SYSLITE: Syntax-Guided Synthesis of PLTL Formulas from Finite Traces

M. Fareed Arif, Daniel Larraz, Mitziu Echeverria, Andrew Reynolds, Omar Chowdhury, Cesare Tinelli
Department of Computer Science, The University of Iowa

Abstract—We present an efficient approach to learn past-time, propositional linear temporal logic formulas (PLTL) from a set of propositional variables and a sample of finite traces over those variables. The efficiency of our approach can be attributed to a careful encoding of the PLTL formula learning problem as a bit-vector function synthesis problem, and the use of an enhanced Syntax-Guided Synthesis (SyGuS) engine to solve the latter. We implemented our approach in a tool called SYSLITE and empirically evaluated its efficacy with two case studies. In these case studies, we observe that SYSLITE on average enjoys a speedup of 44x over current learning approaches for temporal formulas while learning the expected formulas in the vast majority of cases.

I. INTRODUCTION

This paper focuses on the problem of synthesizing past-time, propositional linear temporal logic (PLTL) formulas when given an alphabet (i.e., a set of propositional variables) and a sample of finite traces as inputs. The input sample consists of a set of positive traces and a disjoint set of negative traces. The synthesized PLTL formulas — containing the usual logical connectives, past-time temporal operators, and propositional variables from the input alphabet — then are required to be satisfied by each of the positive traces and falsified by each of the negative traces. In machine learning terms, our goal is to learn classifiers for the input traces. However, in contrast to statistical learning approaches, our setting requires an exact classifier for the sample traces, that is, one that rejects no positive traces and accepts no negative ones [1].

The synthesis of PLTL formulas from finite samples has a variety of applications, including security policy mining from logs [2,3], debugging or understanding the behavior of a system [4], and identifying the root cause of a protocol’s misbehavior [5,6]. The PLTL fragment we consider here represents safety properties amenable to efficient runtime verification [7–11]. This fragment or its variants have been used to represent security, privacy, and safety properties of a system which can then be efficiently enforced through runtime monitoring [10–14].

We use PLTL formula synthesis to learn attack signatures for cellular networks such as 3G, 4G LTE, and 5G from a set of benign (i.e., positive) and attack (i.e., negative) traces. The cellular network attacks we consider here are possible due to the protocol state machine’s inability to handle particular out-of-order protocol packets injected over-the-air by an adversary [5,15–19]. Such attack signatures can be characterized by PLTL formulas when considering the relative ordering of packets and their payloads received/sent by the cellular device. One can envision a protocol monitor installed on a mobile device which captures messages from the cellular modem with the goal of detecting particular attack signatures and notifies the user when such attacks are detected. To our knowledge, there exist no attack notification mechanisms of this kind currently. Efficiently solving the PLTL formula synthesis problem is the first technical step towards building such mechanisms.

Prior work. The prior work most relevant to ours is the one described by Neider and Gavran [4]. They present two methods for synthesizing propositional, future-only linear temporal logic (LTL) formulas given an alphabet and a sample of infinite traces. The first method formulates the LTL formula synthesis problem as a Boolean satisfiability problem and then uses an off-the-shelf SAT solver to solve that problem. Because the SAT-based approach does not scale, the authors then develop a second method based on decision tree learning where the SAT-based method is used as an oracle to generate predicates for the decision tree. More recently, Rienert [20] improves on Neider and Gavran’s SAT-based method by precomputing models for shape constraints required by the original method. The approaches following in these works are not directly applicable to attack signature generation due to one or more of the following reasons: (1) they consider samples with infinite traces only; (2) they synthesize LTL formulas containing only future temporal operators, which are not necessarily monitorable at runtime; (3) they impose certain shape restrictions on the synthesized formula which lead to lengthy formulas.

Exploring possible approaches. Since the prior methods above [4,20] are not directly applicable to our problem domain, we started by first adapting them to the synthesis of PLTL formulas from finite traces. In our evaluation, we observe that these approaches either do not scale or do not yield succinct formulas. We then tried to reduce the synthesis problem to a Satisfiability Modulo Theory (SMT) problem where the PLTL syntax is encoded as an algebraic data-type (ADT) and the formula to synthesize is represented by a free variable $f$ with that type. We encoded the requirements of acceptance of the positive traces and rejection of the negative traces as constraints on $f$ and used an SMT solver with finite model finding capabilities [21,22] to obtain models of the ADT problem. Such models assign to $f$ a datatype value representing a candidate solution to the synthesis problem. Unfortunately, this SMT-based approach is not scalable either, which prompted us to consider an encoding of our synthesis problem as a Syntax-Guided Synthesis (SyGuS) problem [23] over ADTs. Similarly to previous approach, however, the SyGuS approach proved to be not scalable. The main reason in both cases seems to be that ADTs are user-defined and hence are do not benefit from the sort of specialized optimizations that SMT solvers employ for other builtin theories.

Our approach. This brings us to our final approach in which we encode the problem as a SyGuS problem with fixed-size bitvectors and use a specific SyGuS engine [24] to solve the problem. In our encoding, we view the projection of a trace with
respect to a propositional variable as a fixed-size bit-vector and then lift the semantics of logical and past temporal connectives to operate over bit-vectors. Such an encoding has the following advantages: (1) since fixed-size bit-vectors are natively supported by the SyGuS solver, we benefit from the solver’s various optimization techniques (e.g., rewrite rules) for them; (2) restrictions on the shape of the formula to be learned can be readily added as syntactic constraints on the SyGuS problem; (3) semantics constraints capturing the formula’s consistency with sample traces can be efficiently evaluated through direct bit-vector operations on whole traces, unlike prior approaches which operate on each individual point on a trace; (4) with an appropriate term enumeration strategy within the SyGuS solver, it is possible to obtain candidate formula of minimal size together with other candidates; (5) thanks to the SyGuS solver’s symmetry breaking criteria (i.e., agreement over the sample traces), our approach can enumerate different shapes of formulas while maintaining scalability.

Implementation and evaluation. We have implemented our approach in a novel tool called SYSLITE which uses the CVC4SY SyGuS engine [24]. We also adapted for our setting and implemented the prior methods [4, 20] mentioned earlier and considered them as baselines in our experiments. We evaluated the various approaches based on their scalability and ability to synthesize succinct PLTL formulas. To verify the generality of our SyGuS approach, in a first case study, we collected a number of PLTL formulas from the literature and considered them as target formulas to be learned. For each seed formula, we generated random traces and classified them as positive or negative based on whether they satisfied or falsified the formula. We then fed a subset of these classified random traces to both SYSLITE and our implementation of the baseline approaches, and compared the synthesized formulas with the corresponding target formulas. We observed that SYSLITE enjoys an average 60x speedup over the baseline while synthesizing a formula logically equivalent to the seed formula in most cases.

In a second case study, we used real-world cellular network traces for 11 known attacks [5, 15–19] and compared SYSLITE with the baseline. We observed that on-average SYSLITE can learn the attack signatures 28x times faster than the baseline while still being able to generate succinct attack signatures.

Contributions. To summarize, this paper makes the following technical contributions:

1. We studied a number of possible approaches for PLTL formula learning from samples, including extensions of prior SAT-based approaches originally applied to learning LTL formulas with future operators only. Our empirical evaluations of the approaches demonstrate that none of these explored approaches scale to realistic trace lengths and numbers of input traces.
2. We propose a new, more scalable learning approach which formulates the learning problem as a SyGuS problem and relies on a high-performance SyGuS engine to generate candidate solutions. Our encoding uses the theory of fixed-size bit-vectors which is natively supported by the underlying SyGuS solver, enabling us to benefit from several specific optimizations.
3. Our PLTL formula learning approach is implemented in a new tool, SYSLITE, which uses the CVC4SY SyGuS engine as a backend. We have empirically evaluated its efficacy on two case studies while considering previous state-of-the-art methods as baselines. The case studies show that that SYSLITE on-average enjoys a 44x speed-up over the baselines while, at the same time, being able to learn the expected behavior in almost all cases.

II. Technical Preliminaries

Many-Sorted First-Order Logic. We start off by briefly reviewing the usual notions and terminology of many-sorted first-order logic with equality (∼). We assume the usual definitions of signature, well-sorted terms, literals, and formulas [25]. A theory is a pair \( T = (\Sigma, I) \) where \( \Sigma \) is a signature and \( I \) is a non-empty class of \( \Sigma \)-interpretations, the models of \( T \), that is closed under variable reassignment and isomorphism. A \( \Sigma \)-formula \( \varphi \) is \( T \)-satisfiable (respectively, \( T \)-unsatisfiable) if it is satisfied by some (resp., no) interpretation in \( I \). A satisfying interpretation for \( \varphi \) models \( \varphi \). A formula \( \varphi \) is valid in \( T \) (or, \( T \)-valid), written \( \models_T \ varphi \), if every model of \( T \) is a model of \( \varphi \).

Theory of Fixed-size bit-vectors. The theory \( T_{BV} = (\Sigma_{BV}, I_{BV}) \) of fixed-size bit-vectors as defined in the SMT-LIB 2 standard [26] consists of the class of interpretations \( I_{BV} \) and signature \( \Sigma_{BV} \), which includes a unique sort for each positive integer \( n \) (representing the bit-vector width). We assume that \( \Sigma_{BV} \) includes all bit-vector constants for each \( n \), represented here as bit-strings or, to simplify the notation, by the corresponding natural number \( \{0, \ldots, 2^n-1\} \). We write a \( \Sigma_{BV} \)-term (or, bit-vector term) \( t \) of width \( n \) as \( t[n] \) when we want to specify its bit-width explicitly. We refer to the \( i \)-th bit of \( t[n] \) as \( t[i] \) with \( 0 \leq i < n \). We consider \( t[0] \) as the least significant bit (LSB), and \( t[n-1] \) as the most significant bit (MSB) of \( t \), and denote the subvector of \( t \) from index \( j \) down to \( i \) as \( t[j \ldots i] \), as well as the following bitwise operators: logical negation (¬), conjunction (\&), disjunction (\lor), and shift to the left (\ll). We will use the following arithmetic bit-vector operators: addition (+), arithmetic negation (~), and unsigned shift to the left (\ll).

SyGuS Problem. A SyGuS problem for a function \( f \) in a theory \( T \) consists of (1) semantic restrictions, or a specification, given by a (second-order) \( T \)-formula of the form \( \exists f, \varphi \), and (2) syntactic restrictions on the definitions for \( f \), given by a context-free grammar \( R \). A solution for \( f \) is a lambda term \( \lambda x . e \) of the same type as \( f \), such that (i) \( \varphi [ I \mapsto \lambda x . e ] \) is \( T \)-valid (modulo beta-reductions) and (ii) \( e \) is in the language generated by \( R \).

Past-Time Propositional Linear Temporal Logic (PLTL). The formulas we learn are of the form \( \Lambda \varphi \), where \( \varphi \) is a PLTL formula and \( \Lambda \) is a future temporal operator over finite traces (discussed below).

