\ n; \mathbf{T}(p, r, M \cup \{n\}) \quad \textbf{(T4)}$$

$$\mathbf{T}(\mathsf{send}(r, m); p, r, M) \triangleq \mathsf{out}(m); \mathbf{T}(p, r, M) \quad \textbf{(T5)}$$

$$\mathbf{T}(\mathsf{recv}(r, t); p, r, M) \triangleq \mathsf{in}(t); \mathbf{T}(p, r, M \cup \{t\}) \quad \textbf{(T6)}$$

$$\mathbf{T}(\mathsf{op}(r, \mathsf{binds}(t_1, t_2)); p, r, M) \triangleq \mathsf{let}\ t_1 = t_2\ \mathsf{in}\ \mathbf{T}(p, r, M \cup \{t_1\}) \quad \textbf{(T7)}$$

$$\mathbf{T}(\mathsf{op}(r, f(\bar{t})); p, r, M) \triangleq \mathsf{event}\ F(\bar{t}); \mathbf{T}(p, r, M) \quad \textbf{(T8)}$$

Fig. 5: Rewriting rules $\mathbf{T}$

graphical message sequence chart (MSC) of the protocol (Line 8) and reports of unbounded variables from program analysis. The user (denoted by 👤) can directly edit the intermediate result (Line 9). When evaluating our overall approach for automatic modeling in Section VII-C, we do not allow any user interaction for repairing.

C. Rewriter: From lambda calculus to SAPIC$^+$

Rewriting rules. The local role processes $\bigcup_r \mathcal{P}_r$ can be extracted from our protocol specification Λ, where r denotes a role in a SAPIC$^+$ specification. In the following, we propose a set of rules $\mathbf{T}$ shown in Figure 5, which rewrite our lambda expressions into statements in the SAPIC$^+$ specification. The soundness of $\mathbf{T}$ will be discussed in Section VI. Our lambda calculus specification is role-oriented, i.e., every statement includes a role variable which indicates who executes the action. Before introducing the rewriting rules, we first define *role specification* which corresponds to a local process in SAPIC$^+$ specification as follow:

Definition 6 (Role specification). *Given a specification $\Lambda = (P, R, \mathcal{T})$ and a role variable $r \in R$, we define the corresponding role specification as a 3-tuple (p, r, M), where p is a sequence of lambda expressions with the same role index r and M is the initial knowledge set containing the variables that can be used initially.*

The top rule **T1** in Figure 5 states that the rewriting result of the lambda expressions P is a disjoint set of rewriting results of all role specifications extracted from P. **T2–8** describe the recursive application of $\mathbf{T}$ to derive a local process from a role specification. Specifically, Rule **T2** transforms the event knows into the local process's signature; Rules **T5–6** modify I/O events; and **T7–8** convert op events into corresponding let-bindings and signal events. Our rewriting rules $\mathbf{T}$ are defined under a default context. We assume that: (1) The threat model is Dolev-Yao [33], where an adversary can inject, modify and intercept messages on the network. (2) The initial knowledge set of role is an empty set.

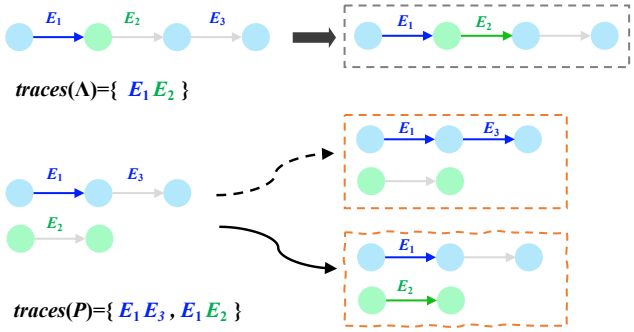


Fig. 6: Execution models of our lambda calculus Λ and SAPIC$^+$ specification $\mathcal{P}$. There are two distinct threads t_1 and t_2, each playing a different role. In Λ, two threads compose a protocol instance, and the execution order of events is constrained by the lambda expressions P. In SAPIC$^+$ specification $\mathcal{P}$, however, two threads execute in parallel. Considering the two-step traces, we have $traces(\Lambda) = \{E_1E_2\}$ and $traces(\mathcal{P}) = \{E_1E_3,\ E_1E_2\}$.

Top specification synthesis. After applying translation rules for each role specification, the local processes $\bigcup_r \mathcal{P}_r$ can be obtained, which corresponds to the role's behaviors portrayed in textual protocol description. However, as defined in Section II, a complete verifiable symbolic model should also include the protocol's initialization, which we call *top* specification $\mathcal{P}_\mathsf{t}$. We first collect all of the signatures of the local process $\mathcal{P}_r$, then incorporate in-context few-shot learning to teach LLMs how to give a top specification $\mathcal{P}_\mathsf{t}$. Now the complete specification file, $\bigcup_r \mathcal{P}_r \cup \mathcal{P}_\mathsf{t}$, can be obtained and then compiled to different protocol verifiers. Each part of these translations is proven to be sound, which is also utilized to establish our overall soundness.

