Checking LTL[F,G,X] on Compressed Traces in Polynomial Time

Minjian Zhang
minjian2@illinois.edu
University of Illinois at Urbana-Champaign
USA

Umang Mathur
umathur3@illinois.edu
University of Illinois at Urbana-Champaign
USA

Mahesh Viswanathan
vmahesh@illinois.edu
University of Illinois at Urbana-Champaign
USA

ABSTRACT
The problem of checking if a program execution meets a formal specification arises in many software engineering tasks including runtime verification and designing test oracles. When online analysis is not possible, execution trace logs are stored for offline postmortem analysis, often in a compressed format to reduce disk space and warehousing requirements. A straightforward method for checking if a compressed execution satisfies a property is to first decompress it and then analyze the resulting uncompressed execution.

In this paper, we consider the problem of checking if an execution trace, compressed using a grammar-based lossless compression scheme, satisfies a specification expressed in linear temporal logic, without explicitly decompressing it. In general, this problem is known to be intractable (PSPACE-hard in the size of the trace and the LTL formula). We show that the problem can be solved in polynomial time for the fragment LTL[F,G,X], which comprises of all Boolean and modal operators of LTL except the until operator.

Our algorithm for analyzing SLPs (a grammar-based compression scheme) is effective in practice — for a suite of large execution traces obtained from open source projects, our algorithm shows significant speed ups when compared with the performance of checking LTL properties over corresponding uncompressed traces.

ACM Reference Format:

1 INTRODUCTION
Consider the membership problem that can be abstractly defined as follows: Given a program execution \( \tau \) determine if \( \tau \) is a "good" behavior. This computational problem plays a key role in several approaches whose goal is the engineering of reliable and secure software. The first sub-area where it arises is runtime verification [10, 27] where one dynamically monitors the behavior of a system with the goal of determining if the observed behavior conforms to system requirements. This approach can be used to augment testing by observing system behavior along paths that were inadequately exercised during testing and thereby increasing code coverage. One key computational task in runtime verification is to solve the membership problem, where the monitored execution needs to be checked against system requirements. A second sub-area where the membership problem arises is in designing test oracles [9]. Given executions of a system exercised by a test suite, a test oracle is a program that distinguishes between correct and incorrect behaviors of the system. Thus, a test oracle can be seen as solving the membership problem for the specific system being tested. The membership problem also plays a key role in intrusion detection [38]. Log files that record the interaction between a network of elements need to be examined to detect patterns of "intrusive behavior" so that corrective measures can be taken to avoid security compromises. Since these log files are large, and there are multiple patterns of intrusive behavior, intrusion is typically detected automatically by a program that solves the membership problem to determine if the log files do not contain any intrusive patterns. Finally, the membership problem also needs to be solved when statistically model checking a system. Statistical model checking [4, 39] is an approach to verify quantitative properties of stochastic systems. In this approach, the model checker executes the stochastic system a few times to draw a statistical sample of system behaviors, and then use hypothesis testing to determine the likelihood of a property being true of a system. A crucial step in this process is building an oracle that determines for each execution, whether it satisfies a desired logical property.

One practical challenge in each of these application areas that rely on the membership problem is the size of the program execution that needs to be analyzed. Program traces that arise in runtime verification, testing or statistical model checking are often huge, containing millions of events. Long traces are often necessary to exercise large parts of the code base to ensure good code coverage. Log files analyzed for intrusions often record interactions that take place over long windows of time, sometimes over multiple days. The challenge therefore is two-fold: how to store such long traces/logs, and how to effectively analyze them. The common solution to address the warehousing needs for such traces is to compress them and then store them in compressed format.

Given that program executions need to be compressed to address storage costs, the important question in the context of the membership problem is, to find effective solutions to the problem when the input trace is compressed. This question is not new, and has a rich history, especially in theoretical computer science [7, 8, 12, 14, 15, 30, 36, 41]. The short summary of results in this space is as follows. There is always a naïve algorithm to solve the membership problem on compressed traces — uncompress the trace and check membership. In many situations this is often (provably) the best algorithm possible [7, 15, 36]. However, there are exceptions where the membership problem can be solved in time that is polynomial in the size of the compressed trace [29]; one notable example is dynamic race detection on compressed strings [23].