Definition 1 (Syntax). We use meta-variables \( \Phi \) and \( \Psi \) to denote well-formed PLTL formulas, which are defined as follows:

\[
\Phi, \Psi ::= T \mid \bot \mid p \mid \sigma^1 \Phi \mid \Phi \sigma^2 \Psi
\]

where \( p \) belongs to a non-empty set, or alphabet, \( A \) of propositional variables. The language also has unary operators \( \sigma^1 \in \{\neg, \ominus, \odot, \square\} \) and binary operators \( \sigma^2 \in \{\land, \lor, \lnot \} \). A core formula is a formula that does not contain the operators \( \lor, \land, \ominus \). The size of a formula \( \Phi \), denoted with \( |\Phi| \), is the number of its proper subformulas.

Informally, \( T \) and \( \bot \) are the universally true and the universally false formulas, respectively, and \( \land, \lor, \lnot \) are the
usual Booleans operators. On the other hand, $\circ, \Diamond, \Box$, and $S$ are past temporal operators, respectively read as “yesterday”, “once”, “historically”, and “since. Unary operators have a higher precedence than binary operators, and temporal operators have a higher precedence than logical operators.

We fix an alphabet $A$ for the PLTL formulas we consider in the rest of the paper. The standard PLTL semantics is defined over infinite traces in a Kripke structure [27]. For our purposes, however, it is more useful to define a semantics of PLTL over finite traces. A finite trace $\sigma$ (of length $n \in \mathbb{N}$ over $A$) is a sequence $(\sigma_0, \ldots, \sigma_{n-1})$ of states where a state is a total mapping from $A$ to the set $\{t, \top\}$ of Boolean values. Let $\sigma = (\sigma_0, \ldots, \sigma_{n-1})$ be a trace of length $n$. For a propositional variable $p \in A$ and we denote by $\sigma(p)$ the projection of $\sigma$ over $p$, that is, the sequence of Boolean values $(\sigma_0(p), \ldots, \sigma_{n-1}(p))$.

**Definition 2 (Semantics).** The semantics of PLTL is provided by a ternary satisfaction relation $\models$ defined inductively over core PLTL formulas as follows for all finite traces $\sigma = (\sigma_0, \ldots, \sigma_{n-1})$ and positions $i \in [0, n-1]$. 

- $\sigma, i \models \top$
- $\sigma, i \models p$ if $\sigma_i(p) = \top$
- $\sigma, i \models \neg \Phi$ if $(\sigma, i) \not\models \Phi$
- $\sigma, i \models \Phi \land \Psi$ if $(\sigma, i) \models \Phi$ and $(\sigma, i) \models \Psi$
- $\sigma, i \models \bigcirc \Phi$ if $i > 0$ and $(\sigma, i-1) \models \Phi$
- $\sigma, i \models \Phi \land \Psi$ if there is an $j \in [0, i]$ such that $(\sigma, j) \models \Psi$ and $(\sigma, k) \models \Phi$ for all $k \in [j+1, i]$.

This semantics is extended to the full language of PLTL by treating the additional operators as syntactic sugar according to the following equivalences: $\perp \equiv \neg \top; \Phi \lor \Psi \equiv \neg (\neg \Phi \land \neg \Psi); \Diamond \Phi \equiv \top S \Phi; \Box \Phi \equiv \neg \Diamond \neg \Phi$. We write $\sigma \models \Phi$ as a shorthand for $\sigma, 0 \models \Phi$. Finally, we write $\sigma \models \square \Phi$ to indicate that $\sigma, i \models \Phi$ for all $i \in [0, n-1]$ where $n$ is the length of $\sigma$.

**III. PROBLEM DEFINITION AND POSSIBLE APPROACHES**

In this section, we formalize the problem of PLTL formula synthesis from finite samples and discuss potential but inefficient approaches for solving it. We start by introducing the auxiliary notion of consistency used in our problem definition.

**Definition 3 (Consistency).** A PLTL formula $\Phi$ is consistent with a finite sample $D = (\mathcal{P}, \mathcal{N})$ of positive finite traces $\mathcal{P}$ and negative finite traces $\mathcal{N}$ with $\mathcal{P} \cap \mathcal{N} = \emptyset$ if and only if the following two conditions hold.

1. $\sigma^+ \models \square \Phi$ for all traces $\sigma^+ \in \mathcal{P}$.
2. $\sigma^- \not\models \square \Phi$ for all traces $\sigma^- \in \mathcal{N}$.

A formula $\Phi$ consistent with $D$ is minimal if no PLTL formula $\Psi$ with $|\Psi| < |\Phi|$ is consistent with $D$.

**Problem Definition 1 (PLTL Formula Synthesis from Finite Samples).** The PLTL formula synthesis problem for a given sample $D = (\mathcal{P}, \mathcal{N})$ is the problem of finding one or more PLTL formulas $\Phi$ that are consistent with $D$.

**A. Possible Approaches**

We considered several natural approaches to the PLTL synthesis problem. Unfortunately, our experimental evaluation revealed that they do not scale well. It is, however, valuable to discuss them here because their weaknesses point to potential performance bottlenecks which any synthesis algorithm must overcome to be effective in practice. We describe a better approach in Section IV.

**SAT-based Approaches.** We adapted our context prior SAT-based approaches for learning LTL formulas from samples containing only infinite traces [4, 20]. These approaches look for formulas of increasing size, measured as the depth of the formula’s abstract syntax tree (AST) which, in essence, guarantees the identification of minimal formulas consistent with a given sample $D$. As in the approach by Neider and Gavran [4], for a given depth $d$, the PLTL formula synthesis problem can be posed as the problem of checking the satisfiability of a formula $\gamma_d$ of propositional logic. The reduction is meant to be such that, $\gamma_d$ is satisfiable exactly when the original synthesis problem is solvable. Moreover, it is possible to construct a PLTL solving the synthesis problem from any propositional model of $\gamma_d$. The formula $\gamma_d$ has the form $\gamma_{d_{\text{syn}}} \land \gamma_{d_{\text{sem}}}$ where $\gamma_{d_{\text{syn}}}$ tries to capture syntactic restrictions on the expected solution (a well-formed PLTL formula with depth $d$) whereas $\gamma_{d_{\text{sem}}}$ captures the semantic restriction that the extracted solution is consistent with the sample. In turn, $\gamma_{d_{\text{syn}}}$ has the form $\gamma_{d_{\text{shape}}} \land \gamma_{d_{\text{label}}}$ where models of $\gamma_{d_{\text{shape}}}$ determine possible AST shapes of depth $d$ (including some infeasible ones) and models of $\gamma_{d_{\text{label}}}$ assign labels (i.e., propositions, logical or temporal operators) to the AST nodes.

To identify different feasible formulas, this SAT-based approach can be executed in enumerative mode by blocking a returned model of $\gamma_d$ and reissuing a call to the SAT solver with $\gamma_d$ as well as the blocking formula. Similarly to the original work, this approach does not scale to realistically sized traces or large numbers of them, as we discuss in our evaluation section.

Recently, Riener [20] improved on Neider and Gavran’s method by precomputing the models of the formula $\gamma_{d_{\text{shape}}}$ for a given depth $d$ and then supplying them with the rest of the formulas in $\gamma_d$. Unlike the approach by Neider and Gavran, models of $\gamma_{d_{\text{shape}}}$ are well-formed AST shapes. Thus, models of $\gamma_{d_{\text{syn}}}$ are indeed well-formed PLTL formulas. Riener achieves these stronger syntactic restrictions using an underlying representation based on chains instead of directed acyclic graphs as in Neider and Gavran. This approach essentially trades-off input size for execution time. We adapted this approach to our context but observed that scalability issues persist, especially, when the alphabet size is larger than 3.

Finally, we also considered a second approach by Neider and Gavran [4] which combines a classical decision tree learning algorithm with their SAT-based approach. In a first phase of this approach, the SAT-based algorithm is executed over $k$ positive and $k$ negative traces to obtain a candidate formula. The approach keeps choosing randomly from $2k$ traces until all the example traces can be separated or a timeout is reached. At that point, it invokes the decision tree learning algorithm which essentially uses the candidate formulas generated in the first phase as possible predicates for the decision tree. Because the decision tree learning algorithm combines these predicates into if-then-
else clauses, it only applies to logical languages that are closed under negation. Unfortunately, the presence of the outermost ⊤ operator in our PLTL fragment of interest, makes this fragment not closed under negation and hence this second approach is not applicable to our case.

**SMT-based Approach.** One of the scalability challenges of SAT-based algorithms can be attributed to the inefficient enumeration of the well-formed PLTL formulas. This is particularly apparent in the approach of Riener [20] who attempts to address this challenge through precomputation. A natural potential solution is to move to an SMT-based approach where the formula to be synthesized is a value of an algebraic data type (ADT) \( \Delta \) that captures the abstract syntax of well-formed PLTL formulas directly. Each PLTL propositional constant and (logical and temporal) operator is modeled by a corresponding constructor of \( \Delta \) with the same arity. Traces can be encoded as (partially defined) Boolean maps from propositional constants and trace positions. The PLTL semantics is captured by an *evaluation function*, a recursively defined total function that takes a trace \( t \) and a data type \( d \) as input and returns true if and only if \( t \) satisfies the formula represented by \( d \). The synthesis problem then reduces to adding constraints on a fresh constant \( \varphi \) of type \( \Delta \), standing for the formulas to be synthesized, stating that evaluation of \( \varphi \) is true for all the positive traces and false for all the negative ones. Syntespising the PLTL formula thus reduces to asking the SMT solver to find a model of the problem. If it succeeds, the ADT value assigned to \( \varphi \) describes a possible solution. In our evaluation, we observed that such an approach is unfortunately not scalable, possibly due to the inherent complexity of solving SMT problems over ADTs.

**SyGuS-based Approach.** We explored next a SyGuS-based approach where the PLTL syntax is encoded as a context-free grammar whereas the consistency with the sample set is given as the specification. Although more scalable than the SMT-based one, this approach is still not sufficiently scalable for our case studies. An analysis of our SyGuS encoding revealed the following two weaknesses whose mitigation led us to our final approach, discussed in the next section. First, since algebraic data types are user-defined, reasoning about them does not benefit from the specialized optimizations (e.g., rewrite rules, symmetry breaking) available to SMT solvers for other built-in theories such as bit-vectors or linear integer arithmetic. Second, both this and the SMT-based approach require evaluating a candidate solution at each position of each trace in order to guarantee consistency with the sample. Expressing such a constraint requires the use of quantified formulas (with quantification over traces and positions) and recursive function definitions (for the evaluation function) both of which are expensive to reason about.

**B. Lessons learned**

After analyzing the different approaches above to the PLTL synthesis problem, we identified the following performance bottlenecks, which we tried to address in our final approach. First, the SAT-based approaches do not have an efficient way of considering *only well-formed PLTL formulas*, a substantial bottleneck. Second, except for the SyGuS-based approach, none of the aforementioned ones apply any form of symmetry breaking optimizations to rule out or reduce the generation of formulas similar to previously generated ones, substantially hampering the generation of diverse PLTL formulas consistent with the input sample. Finally, all the approaches attempt to achieve sample-consistency through (quantified or explicit) constraints on *individual* trace positions, thus missing out on whole-trace-level optimizations, which are crucial to scalability. (Examples of our SMT-based and SyGuS-based encodings can be found in Appendix A.)

**IV. PLTL SYNTHESIS WITH SYGUSS**

In this section, we present an efficient approach for synthesizing a PLTL formula consistent with a finite sample \( D \) using a SyGuS solver over the theory of fixed-sized bit-vectors. The approach relies on the observation that a PLTL formula over finite traces of length at most \( n \) can be encoded as a function over bit-vectors of size \( n \). Thus, the problem of synthesizing a PLTL formula is reduced to the synthesis of a bit-vector function.