VI. Soundness Discussion

In this section, we (1) establish trace inclusion relation between different models presented above. Then, we (2) show that for any safety property which is verified by the prover also holds on λ-DSL model and SAPIC$^+$ model. The overall soundness is intuitively implied combining (1) and (2).

We first prove the soundness of the transformation $\mathbf{T}$ from DSL specification Λ to SAPIC$^+$ process $\mathcal{P}$, following the approach in [14]. The lemma holds if $traces(\Lambda) \subseteq traces(\mathcal{P})$. The intuition behind this can be explained as follows: the traces of Λ are constrained by the event order defined globally, while the traces of $\mathcal{P}$ are defined over the local event order within the role specification. The global event order implies the local one, thereby establishing a trace inclusion relationship between Λ and $\mathcal{P}$. We present a simple example of both execution models of Λ and $\mathcal{P}$ to give an intuition in Figure 6.

Lemma 1 (Soundness of rewriting). *Let Λ be a protocol specification of our DSL, and φ a safety trace property, then*

$$\mathcal{P} \vDash \varphi \ \Rightarrow\ \Lambda \vDash \varphi$$

where $\mathcal{P} = [\![\Lambda]\!]^{\mathbf{T}}$ denotes the process rewritten from Λ.

Theorem 1 (Soundness of compilation [12]). *Let $\mathcal{P}$ be a process in* SAPIC$^+$ *and φ a trace property formula, then*

$$\mathcal{P} \vDash \varphi \;\Leftrightarrow\; [\![\mathcal{P}]\!]^{\langle\rangle} \vDash [\![\varphi]\!]^{\langle\rangle}$$

where $[\![\mathcal{P}]\!]^{\langle\rangle}$ denotes the low-level symbolic models compiled from process $\mathcal{P}$. $\langle\rangle$ denotes three verifiers: TAMARIN, PROVERIF and DEEPSEC.

Corollary 1. $[\![\mathcal{P}]\!]^{\langle\rangle} \vDash [\![\varphi]\!]^{\langle\rangle} \;\Rightarrow_{\mathsf{Thm.1}}\; \mathcal{P} \vDash \varphi \;\Rightarrow_{\mathsf{Lem.1}}\; \Lambda \vDash \varphi$

Theorem 1 has been proven in [12], which demonstrates the soundness of the compilation. Combining Lemma 1, we can obtain the overall soundness result (Corollary 1) between three models, which has been explained earlier in this section. The detailed proof of Lemma 1 is included in our extended paper version in [1].

VII. EVALUATION AND TOOL IMPLEMENTATION

In this section, we first evaluate our approach, including both the parsing and repairing methods. Next, we construct a benchmark and evaluate our overall approach on it. Additionally, we implement a self-contained tool with frontend to make our method accessible. The LLMs we used to conduct experiments include GPT-3.5-turbo, GPT-4, GPT-4o and Google Gemini-pro, with a temperature setting of 0.4 for both semantic parsing and automated repairing. All of the generated symbolic models run on a TAMARIN prover in version 1.8.0, which includes a SAPIC$^+$ platform.

A. Intermediate Language Extraction Evaluation

We first evaluate how much of the protocol document input's semantics can be captured by our intermediate language.

Methodology. We evaluate the output of LCCG parser in 8 different protocols as shown in Table III. The protocol documents are sourced from various origins, including IETF RFCs [34], [16], academic papers [35], [36], the TAMARIN manual [37], and informal texts such as teaching assignments [38]. Prior to parsing these documents with LCCG, we manually segment each document into a list of chunks.

Ground truth and metrics. For each protocol, if a corresponding SAPIC$^+$ model $\mathcal{P}$ is not available in [18], we manually construct it. We then provide lambda expressions as the ground truth, ensuring that they are well-formed and that SAPIC$^+$ model $\mathcal{P}$ can be rewritten from these expressions using $\mathbf{T}$. We use three ***metrics*** to quantify the parsing result:

- *Exact Coverage (EC)*: which measures the coverage of the generated expressions over the ground truth, calculated as $EC = n_{\mathsf{g}}/N$;
- *Bounded expressions rate (BER)*: which evaluates the bounded expressions rate, calculated as $BER = n_{\mathsf{b}}/N$;
- *Error Rate (ER)*: which measures the error rate of the parsing result, calculated as $ER = 1 - EC$. The error may include **#1.** expressions outside the ground truth (e.g., an extra send event), and **#2.** ground truth expressions that are incorrectly expressed (e.g., a wrong message),

TABLE II: Original protocol documents used for evaluation.