The main question we investigate in this paper is the following: Given a program trace \( \tau \) in compressed form and a formula \( \phi \) in
linear temporal logic (LTL) [37], determine if r satisfies φ. LTL is widely used in testing and verification. It’s popularity relies on the fact that, on the one hand it is rich enough to express many requirements that typically arise in software engineering, and on the other hand, the absence of explicit quantification, makes it simple enough for a practitioner to easily write properties. Compression schemes we consider are those where a program trace is represented using a straight line program (SLP). SLPs are special context-free grammars where the language of the grammar contains exactly one string, namely, the trace it represents. Several lossless compression schemes, like run-length encoding and Lempel-Ziv encodes [44], can be efficiently converted into SLPs with similar size. There are efficient implementations of compressions algorithms that produce an SLP representation of a given execution [3, 21, 22, 25, 35, 42–44].

The problem of determining if a finite trace compressed using an SLP satisfies an LTL property, has been studied before. The problem is known to be intractable — it is PSPACE-hard [32]. Therefore, we ask if there is a rich fragment of LTL for which the problem can be efficiently solved. We consider the fragment LTL [F, G, X] which is the collection of all LTL formulas that are built from propositions using boolean operators, and only the temporal operators X (next), F (eventually or finally), and G (always or globally); in particular, U (until) cannot be used in the formulas of LTL [F, G, X]. The fragment LTL [F, G, X] is expressively very rich. Over infinite traces, LTL [F, G, X] can express properties in each class of the safety-progress classification of temporal properties introduced by Manna and Pnueli [31] 1. Our main result is that the problem of checking of a finite trace represented by an SLP satisfies an LTL [F, G, X] formula can be decided in time that is polynomial in the size of the SLP (compressed trace) and the formula.

We now outline the technical challenges and our theoretical contributions in obtaining this result. The principal idea used in verification, runtime verification, and automatic test oracle generation for temporal properties is to exploit the connection between LTL formulas and automata — translate the formula into an automaton, and then “run” the automaton with the program or trace to verify or test. For runtime verification or test oracles, the automaton constructed from the formula needs to be deterministic. This idea can also be used when checking compressed traces where you effectively “run” the deterministic automaton on the grammar representing the trace.

However, for even for LTL [F, G] formulas 2, the smallest non-deterministic automaton is exponential and the smallest deterministic automaton is doubly exponential in the size of the formula [5]. Theoretical lower bounds establish that this cannot be improved. Our first observation is that if the finite (uncompressed) trace is processed right-to-left instead of left-to-right, then there is an exponential sized, deterministic automaton for each LTL [F, G, X] formula, that can solve the membership question. “Running” an automaton left-to-right or right-to-left on an SLP is very similar and so this change does not fundamentally change the algorithm for compressed traces. However, the fact that the automaton is exponential sized would affect the complexity; for compressed traces, the running time of an algorithm using this automaton would be exponential in the formula. To combat this, we observe that the automaton we design for LTL [F, G, X] has special “monotonicity” properties and has a small “diameter”. These two observations can be combined to observe that there are “essentially” $O(m)$ state changes ($m$ here refers to the size of the LTL [F, G, X] formula) when the automaton is run on any (uncompressed) trace, no matter what the length of the trace is. Finally, we exploit the special structure of the states of this automaton, to design an algorithm for compressed traces. To prove that this algorithm indeed runs in time that is polynomial in the formula size and the grammar, requires carefully counting the number of substrings that arise in a string represented by an SLP.

We evaluate the performance of our algorithm for checking compressed traces over open source Java projects obtained from GitHub (largely derived from prior study [28]). We also use 10 LTL[F, G, X] properties describing specifications for the use of iterators, collections, file objects, etc. Our evaluation suggests that, large traces from open source projects can be effectively compressed (with an average compression ratio of 641) and that compressed traces can be effectively analyzed checked against these specifications, leading to significant speed ups (averaging at 34%).