Similarly to a bit-vector encoding presented by Baresi et al. [28], we use bit-vectors of size \( n > 0 \) to represent the truth values of PLTL formulae at positions \([0, n − 1]\) of a given trace of length \( n \). More precisely, for each atomic proposition \( p \in \mathcal{A} \), we use a bit-vector variable \( \overline{p} \) such that \( \overline{p}[i] \) captures the value of proposition \( p \) at all instants \( i \) from 0 to \( n − 1 \). The bit-vector representation of \( \top \) for length \( n \), denoted with \( \bot[n] \), is the bit-vector constant 0 of size \( n \), while the bit-vector representation of \( \bot \), denoted with \( \top[n] \), is the value of \( \sim \bot[n] \). For any other PLTL formula \( \Phi \), we describe the value of \( \Phi \) at positions 0 through \( n − 1 \) in a trace by the bit-vector obtained by recursively performing operations on the bit-vectors corresponding to the sub-formulas of \( \Phi \). The operations performed depend on the structure of \( \Phi \) and follow the transformations shown in Table I.

<table>
<thead>
<tr>
<th>( \Phi )</th>
<th>( \overline{\Phi} )</th>
<th>unfolded bit-vector encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sim \Psi )</td>
<td>( \sim \overline{\Psi} )</td>
<td>( \sim \overline{\Psi} )</td>
</tr>
<tr>
<td>( \Psi_1 \wedge \Psi_2 )</td>
<td>( \overline{\Psi}_1 \land \overline{\Psi}_2 )</td>
<td>( \overline{\Psi}_1 \land \overline{\Psi}_2 )</td>
</tr>
<tr>
<td>( \Psi_1 \lor \Psi_2 )</td>
<td>( \overline{\Psi}_1 \lor \overline{\Psi}_2 )</td>
<td>( \overline{\Psi}_1 \lor \overline{\Psi}_2 )</td>
</tr>
<tr>
<td>( \Diamond \Psi )</td>
<td>( \lor \overline{\Psi} )</td>
<td>( \lor \overline{\Psi} )</td>
</tr>
<tr>
<td>( \Box \Psi )</td>
<td>( \land \overline{\Psi} )</td>
<td>( \land \overline{\Psi} )</td>
</tr>
<tr>
<td>( \Psi_1 S \Psi_2 )</td>
<td>( \overline{\Psi}_1 \overline{\Psi}_2 \lor (\overline{\Psi}_1 \overline{\Psi}_2 \land \overline{\Psi}_1 \overline{\Psi}_2) )</td>
<td>( \overline{\Psi}_1 \overline{\Psi}_2 \lor (\overline{\Psi}_1 \overline{\Psi}_2 \land \overline{\Psi}_1 \overline{\Psi}_2) )</td>
</tr>
</tbody>
</table>

Table I also introduces new bit-vector operators, \( \Diamond, \lozenge, \Box, \) and \( S \) to denote, respectively, the bit-vector encodings for the temporal operators \( \Diamond, \lozenge, \Box, \) and \( S \). To establish the correctness of the connection between the bit-vector encoding and the semantics of PLTL (see Theorem 1) and also for explaining the example we first introduce the following notation: for a propositional variable \( p \in \mathcal{A} \) and a trace \( \sigma \) of length \( n \), \( \sigma(p) \) denotes the bit-vector representation of \( \sigma(p) \), that is, for all \( i \in [0, n − 1] \), \( \sigma(p)[i] = 1 \) if \( \sigma_i(p) = 1 \), and \( \sigma(p)[i] = 0 \) if \( \sigma_i(p) \) is false.

To see more concretely how the translation works we explain, for instance, the correspondence between the unary PLTL operator \( \Diamond \) (read: true at least once in the present or past) and its bit-vector counterpart \( \Diamond \) with an example.

**Example 1.** Let \( p \) be a propositional variable and let \( \sigma \) be a
trace of length 6 with \( p \) is true only at positions 3 and 4 of \( \sigma \). The projection \( \sigma(p) \) is represented by the bit vector 011000 with the most significant (i.e., leftmost) bit corresponding to \( \sigma_6(p) \), the next most significant bit corresponding to \( \sigma_5(p) \) and so on. So \( \sigma(p) = 011000 \). Intuitively, the valuation of \( \diamond p \) over \( \sigma \) then be represented by the bit-vector 111000. To verify that let \( \overrightarrow{p} \) be the bit-vector variable corresponding to \( p \). According to our translation, \( \overrightarrow{\diamond p} = \overrightarrow{(\overrightarrow{p})} = \overrightarrow{\neg p} \mid \neg \overrightarrow{\overrightarrow{p}} = -\overrightarrow{\overrightarrow{p}} \mid \overrightarrow{\overrightarrow{p}} \) where | is bitwise disjunction and \( \neg \) is arithmetic negation (two’s complement). If we evaluate the resulting bit-vector formula with the valuation \( \alpha = \{ \overrightarrow{p} \mapsto 011000 \} \) we have

\[
\alpha(-\overrightarrow{\overrightarrow{p}} \mid \overrightarrow{\overrightarrow{p}}) = \begin{cases} 011000 & | -\overrightarrow{\overrightarrow{p}} \mid \overrightarrow{\overrightarrow{p}} = 011000 \\ 101000 & | \overrightarrow{\overrightarrow{p}} = 101000 \end{cases} = 111000
\]
as expected. \( \square \)

**Theorem 1.** Let \( \Phi \) be a PLTL formula over the alphabet \( A = \{ p_1, \ldots, p_m \} \) and let \( \sigma \) be a trace of length \( n \) over \( A \). Then,

\[
\sigma \models \square \Phi \iff \models_{\text{TV}} \overrightarrow{\Phi} \{ \bar{p} \mapsto \sigma \} \simeq \overrightarrow{\tau} \]  

where \( \bar{p} = (\overrightarrow{p_1}, \ldots, \overrightarrow{p_m}) \) and \( \sigma = (\overrightarrow{\sigma(p_1)}, \ldots, \overrightarrow{\sigma(p_n)}) \).

**Proof.** By induction on the structure of \( \Phi \). (A more detailed proof can be found in Appendix B).

We now show how we use the bit-vector encoding above to reduce the problem of synthesizing a PLTL formula consistent with a sample into a SyGuS problem over bit-vectors. More precisely, given propositional variables \( p_i \in A \), with \( 1 \leq i \leq m \), and a sample \( D = (\mathcal{P}, \mathcal{N}) \) whose longest trace has length \( n \), we propose to synthesize a bit-vector function \( f(\overrightarrow{p_1}, \ldots, \overrightarrow{p_m}) \) such that if \( \lambda \overrightarrow{p_1}, \ldots, \lambda \overrightarrow{p_m} \in e \) is a solution for the SyGuS problem, then there exists a PLTL formula \( \Phi \) consistent with \( D \) whose bit-vector encoding is \( e \) (that is, \( \overrightarrow{\Phi} = e \)).

To meet the requirements on \( f \), we impose the following syntactic and semantic restrictions. The former are given by the following context-free grammar:

\[
\Psi := \overrightarrow{\tau} \mid \overrightarrow{\jmath} \mid \overrightarrow{\neg p} \mid \overrightarrow{p} \mid \Phi \mid \Phi \overrightarrow{\cdot} \Psi \mid \Psi \overrightarrow{\cdot} \Psi
\]

where \( \overrightarrow{p} \) is \( \overrightarrow{p_j} \) for some \( j \in [0, m] \), \( \overrightarrow{\cdot} \in \{ \overrightarrow{\circ}, \overrightarrow{\neg}, \overrightarrow{\neg} \} \) are the unary operators, and \( \overrightarrow{\cdot} \in \{ \&\cdot, \mid \} \) are the binary operators. Notice that, although \( \overrightarrow{\circ}, \overrightarrow{\neg} \), and \( \overrightarrow{\neg} \) do not belong to the theory of bit-vectors, they can be defined using a bit-vector function in the SyGuS problem (see Table 1).

In addition, the function \( f \) of subject to the following semantic restrictions where \( | \sigma | \) denotes the length of trace \( \sigma \):

1. \[ \sum_{\sigma \in \mathcal{P}} | f(\sigma(p_1), \ldots, \sigma(p_n)) | | \sigma | - 1 : 0 | \simeq | \overrightarrow{\tau} | | \sigma | - 1 : 0 | \]
2. \[ \sum_{\sigma \in \mathcal{N}} | f(\sigma(p_1), \ldots, \sigma(p_n)) | | \sigma | - 1 : 0 | \not\simeq | \overrightarrow{\tau} | | \sigma | - 1 : 0 | \]

The two constraints enforce the consistency of the solution respectively with the positive traces and the negative traces. Notice that, since an input may include traces of different length, we compare only the relevant positions for each trace.

V. IMPLEMENTATION AND EVALUATION OF SYSLITE

In this section, we discuss the implementation of SYSLITE and our empirical evaluation of it based on two case studies.

A. SYSLITE Implementation

SYSLITE is a wrapper around the syntax-guided synthesis solver (SyGuS) CVC4SY which is part of the SMT solver CVC4 [29] and now incorporates additional optimizations for PLTL synthesis. CVC4SY supports various theories, including that of fixed-size bit-vectors, used in our encoding, and implements several specialized synthesis algorithms for various types of synthesis conjectures [30]. We rely on its support for enumerative counterexample-guided inductive synthesis (CEGIS) which was recently improved with several novel strategies [31].

In enumerative CEGIS [32], candidate solutions are generated based on some ordering, typically on term size. In our setting, a candidate solution is a function whose definition involves the bit-vector symbols from Section IV. CVC4SY uses advanced techniques to aggressively reduce the number of candidate solutions it generates. In particular, it uses fast incomplete techniques based on term rewriting to avoid generating candidate solutions \( s' \) that can be shown to be logically equivalent to some previous candidate \( s \). This technique, which is a form of symmetry breaking, is critical for the scalability of enumerative approaches [30]. Our encoding of PLTL formulas as bit-vector constraints was motivated by the intent to capitalize on CVC4SY’s existing infrastructure for establishing the equivalence of bit-vector terms, in particular, dedicated rewriting techniques developed to accelerate SyGuS enumeration [33].

For synthesis conjectures (i.e., semantic restrictions) \( \exists f. \varphi \) where all applications of \( f \) in \( \varphi \) have concrete values as arguments, CVC4SY can apply a stronger version of symmetry breaking that considers equivalence under examples. Let the concrete inputs for \( f \) in \( \varphi \) be \( c_1, \ldots, c_n \). Using this technique, the solver discards from consideration while constructing a new candidate solution for \( f \) any term \( t' \) that over the inputs \( c_1, \ldots, c_n \) evaluates exactly as some previously discarded term \( t \). For example, the terms \( x \& y \) and \( x \) take the same value over the inputs \( (0001, 0001), (0000, 0001), (1010, 1110) \) for \( (x \cdot y) \). Hence, one of them (\( x \& y \), due to its larger size) will be excluded from consideration in candidate solutions since it is equivalent to \( x \) for all relevant inputs as specified in the conjecture. In practice, this heuristics is traditionally applied when the synthesis conjecture specifies a set of input/output pairs for the function \( f \) to synthesize (with constraints of the form \( f(c_i) = o_i \)). We have generalized symmetry breaking in CVC4SY to apply the heuristics to any conjecture \( \exists f. \varphi \) where \( f \) is applied to concrete inputs, even when \( \varphi \) is not just a conjunction of input/output constraints. In our specific context, this enables symmetry breaking constraints for the negative traces, and also allows us to have traces of different length in the same problem.