Protocol	Document Source	Size
NSPK	Online Teaching Assignment [38]	210
Toy	Exercises for starting with TAMARIN	226
NSSK	Needham–Schroeder, Wikipedia [39]	392
NAXOS	TAMARIN Manual [2], p36	234
Otway-Rees	GIAC Certification Paper [36]	351
SSH	RFC 4253 [34], Section 7.2 and 8	991
IKEv2	ACSAC'21 [40], Section 3, p3	379
KEMTLS	CCS'20 [35], Section 3	659
EDHOC	draft-ietf-lake-edhoc-02 [41], p20	1029

where n_{g} and n_{b} denote the number of lambda expressions within ground truth and bounded expressions[4], respectively. N is the total number of generated expressions.

The parsing results with GPT-4 model are shown in Figure 7. It is straightforward to see that generating well-formed lambda expressions in one round is hard, every parsing results contain errors and the exact coverage ranges from 9.09% (case Toy) to 87.56% (case EDHOC). The syntax errors tend to occur in long lambda expressions, e.g., when multiple expressions are nested, leading to mismatches in parentheses. Reviewing the original documents with generated lambda expressions, we got several interesting observations and findings.

#Finding 1. For those smaller cases whose protocol descriptions are explicit and specific (e.g., NSSK [39], Otway-Rees [36]), our lambda expression is effective to capture their semantics, with over 54.55% *EC*. Most of the generated expressions are bounded, with over 78.26% *BER*. For these cases, the main problem is the unreadibility of message in recv event, which will be resolved in the repairing phase.

#Finding 2. For those complex protocols, a singular protocol document is not always self-contained enough to derive a symbolic model from it. Considering the cases of IKEv2 [40], EDHOC [16], and KEMTLS [35] in our evaluation, the *EC* value ranges from only 12.44% to 29.17%. There are many unbounded variables (about 20%) in their corresponding generated lambda expressions. Take the KEMTLS protocol as an example. A snippet of the original paper we used for parsing is:

> **KEMTLS, CCS'20 [35]** **Section 3, para 5**
>
> Phase 2: KEMTLS follows the TLS 1.3 key schedule, which applies a sequence of HKDF operations to the shared secret sse and the transcript to derive Client and Server traffic secrets CHTS and SHTS.

A lambda expression parsed from the protocol snippet is:

$$\mathsf{op}(\mathsf{Client}, \mathsf{binds}(\mathsf{CHTS}, \mathsf{HKDF}(\mathsf{sse}, \mathsf{transcript}))),$$

where variable transcript is unbounded. As explained in Section V-B, unbounded variables may indicate ambiguities or implicitness in the protocol documentation. In fact, [35] does not provide further description for variable transcript,

[4]A bounded expression refers to the term *grounding event* defined in Section IV-B. Specifically, it means there are no unbounded variables in the expression.

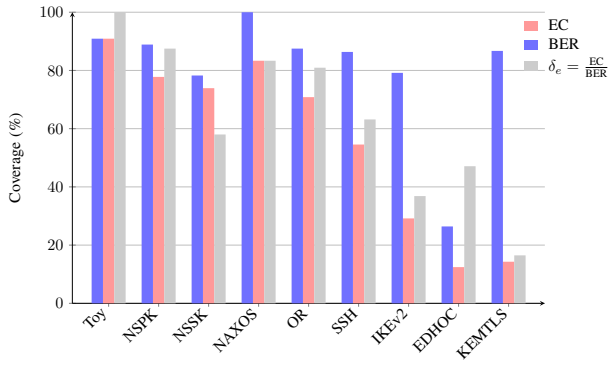


Fig. 7: Parsing results for 8 protocols with GPT-4 model.

which confuses LLM to make a correct modeling choice. Conversely, our approach can localize such problems effectively with syntactic check techniques, i.e., detection for unbounded variables, which is implemented as Analysis function.

#Finding 3. The metric *BER* reflects whether the lambda expressions can be rewritten to SAPIC$^+$ specification (as defined in [26], a process can be compiled only when there is no unbounded variable). Another interesting finding we got is that the value $\delta_e = EC/BER$ reflects the degree of abstraction in natural language descriptions relative to symbolic models to some extent: the greater the value of δ_e is, the more specific the natural language descriptions are indicated. Take the protocol IKEv2 as an example, whose parsing result on *EC*, and *BER* are 29.17%, 79.17% respectively. A chunk of the original paper we used for parsing is:

> **IKEv2, ACSAC'21 [40]** **Section 3, page 3**
>
> IKE_SA_INIT_Responder: The Responder also chooses a private ephemeral key eR and uses it to calculate the shared DHkey. Together with $\mathsf{nonce_r}$ and $\mathsf{nonce_i}$, this shared key is used to derive the session key (called keymat).