The rest of the paper is organized as follows. Section 2 discusses background relevant for the exposition. Section 3 discusses the overview of our algorithm for checking LTL[F, G, X] formulae on compressed traces, and Sections 4 and 5 discuss the technical details of the algorithm. We present our evaluation in Section 6, related work in Section 7 and concluding remarks in Section 8.  

2 PRELIMINARIES

In this section we present preliminary notations about execution traces, LTL monitoring and the SLP compression format.  

2.1 Execution Traces

In many approaches whose goal is to either prove the correctness of a software or find errors, a key computational problem that needs to be solved is the membership problem, where one needs to determine if a given program execution is correct with respect to a system specification. In this setting, an execution trace (or simply an execution) can be abstractly modeled as a finite sequence of “events” belonging to a set (say) $\Sigma$. The set of events $\Sigma$ is determined by what is visible or has been made visible through instrumentation when the program is executed. Thus, an execution is $\tau = e_0 e_1 \cdots e_{k-1}$ where each $e_i \in \Sigma$; the empty trace/sequence will be denoted by $\epsilon$. Let us fix an execution $\tau = e_0 e_1 \cdots e_{k-1}$. The $i$th event in the execution will be denoted by $\tau[i] = e_i$. We will denote the substring $e_i e_{i+1} \cdots e_j$ by $\tau[i:j]$, the suffix $e_{i+1} e_{i+2} \cdots e_{k-1}$ by $\tau[i:]$ and the prefix $e_0 e_1 \cdots e_{i-1}$ by $\tau[:i]$. The length of execution $\tau$, denoted $|\tau|$, is the number of events in it which is $k$. By definition $|\epsilon| = 0$.

Example 1. Consider the Java class SetTravers$\text{al}$ in Figure 1. Every instance of this class has a member variable $s$, which is a set of integer elements. The method insert inserts all non-negative integers less than or equal to $s$, while the method sumAllExcept returns the sum of those elements of the set $s$ which are different from the integer $val$. We remark that the implementation of sumAllExcept is

---

1 Or in every Borel class that has $\omega$-regular properties.
2 These are LTL formulas that only have $F$ and $G$ as temporal operators; even $X$ is not used.
functionally correct whenever val is not the last value when traversing s using the iterator itr. If val is the last value in s when traversing using itr, the loop body can execute next() (after traversing the node with itr), even though there are no remaining elements, which may raise a Java exception (NoSuchElementException).

The figure also shows a test class SetTraversalTest that implements a unit test testInsertAndSum that first calls insert on an instance st of SetTraversal with the argument 128 and then checks if the sum of elements thus inserted (except the element 111) is as expected. The given unit test passes and, in fact, does not expose the bug outlined above. The execution trace generated due to this test, can nevertheless be used to infer the possibility of an exception. If we instrument calls to the methods hasNext() and next(), then we will observe a trace over the alphabet $\Sigma = \{h, n\}$, where $h$ represents a call to hasNext() and $n$ represents a call to next(). For the unit test testInsertAndSum, we will observe the execution trace $t = \langle h \rangle^{65}n\langle h \rangle^{64}h$. This is because, in this case, the iterator traverses the set s in the order of insertion, and for the first 65 elements (values 0 through 64), the method insert correctly calls hasNext() before next(). However, in the next step, it enters the loop and calls next without checking hasNext(). All the subsequent loop executions generate the sequence $\langle h \rangle^{65}n\langle h \rangle^{64}h$. In subsequent sections, we will discuss how analyzing $t$, in fact, can hint at the possibility of an exception. Observe that $|t| = 256$, $t[1 : 130] = \langle h \rangle^{65}$, $t[130 :] = n\langle h \rangle^{64}h$ and $t[2 : 130] = \langle h \rangle^{64}$.