Since evaluation of terms on concrete examples is a major bottleneck in syntax-guided synthesis solvers, we have additionally implemented in CVC4SY several low-level optimizations for quickly computing the result of PLTL terms on concrete inputs. Thanks to our encoding of PLTL formulas as bit-vector constraints, we can capitalize on the data structures in the core
of CVC4 for representing and efficiently evaluating bit-vectors terms. Our experiments confirm that this is critical to achieving scalability for synthesis tasks we considered.

B. Empirical Analysis Criteria and Configuration

Research questions. In our evaluation of SYSLITE, we aimed to answer the following research questions.

$RQ_1$. How effective is SYSLITE compared to a baseline in synthesizing succinct, diverse, and accurate PLTL formulas?

$RQ_2$. How scalable is SYSLITE compared to the baseline?

Case studies. We address the above questions in the context of the two case studies presented in Sections V-C and V-D, respectively. The first focuses on $RQ_1$, whereas the second focuses on $RQ_2$ based on SYSLITE’s ability to synthesize attack signatures from real cellular network traces.

Baseline. In our evaluation, we compare SYSLITE against a baseline represented by our own implementation of the (first) SAT-based method by Neider and Gavran [4]. We use our own implementation and not theirs because the latter applies to traditional LTL, as opposed to PLTL. We do not discuss here the other approaches we tried, that is, Reiner’s SAT-based approach [20] and our encodings to algebraic data types, since they proved either not scalable or ineffective.

Sample sizes. For both of our case studies, we considered the following sample sizes, each with the same number of positive and negative traces: 50, 100, 250, 500, and 1250. For Case Study I, traces were generated randomly and have length 10 whereas for Case Study II the traces were collected from a cellular network and have length 100.

Training and testing configuration. We used the standard Pareto-principle of classifier evaluation which suggests an (80%–20%) partition of the provided sample into training and testing datasets, respectively. By considering a synthesized PLTL formula $\Phi$ as a classifier for the traces in the testing set, its quality can be measured in terms of precision (the percentage of correctly classified traces among all traces classified as positive by $\Phi$), recall (the ratio of correctly classified positive traces over the total number of positive traces) and their harmonic mean (F1 score). Moreover, the evaluation method also performs cross-validation. It considers the first five solutions generated by SYSLITE and by the baseline, selecting the formula (or formulas, in case of ties) with the highest F1 score.

Evaluation infrastructure. We performed all our evaluations on a 3.40GHz Intel(R) Xeon(R) E3-1240 CPU running CentOS (Linux Kernel 3.10.0-1062.9.1) on 16GB RAM. For each instance of our experiment, we set 3600 seconds as the timeout.

C. Case Study I: PLTL Formulas from Literature

In this case study, our goal was to measure SYSLITE’s effectiveness in synthesizing succinct and accurate formulas given a sample set of traces. For this, we first collected a few representative PLTL formulas from the literature (see Table II). For each of these seed formulas, we generated a sample consisting of randomly generated traces and then checked if SYSLITE and the baseline were able to learn the original formula or an equivalent one. We had both synthesis approaches generate up to 5 candidate formulas before a given timeout.

<table>
<thead>
<tr>
<th>Literature Formula</th>
<th>PLTL Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chinese Wall Policy [10]</td>
<td>$\ell_1$($\overline{\text{access}_n,\overline{\text{user}_n,\overline{\text{threshold}_n}}}$) ∧ ($\overline{\text{access}_n,\overline{\text{user}_n,\overline{\text{threshold}_n}}}$)</td>
</tr>
<tr>
<td>Bank Transaction Policy [10]</td>
<td>$\ell_1$($\overline{\text{access}_n,\overline{\text{user}_n,\overline{\text{threshold}_n}}}$) ∧ ($\overline{\text{access}_n,\overline{\text{user}_n,\overline{\text{threshold}_n}}}$)</td>
</tr>
<tr>
<td>Secure File [10]</td>
<td>$\ell_1$($\overline{\text{secure}_n,\overline{\text{file}_n,\overline{\text{open}_n}}}$) ∧ ($\overline{\text{secure}_n,\overline{\text{file}_n,\overline{\text{open}_n}}}$)</td>
</tr>
<tr>
<td>Financial Institute Policy [10]</td>
<td>$\ell_1$($\overline{\text{institution}_n,\overline{\text{disclose}_n,\overline{\text{customer}_n,\overline{\text{years}_n}}}}$) ∧ ($\overline{\text{institution}_n,\overline{\text{disclose}_n,\overline{\text{customer}_n,\overline{\text{years}_n}}}}$)</td>
</tr>
<tr>
<td>GLBA-4602 [11, 14]</td>
<td>$\ell_1$($\overline{\text{institution}_n,\overline{\text{disclose}_n,\overline{\text{customer}_n,\overline{\text{years}_n}}}}$) ∧ ($\overline{\text{institution}_n,\overline{\text{disclose}_n,\overline{\text{customer}_n,\overline{\text{years}_n}}}}$)</td>
</tr>
<tr>
<td>HIPPA-164508A2 [11, 14]</td>
<td>$\ell_1$($\overline{\text{covered}_n,\overline{\text{entity}_n,\overline{\text{patient}_n,\overline{\text{anonymize}_n}}}}$) ∧ ($\overline{\text{covered}_n,\overline{\text{entity}_n,\overline{\text{patient}_n,\overline{\text{anonymize}_n}}}}$)</td>
</tr>
<tr>
<td>HIPPA-164508A3 [11, 14]</td>
<td>$\ell_1$($\overline{\text{covered}_n,\overline{\text{entity}_n,\overline{\text{patient}_n,\overline{\text{anonymize}_n}}}}$) ∧ ($\overline{\text{covered}_n,\overline{\text{entity}_n,\overline{\text{patient}_n,\overline{\text{anonymize}_n}}}}$)</td>
</tr>
<tr>
<td>Dynamic Separ. of Duty [10]</td>
<td>$\ell_1$($\overline{\text{member}_n,\overline{\text{activates}_n,\overline{\text{role}_n,\overline{\text{status}_n,\overline{\text{period}_n}}}}}$) ∧ ($\overline{\text{member}_n,\overline{\text{activates}_n,\overline{\text{role}_n,\overline{\text{status}_n,\overline{\text{period}_n}}}}}$)</td>
</tr>
</tbody>
</table>

Trace generation: Given a seed formula $\varphi$ from Table II, a desired trace length $\ell$, and a desired sample size $2n$, our trace generation process uses a cryptographically-secure pseudorandom number generator to produce a sample set $\mathcal{P}$ of $n$ positive traces and a sample set $\mathcal{N}$ of $n$ negative traces, all of length $\ell$. It generates a trace $\sigma$ of length $\ell$ by randomly assigning truth values to $\varphi$’s propositional variables for each of the $\ell$ states of $\sigma$. The trace goes in the set $\mathcal{P}$ or $\mathcal{N}$ depending on whether it satisfies $\varphi$ or not, as long as the set in question contains less than $n$ traces; otherwise, it is discarded. Note that, depending on the formula $\varphi$, we may have to oversample for positive or negative traces.

Measuring quality of synthesized formulas. To evaluate the quality of the synthesized formulas, in addition to rely on the usual statistical measures (i.e., precision, recall, and F1 score) on the test dataset, we considered logical equivalence with the seed formula (i.e., being satisfied by exactly the same set of possible traces) as another metric of effectiveness. We used the GOAL tool [34] to check for logical equivalence in PLTL.

<table>
<thead>
<tr>
<th>Seed Formula</th>
<th>SYSLITE</th>
<th>SAT</th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chinese Wall Policy [10]</td>
<td>5/5</td>
<td>5/5</td>
<td>5/5</td>
</tr>
<tr>
<td>Bank Transaction Policy [10]</td>
<td>5/5</td>
<td>5/5</td>
<td>5/5</td>
</tr>
<tr>
<td>Secure File [10]</td>
<td>5/5</td>
<td>5/5</td>
<td>5/5</td>
</tr>
<tr>
<td>Financial Institute Policy [10]</td>
<td>5/5</td>
<td>5/5</td>
<td>5/5</td>
</tr>
<tr>
<td>GLBA-4602 [11, 14]</td>
<td>5/5</td>
<td>5/5</td>
<td>5/5</td>
</tr>
<tr>
<td>HIPPA-164508A2 [11, 14]</td>
<td>5/5</td>
<td>5/5</td>
<td>5/5</td>
</tr>
<tr>
<td>HIPPA-164508A3 [11, 14]</td>
<td>5/5</td>
<td>5/5</td>
<td>5/5</td>
</tr>
<tr>
<td>Dynamic Separ. of Duty [10]</td>
<td>2/5</td>
<td>2/5</td>
<td>2/5</td>
</tr>
<tr>
<td>Total</td>
<td>37/40 (92.5%)</td>
<td>31/40 (77.5%)</td>
<td>18/40 (45%)</td>
</tr>
</tbody>
</table>

Quality of synthesized formulas. Our results on the synthesized formulas’ quality (i.e., equivalence to seed formula) and count are summarized in Table III. For each run of SYSLITE and the baseline for a particular dataset and a seed formula, we select the highest-ranked formula after cross validation among those synthesized in the allotted time, if any. For each original (seed) formula, column Count reports the total of number selected formulas across the 5 training sets of different size. For instance, a value of 2/5 indicates that the algorithm was able to synthesize formulas for 2 of the 5 training sets. Column Quality reports how many of the selected formulas are logically equivalent to the seed formula.

Our evaluation confirms that SYSLITE can learn the seed formula or an equivalent one for each of the five random sample sets in almost all cases. The only exceptions are the Dynamic

---

6 In this case study, we did not observe any ties after cross-validation.
Separation of Duty formula, for which SYSLITE generates two formulas neither of which is equivalent to the seed formula, and the Chinese Wall Policy formula, for which it generates one formula and only for the sample set of size 1250. To put things in perspective, however, note that since the Chinese Wall Policy formula has two variables and traces have length 10, a set of 1250 traces covers just 0.1% of the set of all possible $4^{10}$ traces. Remarkably, SYSLITE is able to learn the right formula with much smaller sample sets in all the other cases, with perfect precision, recall, and F1 scores.

Looking at the baseline approach, it performs gracefully with a few simple seed formulas such as Bank Transaction Policy and HIPAA-164508A3. However, it cannot synthesize any candidates for the Secure File seed formula. Moreover, its synthesized formulas for HIPPA-164508A2, Dynamic Separation of Duty, and Chinese Wall Policy are not equivalent to the seed. Detailed results are available in Appendix D-C.

**Scalability.** The training results for case study I are shown in Figure 1. The X-axis of the graph represents the different training set sizes: 80% of 50, 100, 250, 500, and 1250, while the Y-axis (in log-scale) represents the training time in seconds. Cross validation times are not shown because they are uniform and negligible. The horizontal red line on the top of the graph represents the timeout (3600 seconds). In the graph, we only show results for the 3 seed-formulas for which the SAT-method performs best. Complete results are presented in Appendix D.