One of the parsing result for this chunk is:

$$\mathsf{gen(Responder, eR);}$$
$$\mathsf{op(Responder, binds(epR, pk(eR)));}$$
$$\mathsf{op(Responder, binds(DHkey, func(eR))),}$$

where all variables are bounded. But in fact, the parsing result does not cover any lambda expression in ground truth, because the natural language is not specific. Consequently, the language model cannot determine how to 'calculate the shared DHkey' and resorts to a generic operation func. The protocols EDHOC and KEMTLS have the same problem, whose δ_e are 47.10% and 16.48% respectively.

Different LLMs for parsing. We also evaluate the parsing with other LLMs. Intuitively, the results show that advanced models have better capabilities of understanding semantics. Specifically, advanced LLMs are less likely to construct incorrect terms within lambda expressions, which reduces the second type errors in metric *ER*. Among them, GPT-4 performs the best, with an average *EC* of 56.36% and an average *BER* of 80.47% across all cases.

TABLE III: Parsing results with GPT-4 model.

Protocol	Err. #1	Err. #2	EC	BER	δ_e
Toy	00.00%	9.09%	90.91%	90.91%	100.0%
NAXOS	00.00%	16.67%	83.33%	100.0%	83.33%
NSPK	11.11%	11.11%	77.78%	88.89%	87.50%
NSSK	17.39%	8.70%	73.91%	78.26%	94.44%
Otway-Rees	20.83%	8.33%	70.83%	87.50%	80.94%
SSH	18.18%	27.27%	54.55%	86.36%	63.17%
IKEv2	25.00%	45.83%	29.17%	79.17%	36.84%
EDHOC	20.45%	67.11%	12.44%	26.41%	47.10%
KEMTLS	10.71%	75.00%	14.29%	86.71%	16.48%

Summary of parsing evaluation. The evaluation of the parsing shows that our lambda expressions provide a way for quantitative analysis of document to determine if it is adequate to derive a symbolic model, and further, if it is specific enough to derive a strong model for analysis. Given a document as input, our LCCG parser is capable of providing a more detailed understanding. Even when the input falls short of delivering sufficiently effective information, our approach is still capable of identifying these issues using lambda expressions.

B. Repairing Evaluation

Methodology. Continuing parsing evaluation, we evaluate how the broken expressions can be repaired by LLMs. The effectiveness of repairing is also influenced by its input (the parsing result), so in this evaluation we use the same lambda expressions of every cases as input. Besides, we do not introduce any user-interaction.

Different LLMs for repairing. We evaluate the repairing with different models. The result (included in [1]) shows that the effectiveness of repairing relies on advanced LLMs. For most complex cases, automatic repairing is not enough to resolve all the issues (reduce error average 14.74% with GPT-4). That's the reason why it is necessary to introduce user-interaction when deploying our tool in practice.

C. Overall Approach Evaluation

Benchmark construction. We extend the protocol cases evaluated above and construct a standardized benchmark for evaluating our overall approach, as well as for future proposals. In this work, we focus on protocol cases of manageable scale, excluding extremely large-scale protocols like TLS 1.3 and 5G AKA. The benchmark $D = \{(N_i, \mathcal{P}_i, \mathcal{R}_i) \mid i \in \mathbb{N}\}$ is a set of 3-tuple, where N_i is the protocol document, $\mathcal{P}_i$ is the SAPIC$^+$ specification, and $\mathcal{R}_i$ is the TAMARIN model. The construction mainly consists of the following three steps:

(1) We start from $\mathcal{R}_i$ which has been available at TAMARIN GitHub repository.
(2) To get their corresponding N_i, we read these protocols' original documents e.g., RFCs, extract and reform the core parts corresponding to the model.
(3) For the given model $\mathcal{R}_i$, there are a set of safety properties Φ which have been verified on $\mathcal{R}_i$. We manually build SAPIC$^+$ model $\mathcal{P}_i$, ensuring that $\mathcal{P}_i \approx_\varphi \mathcal{R}_i$ satisfies under every property $\varphi \in \Phi$.

TABLE IV: Statistics of Benchmark.