2.2 Linear Temporal Logic

Linear temporal logic (LTL) is a popular logic for specifying temporal properties of systems, and is widely used to specify correctness properties. In this section, we introduce the syntax and semantics of LTL and some fragments that are relevant in this paper. Since program executions encountered while testing and runtime verification are assumed to be finite, our semantics for LTL will be defined for finite execution traces. While this is not the classical semantics for LTL, it is standard [13]. We will also be using a “letter semantics” for the logic — models are sequences of letters as opposed to sequences of sets of propositions, and formulas are built using letters as opposed to propositions.

Syntax of LTL. Let us fix a finite alphabet $\Sigma$. Then, a formula $\varphi$ in LTL over is given by the following grammar.

$$\varphi ::= a | \neg \varphi | \varphi \wedge \varphi | \varphi \vee \varphi | X \varphi | F \varphi | G \varphi | \varphi U \varphi$$

Here, $a$ is a symbol in $\Sigma$, $\neg$, $\wedge$ and $\vee$ are Boolean connectives and $X$ (`next’), $F$ (`eventually’), $G$ (`always’) and $U$ (`until’) are temporal modal operators. We will use $\varphi_1 \implies \varphi_2$ as a shorthand for $\neg \varphi_1 \vee \varphi_2$.

Semantics of LTL. The semantics of LTL is given by how an LTL formula $\varphi$ evaluates over a finite non-empty trace $r \in \Sigma^+$. We formally describe this evaluation relation $\models$ below; the subscript `$f$’ in $\models_f$ stands for evaluation over finite traces.

$$r \models_f a$$

$$r \models_f \neg \varphi$$

$$r \models_f \varphi_1 \wedge \varphi_2$$

$$r \models_f \varphi_1 \vee \varphi_2$$

$$r \models_f X \varphi$$

$$r \models_f F \varphi$$

$$r \models_f G \varphi$$

$$r \models_f \varphi_1 U \varphi_2$$

Example 2. Consider the program shown in Figure 1 and the execution $r$ produced when calls to methods hasNext() and next() are tracked. As shown in Example 1, $r = \langle h \rangle^{65}n\langle h \rangle^{64}h$, where $h$ represents a call to hasNext() and $n$ represents a call to next().

Intuitively, in a correct implementation, the program should check the existence of a next element (i.e., event h) before accessing the next element (i.e., event n). The program in Figure 1 does not satisfy this intuitive correctness requirement since an event h does not precede the event n when the value 65 is accessed. We can formalize our informal intuition by requiring that ‘there are no successive calls to next()’. However, this by itself is not enough because the execution $nnhn$ does not have successive n events, but the first n event is not preceded by h. So we must also require that the execution does not begin with n. We could write this as $\varphi = (\neg n) \wedge G (n \implies \neg X(n))$. One can see that $r$ does not satisfy this property (as desired) because $r[129 :] = nn\langle h \rangle^{64}h$ does not satisfy $n \implies \neg Xn$.

Fragments of LTL. We will consider a couple of fragments of LTL obtained by restricting the modal operators that appear in formulas. The first fragment is LTL[$X$] which consists of formulas built from events and Boolean operators using only X operator. The next fragment is LTL[F, G, X] which uses the modal operators X, F, and G, but does not use U. We skip the formal BNF grammar for these fragments.
Formulas in LTL[F, G, X] have a normal form that is obtained by pushing X as far as in possible. Since $X(\phi_1 \lor \phi_2) \equiv (X\phi_1) \lor (X\phi_2)$, $X(\phi_1 \land \phi_2) \equiv (X\phi_1) \land (X\phi_2)$, and $X\neg \phi \equiv \neg X\phi$, we can push X inside conjunctions, disjunctions and F operators. However, over finite executions X cannot be pushed inside ‘¬’ or ‘G’ operators. To see this, consider the execution $\sigma = h\eta$. Observe that $\sigma$ satisfies $X\Gamma n$, but not $Xn\Gamma$. Similarly, $\tau = n\eta$ satisfies $\neg Xh$ but not $X\neg h$. The normal form can be described by the following BNF grammar.