In our evaluation, SYSLITE was able to generate results for almost all combinations of seed formula and training set size while exhibiting an average 60x speedup over the baseline. The exception, already mentioned, is the Dynamic Separation of Duty formula where it timed-out on the training sets with more than 100 traces. This is likely due to the large size of the formulas to be synthesized which requires SYSLITE to enumerate internally a very large number of terms. The baseline method was unable to generate any formula and timed-out, even for the smallest sample (of 50 traces) for the Secure File formula. For a few of the other seed formulas, it failed to synthesize a candidate even for the small sample sets (of size 50 and 100). For example, in HIPPA-164508A2 policy it failed to synthesize any formula for sample size larger than 50 traces; for the Dynamic Separation of Duty and Financial Institute it was unable to deal with sample sets with more than 100 traces. These scalability problems are the main cause of its low formula-quality scores (shown in Table III) and low statistical measures scores (not shown).

**D. Case Study II: 4G LTE Attack Signature Generation**

Our second case study focused on synthesizing attack signatures, represented as PLTL formulas, for cellular networks from a set of benign (i.e., positive) and attack (i.e., negative) traces. Once again, we considered the scalability and effectiveness of SYSLITE versus the SAT-based baseline. The choice of this application domain was motivated by the vital role cellular networks play in a modern nation’s infrastructure, which makes them a frequent target for malicious attacks [5, 15–17, 35, 36].

As with any protocol, the cellular network protocol allows only specific orderings of messages (packets) sent or received by a cellular device, and predicates over their payload (e.g., the sequence number is in a range). For a given type of attack, the synthesized attack signature is expected to be satisfied, ideally, by all and only the benign protocol executions, those not containing an attack. This way, one can deploy a runtime monitor [37] for each attack type that checks whether the current execution violates (i.e., falsifies) the attack signature and issues an alert as soon as it detects a violation. Currently, there are no mechanisms that can achieve this goal efficiently. Being able to automatically synthesize effective attack signatures is the natural first step towards that.

In light of this, our case study focused on 11 known, representative attacks that are detectable from the vantage point of a cellular device (see Table IV). These attacks target weaknesses of the cellular network protocol in the Non-Access Stra-
tum (NAS) layer, responsible for communication between a cellular device and the core network, and the Radio Resource Control (RRC) layer, responsible for the communication between a device and the base station [5, 15–19]. While other attacks exist [6, 15–17, 38–45], they are not detectable from a device’s point of view and thus are not considered in our case study.

Trace gathering. We now discuss how we gathered benign traces and generated attack traces through testbed experiments.

**Benign Traces:** We collected benign traces through random sampling of traces from a crowd-sourced platform to which users all over the world submit their cellular network traces through an Android app called MobileInsight [46]. Our collected traces include 1892 NAS layer traces containing about 52K messages and 2045 RRC layer traces containing about 1.5M messages. We cleaned up the traces so that each contained 100 states as this is sufficient for the attacks we considered.

**Malicious Traces:** To collect malicious traces, we first implemented each attack and its variants using srsLTE [47] and software-defined radios in a testbed. srsLTE is an open-source cellular network stack which permits the modification of different components of the network. Then, to collect the attack traces we used SCAT [48], a desktop application capable of extracting 4G LTE modem traffic exposed by certain devices through a USB interface. Finally, we inserted one or more copies of the malicious traces at arbitrary positions of some arbitrarily chosen benign traces to obtain our set of malicious traces. Such an approach mimics a real-world scenario in which attacks occur while synthesizing multi-session attacks from RRC traffic. In case of the Measurement Report attack, the baseline timed out for all sample sizes and did not yield any signature. In contrast, and as illustrated in Figure 2, we observed that the SYSLITE is scalable and efficient in synthesizing multi-session attacks signatures exhibiting on average a 28x speedup over the baseline. We stress that scalability is essential in this context to promptly generate attack signatures for newly discovered attacks before attackers can cause substantial damage.

**Fig. 2.** Training Results of Case Study II.

**Quality of the synthesized attack signatures.** In this case study, our quality criteria are signature succinctness and correctness in capturing the attack. We consider an attack signature to be succinct if it can concisely capture the attack’s root cause without including any superfluous events (e.g., messages received/sent) or conditions (e.g., predicates over message payload). Visual inspection of the signatures returned by SYSLITE and the baseline shows that those generated by SYSLITE are more succinct. Those returned by SYSLITE are shown in Table IV.

Looking at correctness, our evaluation shows that all the attack signatures synthesized by either the SAT-based baseline or SYSLITE for the NAS layer have a perfect (100%) precision, recall and F1 on the testing set. However, the baseline is able to synthesize signature only with samples of small size. For the RRC layer attacks, SYSLITE is able to score perfectly on the test dataset based on statistical measures (i.e., precision, recall, F1). The baseline, however, does not achieve a 100% precision, recall, and F1 score as it cannot synthesize any signature for the Measurement Report attack. We have also manually vetted the correctness of the synthesized attack signatures by both SYSLITE and the baseline based on our domain expertise on cellular security and observed that the signatures, when generated, correctly identified (i.e., rejected) traces in which the attacks occur.

**Scalability.** The scalability results for Case Study II are shown in Figure 2. The graph’s X-axis shows the sizes of the different training sets we used whereas the Y-axis (in log-scale) reports the corresponding training time in seconds. Timeout time is shown as a red horizontal line. For ease of exposition, we show only the training results for 3 NAS and 2 RRC layers attacks. For the rest of attacks, the results follow a similar trend. Complete results are presented in Appendix D.

We conjecture that the performance of the baseline is comparable with that of SYSLITE when learning attack signatures on the NAS protocol layer because it induces attacks spanning only a single protocol session. Thus, the patterns are relatively easier to learn. On the other hand, for the RRC layer attacks, the sequences of attack steps can be complex and the attack may span over multiple sessions thus making it challenging to learn (see Appendix D). Indeed, the baseline timed out more frequently while synthesizing multi-session attacks from RRC traffic. In case of the Measurement Report attack, the baseline timed out for all sample sizes and did not yield any signature. In contrast, and as illustrated in Figure 2, we observed that the SYSLITE is scalable and efficient in synthesizing multi-session attacks signatures exhibiting on average a 28x speedup over the baseline. We stress that scalability is essential in this context to promptly generate attack signatures for newly discovered attacks before attackers can cause substantial damage.

**VI. Conclusion**

We have presented an efficient approach for synthesizing PLTL formulas from a set of finite traces. The approach reduces the problem to a bit-vector function synthesis problem and then uses an enhanced version of the CVC4SyGuS solver to solve the latter. The reduction to bit-vector function synthesis proves critical for performance not only because CVC4SyS implements specific optimization for bit-vectors but also because it allows us to express efficiently the requirements capturing the consistency of the solution with the samples. The conventional wisdom that SyGuS solvers are more efficient for problems over natively supported theories compared to reductions to other SMT theories (such as algebraic datatypes) or to SAT is corroborated by our experimental evaluation.

Possible directions for future work include understanding the impact of grammar representation (i.e., which temporal opera-


**APPENDIX**

I. ENCODING OF SMT-BASED AND SYGU$S$-BASED APPROACHES

In this section, we present all proposed the encodings for PLTL synthesis that were explained earlier in the section III and IV. We have instantiated these encodings using traces shown in Table V. These traces were generated using the following seed formula.

Seed Formula. The seed formula describes a common situation in an emergency alert system (EAS). "Every time if there is a failure, then since last failure there has been some alarm previously or there was no failure at all".

\[ \square(failure \Rightarrow \neg (\diamond \neg \text{alarm} \land S \text{failure})) \]

Generated Traces. Using the above seed formula and cryptographically-secure pseudo-random number generator (CSPRNG), we generate a sample dataset containing 3 positive and 6 negative traces.

SYSLITE \(^3\) can be used to synthesize the seed formula from these randomly generated input traces.

A. SMT-based Encoding using SMT-LIB 2.

For detailed discussion please refer to section III.

\[ \text{(set-logic ALL)} \]
\[ \text{;; Options} \]
\[ \text{(set-option :fmf-bound true)} \]

\(^3\)SYSLITE is available on https://github.com/CLC-UIowa/SySLite/

| TABLE V. Generated sample set containing positive traces [1-3] and negative traces [4-9]. |
|---------------------------------|---------------------------------|
| failure | alarm |
| \( \sigma_1 \sigma_2 \sigma_3 \sigma_4 \) | \( \sigma_1 \sigma_2 \sigma_3 \sigma_4 \) |
| 1. | 10000 |
| 2. | 10010 |
| 3. | 01001 |
| 4. | 11 |
| 5. | 101 |
| 6. | 10111 |
| 7. | 01011 |
| 8. | 00111 |
| 9. | 00011 |

(set-option :fmf-fun true)
(set-option :produce-models true)

;; Declare Variable Sort
(declare-sort VarId () Int)

;; Declare Unary Operators.
(declare-datatype UNARY_OP ( NOT Y G H O ))

;; Declare Binary Operators
(declare-datatype BINARY_OP ( AND OR IMPLIES ))

;; Syntax of PLTL
(declare-datatype Formula ( Top Bottom (P (Id VarId)) (Op1 (op1 UNARY_OP) (f Formula)) (Op2 (op2 BINARY_OP) (f1 Formula) (f2 Formula)) )

) ;; Trace Constructs
(define-sort Trace () Int)
(define-sort Time () Int)

;; Length of Positive [1-3] and Negative Traces [4-5] 
from Table V
(define-fun len ( (ttr Trace) ) Int )
(ite (tt 1) 4 (ite (tt 2) 4 (ite (tt 3) 4 (ite (tt 4) 1 (ite (tt 5) 2 (ite (tt 6) 4 (ite (tt 7) 4 (ite (tt 8) 4 (ite (tt 9) 4 0) )))))) )

;; Positive [1-3] and Negative Traces [4-9]
instantiation using values from Table V
(define-fun val ( (tt (tt r) ) (t z) (x VarId)) Bool)
(ite (tt 1) (t 4) (ite (tt 0) (t 0) (x 0) ))
(ite (tt 1) (t 4) (ite (tt 0) (t 0) (x 0) ))
(ite (tt 1) (t 4) (ite (tt 0) (t 0) (x 0) ))
(ite (tt 1) (t 4) (ite (tt 0) (t 0) (x 0) ))
(ite (tt 1) (t 4) (ite (tt 0) (t 0) (x 0) ))
(ite (tt 1) (t 4) (ite (tt 0) (t 0) (x 0) ))
(ite (tt 1) (t 4) (ite (tt 0) (t 0) (x 0) ))
(ite (tt 1) (t 4) (ite (tt 0) (t 0) (x 0) ))
(and (= tr 3) (= t 2) (= x 1))
(and (= tr 3) (= t 4) (= x 0))
(and (= tr 4) (= t 4) (= x 1))
(and (= tr 4) (= t 1) (= x 0))
(and (= tr 5) (= t 0) (= x 0))
(and (= tr 5) (= t 2) (= x 0))
(and (= tr 6) (= t 0) (= x 0))
(and (= tr 6) (= t 0) (= x 1))
(and (= tr 6) (= t 2) (= x 0))
(and (= tr 6) (= t 4) (= x 0))
(and (= tr 7) (= t 0) (= x 1))
(and (= tr 7) (= t 1) (= x 0))
(and (= tr 7) (= t 1) (= x 1))
(and (= tr 7) (= t 2) (= x 1))
(and (= tr 7) (= t 3) (= x 0))
(and (= tr 7) (= t 4) (= x 0))
(and (= tr 8) (= t 0) (= x 0))
(and (= tr 8) (= t 2) (= x 0))
(and (= tr 8) (= t 3) (= x 0))
(and (= tr 8) (= t 4) (= x 0))
(and (= tr 9) (= t 0) (= x 1))
(and (= tr 9) (= t 2) (= x 1))
(and (= tr 9) (= t 3) (= x 0))
(and (= tr 9) (= t 3) (= x 1))
(and (= tr 9) (= t 4) (= x 0))
(and (= tr 9) (= t 4) (= x 1))
)