Protocol	Success	Ratio	LoC_S	LoC_M	Prop	Time (s)
example	✓	5/5	40	54	2	0.56
Toy	✓	5/5	37	44	3	0.23
NSPK	✓	3/5	100	333	4	22.62
NSSK	✓	3/5	117	320	7	6.26
SigFox	✓	4/5	30	95	3	1.14
LAKE	✓	2/5	46	78	6	90.36
NAXOS	✗	0/5	44	87	3	23.24
X509.1	✓	4/5	66	180	4	8.66
SSH	✗	0/5	74	342	5	4.23
EDHOC	✗	0/5	91	707	10	31.38
KEMTLS	✗	0/5	117	722	5	24.68
Yahalom	✓	2/5	104	396	7	7.89
Kao Chow	✓	2/5	80	334	6	6.83
SPLICE/AS	✗	0/5	155	501	7	115.06
Otway Rees	✓	1/5	163	314	9	35.09
Woo and Lam	✓	3/5	81	195	5	1.96
Denning-Sacco	✓	2/5	73	156	5	1.91
Stubblebine	✗	0/5	70	388	7	9.05

Definition 7 (Semantic equivalent). *Given two symbolic models $\mathcal{R}_1$ and $\mathcal{R}_2$, we say they are semantic equivalent w.r.t. property φ, i.e., $\approx_\varphi$, if φ holds on both symbolic models.*

We define an approximate and non-strict equivalence relation between two symbolic models in Definition 7, which is used to evaluate the correctness of our generated model. Similar to program testing where a program is considered correct if it can pass all test cases, we take the property φ as the test case here. In Table IV, the mark ✓ denotes that generated model can pass (be verified) all cases (properties).

Benchmark evaluation. We evaluate our approach using a benchmark constructed above, with detailed information provided in Table IV. The input texts vary in size from 51 to 639 tokens, and the lines of codes (LoC_S for SAPIC$^+$, LoC_M for TAMARIN) range from 30 (44) to 163 (722) respectively. We have verified a list of basic security properties, including secrecy, authentication on these protocols. The number of the properties are listed under Prop item. In this experiment, we do not introduce any user-interaction and we set the run iterations as 5 for each protocol case. The results show that we can get a correct model in 10 out 18 cases automatically, with success ratio ranging from 2/5 to 5/5. It is not surprising that some cases failed. As discussed in our two findings (#Finding 2 and #Finding 3) in Section VII-A, these failed cases demonstrate the reliability of our approach: it aims to avoid generating seemingly correct but misleading results from a document that lacks sufficient information.

Summary of Comparisons. We take the few-shot learning as the baseline. As shown in Figure 8, while 3-shot learning generates a maximum of 4 cases, our method shows superior performance, automatically generating correct models in 10 out of 18 cases.

D. Comparison with Correct-by-Construction Approach

To provide complementary evidence of the correctness of our approach, we compare it with another *correct-by-construction* method [8]. There have been several studies

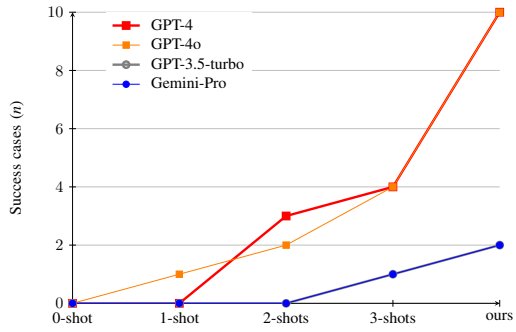


Fig. 8: Comparison of few-shot learning with our method

(e.g., [8], [10], [42]) that utilize the commonly used protocol specification—*Alice&Bob* notation, as input and then generate a symbolic model from it. As our target model is the TAMARIN model, we have chosen [8] for our comparison. We take four protocol cases as examples for our experiments. For each case, we manually construct a corresponding Alice&Bob specification as the input for [8]. After obtaining the TAMARIN model $\mathcal{R}_{\mathsf{anb}}$ from its output, we employ the same evaluation method defined in Definition 7. We verify the safety properties Φ, which are already satisfied by $\mathcal{R}$, against $\mathcal{R}_{\mathsf{anb}}$ to confirm that they also hold in $\mathcal{R}_{\mathsf{anb}}$, i.e., $\mathcal{R}_{\mathsf{anb}} \approx \mathcal{R}$. According to experiment data in [1], given a set of properties Φ, both the symbolic models generated from [8] and our approach can be successfully verified, which demonstrates their equivalence under our definition. All of the baseline models $\mathcal{R}_{\mathsf{anb}}$ are also included in [1].

E. Tool Implementation.

We implement our approach for automatic modeling as a Web-based frontend with HTML and JavaScript. Our benchmark and tool implementation are released on [1].

VIII. DISCUSSION

In this section, we discuss how the trustworthiness of the generated models is ensured by our approach. Other issues, such as model size, threats to validity, *etc.*, are also included.

Guarantee for trustworthiness. At a high-level, we aim to achieve a goal that is similar to that of *correct-by-construction* model synthesis [43]. The difficulty is however the lack of a formal-enough requirement to start with, and without such requirement it is difficult to ensure the strict correctness of the overall process (i.e., from natural language to generated model). That is why we employ LLM to give us some starting point, and then employ an approach similar to that of correct-by-construction model synthesis (i.e., the subsequent transformation). As demonstrated by our *design philosophy*: we need to validate some intermediate results that are intuitive and easy to understand. Meanwhile, we design algorithms and construct proofs to ensure the consistency between these intermediate results and the final output. We believe this effort is inevitable (and hopefully minimal) to ensure the overall correctness as this is the most intuitive representation for

humans to understand (compared to validating the symbolic model directly).

Difficulty of aligning data. We give a further explanation about *"LLMs are unfamiliar with formal specification languages"* mentioned in Section III. While papers ([4], [5]) and open-source models ([44], [45]) could serve as potential pretraining data for LLMs, directly using such datasets remains challenging due to the difficulty of perfectly aligning the documents with the symbolic model. Take TLS 1.3 paper [4] as an example, the authors provide a modeling note with side-by-side comparison of the specification and the model to show how a symbolic model is constructed from the document, which includes certain key 'simplifications' and 'assumptions'. In most cases, such notes are not available, but are the key to align the document with the human-written models. Even with these notes, there is no perfect alignment between them.

Model size and overspecification. The generated models exhibit *size* issues *w.r.t* human-created models, often being larger due to the compilation from SAPIC$^+$ to the final TAMARIN model. In our case studies, the ratio of code size (lines of code) in the generated model to that in reference human-created model roughly ranges from 1.3 to 5.4. Though optimizations of the compilation to reduce the model size is one of the highlighted contributions in [12], our results show that the optimization still has room for improvement. With respect to *overspecification*, our approach leverages a DSL designed to formally model the semantics of natural language sentences through lambda expressions. This design is intended to capture the core semantics without over-analyzing or adding implicit details that are not explicitly stated, thereby mitigating the risk of overspecification.

Threats to validity. (1) *Internal validity.* One potential threat to internal validity is the *types of evaluation protocols*. The protocol cases used in this work primarily involve classic symmetric and asymmetric cryptographic primitives, which are well-understood by LLMs. However, protocols incorporating other cryptographic primitives may exhibit different characteristics and should be considered to ensure a more comprehensive evaluation. (2) *External validity.* A key threat to external validity is our *reliance on GPT-4*. As shown in Section VII, the reproducibility of our results depends on an advanced closed-source LLM (GPT-4), which may undergo updates over time. While the continuous release of symbolic models online could improve performance for both our methods and few-shot prompting, this dependency limits generalizability. To mitigate this, open-source LLMs should be considered.

Limitations. The first limitation lies in the gap between evaluation cases and real-world large scale protocols like TLS 1.3. For a complex protocol case, there could be long-distance and non-sequential dependencies in the protocol document, which makes our parsing algorithm suffer. Another limitation is concerned about our intermediate language. Ideally, we would like our DSL can be as expressive as the low-level SAPIC$^+$ specification. But in this work, we only cover a core subset of its features.

IX. Related Work

Program synthesis. The symbolic model in this work functions as a specialized program. There has been considerable research on synthesizing domain-specific programs from natural language documents, including SQL queries [46] and bash commands [47]. Recent advances in large language models have notably enhanced program synthesis [48], [49], [50], [51], [52], providing substantial data for training and facilitating end-to-end synthesis for most programs. However, data for symbolic model synthesis are scarce, which is the primary reason why our approach differs from these types of program synthesis. We believe that applying our approach to generate symbolic models from natural language presents a valuable method for obtaining high-quality data for future research.

LLM-aided formal verification. We divide formal verification into three phases: (**P1**) build a formal model for the system to be verified, (**P2**) write formal specifications that are expected to be satisfied by the model constructed in **P1**, and (**P3**) prove with theory and tools. While our work focuses on **P1**, there are many works emerging to assist other phases. For **P2**, there are many kinds of formal specifications, e.g., linear temporal logic. In [53], the authors used LLMs to generate temporal logics and fine-tuned a T5 model [54] with generated datesets. [55] and [29] involve user interaction to correct the sub-formula of linear temporal logic. There are also many explorations to involve LLM in **P3** to assist with theorem proving, including [56], [57], [58].

X. Conclusion and Future Work

In this work, we design and implement a novel framework for symbolic protocol model generation with the help of LLMs. The insight of our framework is that we break down the entire process into multiple steps, where each defined within a formal model and refinement relations are established between them. Having conducted case studies on real-world protocols, the results show our method is effective for automatic modeling and the intermediate results can provide evidence for the trustworthiness of the generated model.

Our results suggest future directions. First, we argue that our approach can benefit from fine-tuning on domain-specific data. Fine-tuning an open-source LLM with such data cannot only reduce hallucinations but also enhance the accuracy of the generated models. Another direction is indicated by our #Finding 2. While the input document is not self-contained enough to derive a symbolic model, the subsequent manual efforts are still required. How to guide LLM make subsequent responses could benefit the automation of the overall method.