;; Semantics of PLTL
(define-fun-rec holds ((f Formula) (tr Trace) (t Time)) Bool
(let ((tn (len tr)))
(and (<= 0 t tn)
(match f (
(Top true)
(Bottom false)
((P i) (val tr t i))
((Op1 op g) (match op (NOT (not (holds g tr t)))))
(H (and (holds g tr t) (or (= t 0) (holds f tr (- t 1))))))
(O (or (holds g tr t) (and (< 0 t) (holds f tr (- t 1))))))
(G (and (holds g tr t) (or (= t tn) (holds f tr (+ t 1))))))))
)

;; phi is the formula to be synthesized
(declare-const phi Formula)

;; checking all positive traces
(define-fun-rec holds-for-all-traces ((tr Trace) (f Formula)) Bool
(or (< tr 1)
(and (holds (Op1 G f) tr 0)
(holds-for-all-traces (- tr 1) f))
)
)

;; Positive Traces endpoint
(define-const pos_tr Int 3)

;; Constraint phi w.r.t. positive traces
(assert (holds-for-all-traces pos_tr phi))

;; checking all negative traces
(define-fun-rec fail-for-all-traces ((tr Trace) (f Formula)) Bool
(or (<= tr pos_tr)
(and (not (holds (Op1 G f) tr 0))
(fail-for-all-traces (- tr 1) f))
)
)

;; Constraint phi w.r.t. negative Traces
(assert (fail-for-all-traces 9 phi))

(check-sat)
(get-value (phi))

;Example Formula:
(Infix) failure => !(Y(!(alarm) S failure))
(Prefix) =>(failure, !(Y(S(!(alarm), failure)))

B. SyGuS-ADT Encoding using SMT-LIB 2 & ADT Grammar.

For detailed discussion please refer to section III.

(set-logic ALL)

;;Options
(set-option :sygus-out status-and-def)
(set-option :sygus-rec-fun true)
(set-option :e-matching false)

;; Declare Variable Sort
(define-sort VarId () Int)

;; Declare Unary Operators.
(declare-datatype UNARY_OP (NOT Y G H O))

;; Declare Binary Operators.
(declare-datatype BINARY_OP (AND OR IMPLIES S))

;; Syntax of the Formula
(declare-datatype Formula (Top) (Bottom) (P (Id VarId)) (Op1 (op1 UNARY_OP) (f Formula)) (Op2 (op2 BINARY_OP) (f1 Formula) (f2 Formula)))

)
; Context-free Grammar
(synth-fun phi () Formula
  ((<F> Formula) (<I> Int) (<O1> UNARY_OP) (<O2> BINARY_OP)
    ((<F> Formula (Top Bottom))
      (Op1 <I> <F>)
      (Op2 <O2> <F> <F>)
    )
  )
)

; Trace Constructs
(define-sort Trace () Int)
(define-sort Time () Int)
;; Length of Positive [1-3] and Negative Traces [4-5] from Table V
(define-fun len ((tr Trace)) Int
  (ite (= tr 1) 4
       (ite (= tr 2) 4
            (ite (= tr 3) 4
                 (ite (= tr 4) 4
                      (ite (= tr 5) 4
                           (ite (= tr 6) 4
                                (ite (= tr 7) 4
                                     (ite (= tr 8) 4
                                          (ite (= tr 9) 4 0))))))))))

; Positive [1-3] and Negative Traces [4-9] instantiation using values from Table V
(define-fun val ((tr Trace) (t Time) (x VarId)) Bool
  (or
    (and (= tr 1) (= t 0) (= x 0))
    (and (= tr 1) (= t 0) (= x 1))
    (and (= tr 1) (= t 1) (= x 1))
    (and (= tr 2) (= t 0) (= x 0))
    (and (= tr 2) (= t 0) (= x 1))
    (and (= tr 2) (= t 1) (= x 1))
    (and (= tr 2) (= t 3) (= x 0))
    (and (= tr 2) (= t 3) (= x 1))
    (and (= tr 2) (= t 4) (= x 1))
    (and (= tr 3) (= t 0) (= x 0))
    (and (= tr 3) (= t 0) (= x 1))
    (and (= tr 4) (= t 0) (= x 0))
    (and (= tr 4) (= t 0) (= x 1))
    (and (= tr 5) (= t 0) (= x 0))
    (and (= tr 5) (= t 0) (= x 1))
    (and (= tr 5) (= t 2) (= x 0))
    (and (= tr 5) (= t 2) (= x 1))
    (and (= tr 6) (= t 0) (= x 0))
    (and (= tr 6) (= t 0) (= x 1))
    (and (= tr 6) (= t 2) (= x 0))
    (and (= tr 6) (= t 2) (= x 1))
    (and (= tr 7) (= t 0) (= x 0))
    (and (= tr 7) (= t 0) (= x 1))
    (and (= tr 7) (= t 1) (= x 1))
    (and (= tr 7) (= t 2) (= x 1))
    (and (= tr 7) (= t 3) (= x 0))
    (and (= tr 7) (= t 3) (= x 1))
    (and (= tr 7) (= t 4) (= x 0))
    (and (= tr 7) (= t 4) (= x 1))
    (and (= tr 7) (= t 6) (= x 0))
    (and (= tr 7) (= t 6) (= x 1))
    (and (= tr 8) (= t 0) (= x 0))
    (and (= tr 8) (= t 0) (= x 1))
    (and (= tr 8) (= t 2) (= x 0))
    (and (= tr 8) (= t 2) (= x 1))
    (and (= tr 8) (= t 3) (= x 0))
    (and (= tr 8) (= t 3) (= x 1))
    (and (= tr 8) (= t 4) (= x 0))
    (and (= tr 8) (= t 4) (= x 1))
    (and (= tr 9) (= t 0) (= x 0))
    (and (= tr 9) (= t 0) (= x 1))
    (and (= tr 9) (= t 2) (= x 0))
    (and (= tr 9) (= t 2) (= x 1))
    (and (= tr 9) (= t 3) (= x 0))
    (and (= tr 9) (= t 3) (= x 1))
    (and (= tr 9) (= t 4) (= x 0))
    (and (= tr 9) (= t 4) (= x 1))
  )))

; ; checking all positive traces
(define-fun-rec holds ((f Formula) (tr Trace) (t Time)) Bool
  (let ((tn (len tr)))
    (match f (
      (Top true)
      (Bottom false)
      (Y (and (< 0 t) (holds g tr (- t 1))))
      (H (and (holds g tr t) (or (= t 0) (holds f tr (- t 1))))))
      (O (or (holds g tr t) (and (< 0 t) (holds f tr (- t 1))))))
      (G (and (holds g tr t) (or (= t tn) (holds f tr (- t l))))))
    )))

; ; positive traces endpoint
(define-const pos_tr Int 3)
;; Constraint phi w.r.t. positive traces
(constraint (holds-for-all-traces pos_tr phi))

; ; checking all negative traces
(define-fun-rec fail-for-all-traces ((tr Trace) (f Formula)) Bool
  (match op (NOT (not (holds g tr t)))
    (AND (and (holds f1 tr t) (holds g tr t)))
    (OR (or (holds f1 tr t) (holds g tr t)))
    (IMPLIES (or (not (holds f1 tr t))) (holds g tr t)))
    (S (or (holds g tr t) (and (holds f1 tr t) (and (< 0 t) (holds f tr (- t 1))))))
  )))

; ; checking all positive traces
(define-fun-rec holds ((f Formula) (tr Trace) (t Time)) Bool
  (let ((tn (len tr)))
    (match f (
      (Top true)
      (Bottom false)
      (Y (and (< 0 t) (holds g tr (- t 1))))
      (H (and (holds g tr t) (or (= t 0) (holds f tr (- t 1))))))
      (O (or (holds g tr t) (and (< 0 t) (holds f tr (- t 1))))))
      (G (and (holds g tr t) (or (= t tn) (holds f tr (- t l))))))
    )))

; ; positive traces endpoint
(define-const pos_tr Int 3)
;; Constraint phi w.r.t. positive traces
(constraint (holds-for-all-traces pos_tr phi))

; ; checking all negative traces
(define-fun-rec fail-for-all-traces ((tr Trace) (f Formula)) Bool
  (match op (NOT (not (holds g tr t)))
    (AND (and (holds f1 tr t) (holds g tr t)))
    (OR (or (holds f1 tr t) (holds g tr t)))
    (IMPLIES (or (not (holds f1 tr t))) (holds g tr t)))
    (S (or (holds g tr t) (and (holds f1 tr t) (and (< 0 t) (holds f tr (- t 1))))))
  )))

; ; positive traces endpoint
(define-const pos_tr Int 3)
;; Constraint phi w.r.t. positive traces
(constraint (holds-for-all-traces pos_tr phi))

; ; checking all negative traces
(define-fun-rec fail-for-all-traces ((tr Trace) (f Formula)) Bool
  (match op (NOT (not (holds g tr t)))
    (AND (and (holds f1 tr t) (holds g tr t)))
    (OR (or (holds f1 tr t) (holds g tr t)))
    (IMPLIES (or (not (holds f1 tr t))) (holds g tr t)))
    (S (or (holds g tr t) (and (holds f1 tr t) (and (< 0 t) (holds f tr (- t 1))))))
  )))
C. SyGuS-BV Encoding using SMT-LIB 2 & BitVector Grammar.

For detailed discussion please refer to section III.

(set-logic BV)
(set-option :sygus-out status-and-def)
(set-option :e-matching false)
(set-option :check-synth)

;; Constraint phi w.r.t. negative traces
(constraint (fail-for-all-traces 9 phi))

;; Example Formula:
;; (check-synth)
;; ((Prefix) => (failure, !(Y(S(not(alarm), failure))))
;; ((Infix) failure => !(Y(not(alarm) S failure)))

;; Sequence these traces is reversed
(constraint (fail-for-all-traces 9 phi))
(constraint (fail-for-all-traces 9 phi))
(constraint (fail-for-all-traces 9 phi))
(constraint (fail-for-all-traces 9 phi))

;; Positive examples [1-3] from Table V
;; Sequence these traces is reversed
(constraint (fail-for-all-traces 9 phi))
(constraint (fail-for-all-traces 9 phi))
(constraint (fail-for-all-traces 9 phi))
(constraint (fail-for-all-traces 9 phi))

;; Negative examples [4-9] from Table V
;; Sequence these traces is reversed
(constraint (fail-for-all-traces 9 phi))
(constraint (fail-for-all-traces 9 phi))
(constraint (fail-for-all-traces 9 phi))
(constraint (fail-for-all-traces 9 phi))

II. Proof of Correctness for Bit-vector Based Encoding of PLTL Semantics

Theorem 1. Let $\Phi$ be a PLTL formula over the alphabet $A = \{p_1, \ldots, p_m\}$ and let $\sigma$ be a trace of length $n$ over $A$. Then,

$$\sigma \models \Box_I \Phi \text{ iff } \models_{\text{BV}} \psi \{ \bar{p} \mapsto \bar{\sigma} \} \simeq \bar{\sigma}[n]$$

where $\bar{p} = (\bar{p}_1[n], \ldots, \bar{p}_m[n])$ and $\bar{\sigma} = (\bar{\sigma}(p_1), \ldots, \bar{\sigma}(p_m))$.