Acknowledgements

This research was supported by the Key R&D Program of Zhejiang under Grant No. 2025C01083, and the CCF-Huawei Populus Grove Fund (CCF-HuaweiFM202204). This research was also supported by the National Natural Science Foundation of China (Project No. 62302375), and the Ministry of Education, Singapore under its Academic Research Fund Tier 3 (Award ID: MOET32020-0004).

REFERENCES

[1] Z. Mao, J. Wang, J. Sun, S. Qin, and J. Xiong, "Tool implementation, benchmark, and extended paper version," https://github.com/zerrymore/AutoSM.

[2] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The tamarin prover for the symbolic analysis of security protocols," in *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*. Springer, 2013, pp. 696–701.

[3] B. Blanchet, "An efficient cryptographic protocol verifier based on Prolog rules," in *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11-13 June 2001, Cape Breton, Nova Scotia, Canada*, 2001, pp. 82–96.

[4] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe, "A comprehensive symbolic analysis of tls 1.3," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1773–1788.

[5] D. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler, "A formal analysis of 5g authentication," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 1383–1396.

[6] D. Basin, R. Sasse, and J. Toro-Pozo, "The emv standard: Break, fix, verify," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1766–1781.

[7] "Needham-schroeder public-key protocol - non-termination." [Online]. Available: https://groups.google.com/g/tamarin-prover/c/k9zpOI_fxq0/m/q-C-WdhOCgAJ

[8] M. Keller and P. D. D. Basin, "Converting alice&bob protocol specifications to tamarin," *ETH Zurich*, 2014.

[9] R. Metere and L. Arnaboldi, "Automating cryptographic protocol language generation from structured specifications," in *Proceedings of the IEEE/ACM 10th International Conference on Formal Methods in Software Engineering*, 2022, pp. 91–101.

[10] P. Modesti, "Anbx: Automatic generation and verification of security protocols implementations," in *Foundations and Practice of Security: 8th International Symposium, FPS 2015, Clermont-Ferrand, France, October 26-28, 2015, Revised Selected Papers 8*. Springer, 2016, pp. 156–173.

[11] J. Yen, T. Lévai, Q. Ye, X. Ren, R. Govindan, and B. Raghavan, "Semi-automated protocol disambiguation and code generation," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 272–286.

[12] V. Cheval, C. Jacomme, S. Kremer, and R. Künnemann, "SAPIC$^+$: protocol verifiers of the world, unite!" in *31st USENIX Security Symposium*. Boston, MA, USA: USENIX Association, 2022, pp. 3935–3952.

[13] V. Cheval, S. Kremer, and I. Rakotonirina, "Deepsec: deciding equivalence properties in security protocols theory and practice," in *2018 IEEE symposium on security and privacy (SP)*. IEEE, 2018, pp. 529–546.

[14] C. Sprenger, T. Klenze, M. Eilers, F. A. Wolf, P. Müller, M. Clochard, and D. Basin, "Igloo: Soundly linking compositional refinement and separation logic for distributed system verification," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–31, 2020.

[15] R. Rojas, "A tutorial introduction to the lambda calculus," 2015. [Online]. Available: https://arxiv.org/abs/1503.09060

[16] C. Jacomme, E. Klein, S. Kremer, and M. Racouchot, "A comprehensive, formal and automated analysis of the edhoc protocol," in *USENIX Security'23-32nd USENIX Security Symposium*, 2023.

[17] C. Caleiro, L. Vigano, and D. Basin, "On the semantics of alice&bob specifications of security protocols," *Theoretical Computer Science*, vol. 367, no. 1-2, pp. 88–122, 2006.

[18] Tamarin Prover Team, "Tamarin prover," https://github.com/tamarin-prover/tamarin-prover.

[19] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," Internet Engineering Task Force, Tech. Rep., 2017. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-tls-tls13/21/

[20] 3GPP, "Security architecture and procedures for 5g system." 2018.

[21] EMVCo, *EMV Contactless Specifications for Payment Systems, Book C-2, Kernel 2 Specification, Version 2.8*, April 2019.

[22] ——, *EMV Contactless Specifications for Payment Systems, Book C-3, Kernel 3 Specification, Version 2.8*, April 2019.

[23] Y. Artzi, N. FitzGerald, and L. Zettlemoyer, "Semantic parsing with combinatory categorial grammars." *ACL (Tutorial Abstracts)*, vol. 3, 2013.

[24] R. Gil-Pons, R. Horne, S. Mauw, A. Tiu, and R. Trujillo-Rasua, "Is eve nearby? analysing protocols under the distant-attacker assumption," in *2022 IEEE 35th Computer Security Foundations Symposium (CSF)*. IEEE, 2022, pp. 17–32.

[25] C. Cremers and S. Mauw, "Operational semantics of security protocols," in *Scenarios: Models, Transformations and Tools*, S. Leue and T. J. Systä, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 66–89.

[26] S. Kremer and R. Künnemann, "Automated analysis of security protocols with global state," *Journal of Computer Security*, vol. 24, no. 5, pp. 583–616, 2016.

[27] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the Middle: How Language Models Use Long Contexts," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157–173, 02 2024.

[28] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[29] M. Cosler, C. Hahn, D. Mendoza, F. Schmitt, and C. Trippel, "nl2spec: interactively translating unstructured natural language to temporal logics with large language models," in *International Conference on Computer Aided Verification*. Springer, 2023, pp. 383–396.

[30] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proceedings of the ACM on Programming Languages (PACMPL), Issue OOPSLA*, 2024.

[31] B. W. Kernighan and R. Pike, *The UNIX Programming Environment*. Prentice-Hall, 1984.

[32] E. Shinan and other contributors, "Lark: A modern parsing library for python," https://github.com/lark-parser/lark.

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

[34] T. Ylonen, "Rfc 4253: The secure shell (ssh) transport layer protocol," 2006.

[35] P. Schwabe, D. Stebila, and T. Wiggers, "Post-quantum tls without handshake signatures," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1461–1480.

[36] D. Otway and O. Rees, "Efficient and timely mutual authentication," *ACM SIGOPS Operating Systems Review*, vol. 21, no. 1, pp. 8–10, 1987.

[37] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[38] G. Lowe, "An attack on the needham- schroeder public- key authentication protocol," *Information processing letters*, vol. 56, no. 3, 1995.

[39] Wikipedia contributors, "Needham–Schroeder protocol." [Online]. Available: https://en.wikipedia.org/wiki/Needham-Schroeder_protocol

[40] S.-L. Gazdag, S. Grundner-Culemann, T. Guggemos, T. Heider, and D. Loebenberger, "A formal analysis of ikev2's post-quantum extension," in *Proceedings of the 37th Annual Computer Security Applications Conference*, 2021, pp. 91–105.

[41] G. Selander, J. P. Mattsson, and F. Palombini, "Ephemeral Diffie-Hellman Over COSE (EDHOC)," Internet Engineering Task Force, Internet-Draft draft-ietf-lake-edhoc-02, Nov. 2020. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-lake-edhoc/02/

[42] Y. Zhao, H. Jiang, J. Lv, S. Tan, and Y. Li, "Anb2murphi: A translator for converting alicebob specifications to murphi." in *International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2021, pp. 108–113.

[43] E. W. Dijkstra, "A discipline of programming," 1976.

[44] D. Basin, J. D. L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler, "Tamarin model of 5g aka v15.0.0." [Online]. Available: https://github.com/tamarin-prover/tamarin-prover/blob/develop/examples/ccs18-5G/5G-AKA-bindingChannel/5G_AKA.spthy

[45] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe, "Tamarin models for tls 1.3 draft 21." [Online]. Available: https://github.com/tls13tamarin/TLS13Tamarin/tree/master/src/rev21

[46] J. Guo, Z. Zhan, Y. Gao, Y. Xiao, J.-G. Lou, T. Liu, and D. Zhang, "Towards complex text-to-SQL in cross-domain database with intermediate representation," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, 2019.

[47] X. V. Lin, C. Wang, D. Pang, K. Vu, and M. D. Ernst, "Program synthesis from natural language using recurrent neural networks," *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-17-03-01*, 2017.

[48] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[49] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[50] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[51] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *Journal of Machine Learning Research*, vol. 24, no. 240, pp. 1–113, 2023.

[52] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," 2023. [Online]. Available: https://arxiv.org/abs/2203.13474

[53] Y. Chen, R. Gandhi, Y. Zhang, and C. Fan, "NL2TL: Transforming natural languages to temporal logics using large language models," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Singapore: Association for Computational Linguistics, 2023, pp. 15 880–15 903.

[54] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020.

[55] I. Gavran, E. Darulova, and R. Majumdar, "Interactive synthesis of temporal specifications from examples and natural language," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–26, 2020.

[56] S. Polu and I. Sutskever, "Generative language modeling for automated theorem proving," 2020. [Online]. Available: https://arxiv.org/abs/2009.03393

[57] H. Wang, Y. Yuan, Z. Liu, J. Shen, Y. Yin, J. Xiong, E. Xie, H. Shi, Y. Li, L. Li *et al.*, "Dt-solver: Automated theorem proving with dynamic-tree sampling guided by proof-level value function," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2023, pp. 12 632–12 646.

[58] K. Yang, A. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. Prenger, and A. Anandkumar, "LeanDojo: Theorem proving with retrieval-augmented language models," in *Neural Information Processing Systems (NeurIPS)*, 2023.