Proof. The proof follows by induction in the structure of $\Phi$. For the cases in which $\Phi$ is a propositional variable $p_i \in A$, $\top$, or $\bot$, and for the cases in which $\Phi$ is of the form $\neg \Psi$, $\Psi_1 \land \Psi_2$, $\Psi_1 \lor \Psi_2$, $\Psi_1 \Leftrightarrow \Psi_2$, $\Psi_1 \rightarrow \Psi_2$, we refer the reader to the proofs by Baresi et al. [28]. We only show here the proofs for the cases in which we provide a different bit-vector encoding, that is, when $\Phi$ is of the form $\Diamond \Psi$ or $\Box \Psi$.

- Case $\Phi = \Diamond \Psi$. From the semantics of PLTL and the definition of $\Box_I$, we know that $\sigma \models \Box_I \Psi$ if and only if for all $i \in \{0, \ldots, n-1\}$, there is a $j \in \{0, i\}$ such that $(\sigma, j) \models \Psi$. So, by the previous definition and the induction hypothesis, $\Diamond \Psi$ should be the bit-vector with zeros at all positions (i.e., $\psi = [0, n]$) or a bit-vector with zeros at positions $[0, k-1]$ and ones at positions $[k, n-1]$, where $k$ is the index of the first least
significant non-zero bit of $\Psi$. Now we will prove that this is the case indeed. If $\Psi = \bot_{\{\sigma\}}$, then $-\Psi | \Psi \sim \sum_{\sigma} \sim \bot_{\{\sigma\}}$, as we wanted to prove. Otherwise, $-\Psi[i] = \sim \Psi[i]$ for all $i \in [k + 1, n - 1]$, $-\Psi[k] = 1$, and $-\Psi[i] = 0$ for all $i \in [0, k - 1]$. Thus, $(-\Psi | \Psi)[i] = 1$ for all $i \in [k + 1, n - 1]$. $(-\Psi | \Psi)[k] = 1$, and $(-\Psi[i] | \Psi)[i] = 0$ for all $i \in [0, k - 1]$.

• Case $\Phi = \Box \Psi$. From the semantics of PLTL and the definition of $\Box_j$, we have that $\sigma \models \Box \Psi$ if and only if for all $i \in [0, n - 1]$ and for all $j \in \{0, \ldots, \sigma, j\} = \Psi$. So, by the previous definition and the induction hypothesis, $\sim \Psi$ should be the bit-vector with ones at all positions (i.e., $\bot_{\{\sigma\}}$) or a bit-vector with ones at positions $[k, k - 1]$ and zeros at positions $[k, n - 1]$, where $k$ is the index of the first least significant zero bit of $\Psi$. Now we will prove that this is the case indeed. If $\Psi = \bot_{\{\sigma\}}$, then $\sim(1 + \bot_{\{\sigma\}}) \& \bot_{\{\sigma\}} \sim \bot_{\{\sigma\}}$, as we wanted to show. Otherwise, $\sim(1 + \Psi)[i] = \sim \Psi[i]$ for all $i \in [k + 1, n - 1]$, $\sim(1 + \Psi)[k] = 0$, and $\sim(1 + \Psi)[i] = 1$ for all $i \in [0, k - 1]$. Therefore, $\sim(1 + \Psi) \& \Psi)[i] = 0$ for all $i \in [k + 1, n - 1]$, $\sim(1 + \Psi) \& \Psi)[k] = 0$, and $\sim(1 + \Psi) \& \Psi)[i] = 1$ for all $i \in [0, k - 1]$.

\[\Phi \equiv \neg \Psi \lor \bigoplus (\neg \Psi \cdot \Psi)\]

### III. Example

In Table VI, we show evaluation of an example formula over a simple trace $\sigma$ using bit-vector encoding described in A-C.

**TABLE VI.** $\Phi = \neg \Psi \lor \bigoplus (\neg \Psi \cdot \Psi)$

<table>
<thead>
<tr>
<th>$\sigma$</th>
<th>$\Phi$</th>
<th>$\Psi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bot_{{\sigma}}$</td>
<td>$\bot_{{\sigma}}$</td>
<td>$\bot_{{\sigma}}$</td>
</tr>
<tr>
<td>$\neg \Psi_{{\sigma}}$</td>
<td>$\neg \Psi_{{\sigma}}$</td>
<td>$\bot_{{\sigma}}$</td>
</tr>
<tr>
<td>$\neg \Psi_{{\sigma}} \lor \Psi_{{\sigma}}$</td>
<td>$\bot_{{\sigma}}$</td>
<td>$\bot_{{\sigma}}$</td>
</tr>
<tr>
<td>$\neg \Psi_{{\sigma}} \lor \bigoplus (\neg \Psi_{{\sigma}} \cdot \Psi_{{\sigma}})$</td>
<td>$\bot_{{\sigma}}$</td>
<td>$\bot_{{\sigma}}$</td>
</tr>
</tbody>
</table>

1. Formula $\Phi = \neg \Psi \lor \bigoplus (\neg \Psi \cdot \Psi)$ holds at any time point if both $\neg \Psi$ is true (i.e. $\Psi = \bot_{\{\sigma\}}$) or $\bigoplus (\neg \Psi \cdot \Psi)$ holds.
2. $\Psi_{\{\sigma\}}$ (i.e. $\bigoplus (\neg \Psi \cdot \Psi)$) is true if the yesterday of $\neg \Psi$ is true that carry over the truth value from the previous time point to the current time point (i.e. $\neg \Psi_{\{\sigma\}}$). It is to note that $\sigma_0$ is set to '0' because yesterday of the initial point is always false.
3. $\Psi_{\{\sigma\}}$ (i.e. $\neg \Psi$) is true at any time point if both $\Psi$ (i.e. $\Psi_{\{\sigma\}}$) is true OR $\Psi_0$ set state to true at any time point when both $\sim \Psi$ carried over $q$ value from previous state (i.e. $\Psi_{\{\sigma\}}$) are true because the sum is '0' in that case.
4. $\Psi_{\{\sigma\}}$ using addition carry over the truth to the left when $q$ holds, as long as $\neg \Psi$ holds captured by $\Psi_{\{\sigma\}}$.

**IV. Additional Experimental Data**

**A. Training Dataset for Case Study I**

Table VII shows the training time of SYSLITE and the baseline SAT-based method in Case Study I. We consider enumerating first five solutions returned by each synthesis algorithm and

**TABLE VII.** Table of Training Results for Case Study I.

<table>
<thead>
<tr>
<th>Sample Size</th>
<th>Formula Type</th>
<th>SYSLITE</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>Chinese Wall Policy</td>
<td>2.79</td>
<td>97.17</td>
</tr>
<tr>
<td>80</td>
<td>Chinese Wall Policy</td>
<td>1.53</td>
<td>78.96</td>
</tr>
<tr>
<td>200</td>
<td>Chinese Wall Policy</td>
<td>2.99</td>
<td>738.05</td>
</tr>
<tr>
<td>400</td>
<td>Chinese Wall Policy</td>
<td>5.39</td>
<td>1513.02</td>
</tr>
<tr>
<td>1000</td>
<td>Chinese Wall Policy</td>
<td>37.19</td>
<td>3600</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sample Size</th>
<th>Formula Type</th>
<th>SYSLITE</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>Bank Transaction Policy</td>
<td>3.5</td>
<td>44.33</td>
</tr>
<tr>
<td>80</td>
<td>Bank Transaction Policy</td>
<td>2.97</td>
<td>96.86</td>
</tr>
<tr>
<td>200</td>
<td>Bank Transaction Policy</td>
<td>5.25</td>
<td>312.91</td>
</tr>
<tr>
<td>400</td>
<td>Bank Transaction Policy</td>
<td>6.63</td>
<td>841.94</td>
</tr>
<tr>
<td>1000</td>
<td>Bank Transaction Policy</td>
<td>17.94</td>
<td>3600</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sample Size</th>
<th>Formula Type</th>
<th>SYSLITE</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>Secure File</td>
<td>870.55</td>
<td>3600</td>
</tr>
<tr>
<td>80</td>
<td>Secure File</td>
<td>513.11</td>
<td>3600</td>
</tr>
<tr>
<td>200</td>
<td>Secure File</td>
<td>161.21</td>
<td>3600</td>
</tr>
<tr>
<td>400</td>
<td>Secure File</td>
<td>230.56</td>
<td>3600</td>
</tr>
<tr>
<td>1000</td>
<td>Secure File</td>
<td>2952.12</td>
<td>3600</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sample Size</th>
<th>Formula Type</th>
<th>SYSLITE</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>Financial Institute</td>
<td>11.7</td>
<td>1020.44</td>
</tr>
<tr>
<td>80</td>
<td>Financial Institute</td>
<td>32.33</td>
<td>3257.56</td>
</tr>
<tr>
<td>200</td>
<td>Financial Institute</td>
<td>49.92</td>
<td>3600</td>
</tr>
<tr>
<td>400</td>
<td>Financial Institute</td>
<td>222.29</td>
<td>3600</td>
</tr>
<tr>
<td>1000</td>
<td>Financial Institute</td>
<td>2810.14</td>
<td>3600</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sample Size</th>
<th>Formula Type</th>
<th>SYSLITE</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>GLBA-6802</td>
<td>7.63</td>
<td>1651.74</td>
</tr>
<tr>
<td>80</td>
<td>GLBA-6802</td>
<td>82.97</td>
<td>3600</td>
</tr>
<tr>
<td>200</td>
<td>GLBA-6803</td>
<td>167.97</td>
<td>3600</td>
</tr>
<tr>
<td>400</td>
<td>GLBA-6804</td>
<td>1040.92</td>
<td>3600</td>
</tr>
<tr>
<td>1000</td>
<td>GLBA-6805</td>
<td>2637.34</td>
<td>3600</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sample Size</th>
<th>Formula Type</th>
<th>SYSLITE</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>HIPPA-164508A2</td>
<td>4.37</td>
<td>287.98</td>
</tr>
<tr>
<td>80</td>
<td>HIPPA-164508A2</td>
<td>83.39</td>
<td>3600</td>
</tr>
<tr>
<td>200</td>
<td>HIPPA-164508A3</td>
<td>175.09</td>
<td>3600</td>
</tr>
<tr>
<td>400</td>
<td>HIPPA-164508A2</td>
<td>286.8</td>
<td>3600</td>
</tr>
<tr>
<td>1000</td>
<td>HIPPA-164508A2</td>
<td>2740.32</td>
<td>3600</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sample Size</th>
<th>Formula Type</th>
<th>SYSLITE</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>HIPPA-164508A3</td>
<td>5.38</td>
<td>507.8</td>
</tr>
<tr>
<td>80</td>
<td>HIPPA-164508A3</td>
<td>3.27</td>
<td>81.37</td>
</tr>
<tr>
<td>200</td>
<td>HIPPA-164508A3</td>
<td>4.56</td>
<td>342.7</td>
</tr>
<tr>
<td>400</td>
<td>HIPPA-164508A3</td>
<td>8.29</td>
<td>823.67</td>
</tr>
<tr>
<td>1000</td>
<td>HIPPA-164508A3</td>
<td>14.44</td>
<td>3600</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sample Size</th>
<th>Formula Type</th>
<th>SYSLITE</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>Dynamic Separation of Duty</td>
<td>76.34</td>
<td>2985.08</td>
</tr>
<tr>
<td>80</td>
<td>Dynamic Separation of Duty</td>
<td>94.66</td>
<td>2184.23</td>
</tr>
<tr>
<td>200</td>
<td>Dynamic Separation of Duty</td>
<td>3600</td>
<td>3600</td>
</tr>
<tr>
<td>400</td>
<td>Dynamic Separation of Duty</td>
<td>3600</td>
<td>3600</td>
</tr>
<tr>
<td>1000</td>
<td>Dynamic Separation of Duty</td>
<td>3600</td>
<td>3600</td>
</tr>
</tbody>
</table>

**B. Training Dataset for Case Study II**

Table VII shows the training time of SYSLITE and the baseline SAT-based method in Case Study II. Here SYSLITE gains a 28x speedup over the existing state-of-the-art method.
### TABLE VIII. Table of Training Results for Case Study II.

<table>
<thead>
<tr>
<th>Sample Size</th>
<th>LTE Attack Name</th>
<th>SYSLITE</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>Numb Attack</td>
<td>384.72</td>
<td>67.62</td>
</tr>
<tr>
<td>80</td>
<td>Numb Attack</td>
<td>340.73</td>
<td>165.56</td>
</tr>
<tr>
<td>200</td>
<td>Numb Attack</td>
<td>870.78</td>
<td>493.41</td>
</tr>
<tr>
<td>400</td>
<td>Numb Attack</td>
<td>56.53 [*3]</td>
<td>1381.28</td>
</tr>
<tr>
<td>1000</td>
<td>Numb Attack</td>
<td>296.41</td>
<td>2970.24</td>
</tr>
<tr>
<td>40</td>
<td>Paging with IMSI</td>
<td>13.33</td>
<td>432.57</td>
</tr>
<tr>
<td>80</td>
<td>Paging with IMSI</td>
<td>19.69</td>
<td>770.78</td>
</tr>
<tr>
<td>200</td>
<td>Paging with IMSI</td>
<td>31.71</td>
<td>1641.48</td>
</tr>
<tr>
<td>400</td>
<td>Paging with IMSI</td>
<td>56.27</td>
<td>3600</td>
</tr>
<tr>
<td>1000</td>
<td>Paging with IMSI</td>
<td>154.36</td>
<td>3600</td>
</tr>
<tr>
<td>40</td>
<td>EMM Information</td>
<td>3.69</td>
<td>41.76</td>
</tr>
<tr>
<td>80</td>
<td>EMM Information</td>
<td>3.61</td>
<td>73.66</td>
</tr>
<tr>
<td>200</td>
<td>EMM Information</td>
<td>8.49</td>
<td>255.16</td>
</tr>
<tr>
<td>400</td>
<td>EMM Information</td>
<td>14.7</td>
<td>555.68</td>
</tr>
<tr>
<td>1000</td>
<td>EMM Information</td>
<td>35.76</td>
<td>3600</td>
</tr>
<tr>
<td>40</td>
<td>NULL Encryption</td>
<td>4.9</td>
<td>23.14</td>
</tr>
<tr>
<td>80</td>
<td>NULL Encryption</td>
<td>2.53</td>
<td>50.18</td>
</tr>
<tr>
<td>200</td>
<td>NULL Encryption</td>
<td>4.96</td>
<td>234.46</td>
</tr>
<tr>
<td>400</td>
<td>NULL Encryption</td>
<td>6.16</td>
<td>483.44</td>
</tr>
<tr>
<td>1000</td>
<td>NULL Encryption</td>
<td>14.14</td>
<td>1204.13</td>
</tr>
<tr>
<td>40</td>
<td>RLF Report</td>
<td>193.63 [*3]</td>
<td>3083</td>
</tr>
<tr>
<td>80</td>
<td>RLF Report</td>
<td>207.61 [*2]</td>
<td>3600</td>
</tr>
<tr>
<td>200</td>
<td>RLF Report</td>
<td>656.84 [*4]</td>
<td>3600</td>
</tr>
<tr>
<td>400</td>
<td>RLF Report</td>
<td>332.89 [*4]</td>
<td>3600</td>
</tr>
<tr>
<td>1000</td>
<td>RLF Report</td>
<td>1022.75 [*2]</td>
<td>3600</td>
</tr>
<tr>
<td>40</td>
<td>Malformed Identity Request</td>
<td>2.34</td>
<td>31.63</td>
</tr>
<tr>
<td>80</td>
<td>Malformed Identity Request</td>
<td>3.59</td>
<td>74.47</td>
</tr>
<tr>
<td>200</td>
<td>Malformed Identity Request</td>
<td>6.86</td>
<td>229.06</td>
</tr>
<tr>
<td>400</td>
<td>Malformed Identity Request</td>
<td>10.23</td>
<td>549.56</td>
</tr>
<tr>
<td>1000</td>
<td>Malformed Identity Request</td>
<td>23.61</td>
<td>1788.12</td>
</tr>
<tr>
<td>40</td>
<td>IMSI Cracking</td>
<td>6.29</td>
<td>48.47</td>
</tr>
<tr>
<td>80</td>
<td>IMSI Cracking</td>
<td>8.99</td>
<td>102.33</td>
</tr>
<tr>
<td>200</td>
<td>IMSI Cracking</td>
<td>21.03</td>
<td>244.42</td>
</tr>
<tr>
<td>400</td>
<td>IMSI Cracking</td>
<td>13.35</td>
<td>1064.41</td>
</tr>
<tr>
<td>1000</td>
<td>IMSI Cracking</td>
<td>97.47</td>
<td>3600</td>
</tr>
<tr>
<td>40</td>
<td>Authentication Failure</td>
<td>2.72</td>
<td>18.85</td>
</tr>
<tr>
<td>80</td>
<td>Authentication Failure</td>
<td>2.38</td>
<td>51</td>
</tr>
<tr>
<td>200</td>
<td>Authentication Failure</td>
<td>3.82</td>
<td>159.8</td>
</tr>
<tr>
<td>400</td>
<td>Authentication Failure</td>
<td>8.13</td>
<td>509.88</td>
</tr>
<tr>
<td>1000</td>
<td>Authentication Failure</td>
<td>17.31</td>
<td>1445.6</td>
</tr>
<tr>
<td>40</td>
<td>IMSI Catching</td>
<td>3.3</td>
<td>35.77</td>
</tr>
<tr>
<td>80</td>
<td>IMSI Catching</td>
<td>4.41</td>
<td>77.25</td>
</tr>
<tr>
<td>200</td>
<td>IMSI Catching</td>
<td>5.68</td>
<td>195.26</td>
</tr>
<tr>
<td>400</td>
<td>IMSI Catching</td>
<td>8.98</td>
<td>471.71</td>
</tr>
<tr>
<td>1000</td>
<td>IMSI Catching</td>
<td>22.43</td>
<td>1434.77</td>
</tr>
<tr>
<td>40</td>
<td>Measurement Report</td>
<td>805.03 [*4]</td>
<td>3600</td>
</tr>
<tr>
<td>80</td>
<td>Measurement Report</td>
<td>307.73 [*2]</td>
<td>3600</td>
</tr>
<tr>
<td>200</td>
<td>Measurement Report</td>
<td>742.41 [*2]</td>
<td>3600</td>
</tr>
<tr>
<td>400</td>
<td>Measurement Report</td>
<td>231.77 [*2]</td>
<td>3600</td>
</tr>
<tr>
<td>1000</td>
<td>Measurement Report</td>
<td>3600</td>
<td>3600</td>
</tr>
<tr>
<td>40</td>
<td>Aka Bypass</td>
<td>601.64</td>
<td>2521.27</td>
</tr>
<tr>
<td>80</td>
<td>Aka Bypass</td>
<td>224.94 [*2]</td>
<td>1841.37</td>
</tr>
<tr>
<td>200</td>
<td>Aka Bypass</td>
<td>376.90 [*2]</td>
<td>3600</td>
</tr>
<tr>
<td>400</td>
<td>Aka Bypass</td>
<td>580.47 [*2]</td>
<td>3600</td>
</tr>
<tr>
<td>1000</td>
<td>Aka Bypass</td>
<td>305.4 [*1]</td>
<td>3600</td>
</tr>
</tbody>
</table>
C. Evaluation Dataset for Case Study I

The detailed evaluation results of SYSLITE and the baseline method in Case Study I is presented in Table IX. We have used PTL to Büchi automata translation implementation in a tool called GOAL [34] for checking the equivalence between seed and the synthesized formulas.

<table>
<thead>
<tr>
<th>Literature Formula</th>
<th>Total Sample Size</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
<th>Synthesized Formulas</th>
</tr>
</thead>
<tbody>
<tr>
<td>HIPAA 164508a3</td>
<td>1250</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>GLBA 6802</td>
<td>250</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Bank Transaction Policy</td>
<td>250</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Secure File</td>
<td>1250</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Chinese Wall Policy</td>
<td>1250</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Literature Formula</td>
<td>Total Sample Size</td>
<td>Precision</td>
<td>Recall</td>
<td>F1</td>
<td>Synthesized Formulas</td>
</tr>
<tr>
<td>Secure File</td>
<td>1250</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Secure File</td>
<td>1250</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Secure File</td>
<td>1250</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Secure File</td>
<td>1250</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Secure File</td>
<td>1250</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>

D. Evaluation Dataset for Case Study II

The detailed evaluation results of SYSLITE and the baseline method in Case Study II are shown in Table X.
### Table X. Table of Evaluation Results for Case Study II.

<table>
<thead>
<tr>
<th>LTE Attacks</th>
<th>Total Sample Size</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
<th>Synthesized Formulas</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SVSYS-100%</td>
<td>SYVLYS-100%</td>
<td>SYLR-100%</td>
<td>SYL10-100%</td>
<td>SYN-100%</td>
</tr>
<tr>
<td>Nimb Attack</td>
<td>50</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Nimb Attack</td>
<td>100</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Nimb Attack</td>
<td>250</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Nimb Attack</td>
<td>500</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Nimb Attack</td>
<td>1250</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Paping with IMSI</td>
<td>50</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Paping with IMSI</td>
<td>100</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Paping with IMSI</td>
<td>250</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Paping with IMSI</td>
<td>500</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Paping with IMSI</td>
<td>1250</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>IMSI Cracking</td>
<td>50</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>IMSI Cracking</td>
<td>100</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>IMSI Cracking</td>
<td>250</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>IMSI Cracking</td>
<td>500</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>IMSI Cracking</td>
<td>1250</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>