$$
\begin{align*}
\phi &::= \psi \mid \eta \mid \phi \land \psi \mid \phi \lor \psi \mid F\phi \\
\eta &::= X\psi \mid X\eta \\
\psi &::= a \lor \neg \psi \lor G\phi
\end{align*}
$$

Formulas in LTL[F, G, X] ($\phi$) are one of X-formulas ($\eta$), G-formulas ($\psi$), conjunctions/disjunctions of LTL[F, G, X] formulas, or an F operator applied to an LTL[F, G, X] formula in the normal form. X-formulas are those where the top level operator is X. A X operator can only be applied to either a G-formula or an X-formula. Finally, G-formulas are letters, or negations of LTL[F, G, X] formulas ($\phi$), or have G as the topmost operator. Every LTL[F, G, X] formula can be converted into this normal form, with at most quadratic blowup.

**Example 3.** Consider the LTL[F, G, X] formula $\phi = X(\Gamma n \lor F h)$. The normal form for this can be obtained by pushing X as far inside as possible. Thus, $\phi' = (X\Gamma n) \lor (FXh)$ is the equivalent formula in normal form.

**Automata for LTL and its fragments.** For LTL properties, algorithms for verification, runtime verification, and test oracle generation, all rely on the translation of logic formulas to automata. For a specification $\phi$, the crucial step therefore, is the construction of an automaton $A_\phi$ such that an execution $\tau$ is accepted by $A_\phi$ if and only if $\tau$ satisfies $\phi$. The size of $A_\phi$ has big influence on the complexity of the algorithm. For runtime verification and test oracle generation, the automaton $A_\phi$ needs to be deterministic. Because of the critical role translations from LTL to automata play in algorithms, these have been well studied. Unfortunately, the translation from formulas to deterministic automata can result in at least a double exponential blowup. It is worth emphasizing that the result below holds whether we interpret LTL over finite or infinite executions.

**Theorem 2.1** (Ahar-LaTorre [5]). There is a family $\{\phi_n\}_{n\in\mathbb{N}}$ of LTL[F, G, X] formulas such that the size of $\phi_n$ is $n$ and any deterministic acceptor for $\phi_n$ is of size $\Omega(2^{n^2})$.

**2.3 Compressed Executions**

In this paper, we will present algorithms to solve the membership problem when the program execution is compressed. The compressed execution we consider will be encoded by a straight line program (SLP), which is a special context-free grammar whose language has exactly one string, namely, the execution it represents. Several lossless compression schemes, like run-length encoding and Lempel-Ziv encoding [44] can be efficiently converted into SLPs of similar size. Several efficient algorithms that compress strings using SLPs are known [3, 21, 22, 25, 35, 42–44].

**Straight Line Programs (SLP).** Recall that a context-free grammar $G = (T, N, S, R)$, where $T$ is the set of terminals, $N$ is the set of non-terminals, $S \in N$ is the starting non-terminal, and $R$ is the set of rules, has size $|G|$. Based on Theorem 2.1, this is double exponential in $|\phi|$ and thus intractable. Indeed, theoretical lower bounds establish that there is no algorithm that has a tractable asymptotic complexity for full LTL.
Theorem 2.2 (Markey and Schnoebelen [32]). Given an SLP $G = (T, N, S, R)$ and an LTL formula $\varphi$, the problem of checking if $[S] \models \varphi$ is PSPACE-hard.

The main result of this paper shows that this problem can be solved efficiently (in polynomial time) when the formula is from the LTL fragment $\text{LTL}[F, G, X]$.

3 TECHNICAL OVERVIEW

Recall (from Section 2.3) that there is a simple automata-theoretic algorithm for checking if a compressed trace (SLP $G$) satisfies a given LTL formula $\varphi$ (i.e., $[G] \models \varphi$). This simple algorithm constructs a DFA $A_\varphi$ corresponding to the given LTL formula $\varphi$ (in time $O(|A_\varphi|)$), and then inductively computes, for every non-terminal $A$ of $G$ and for every state $q$ of $A_\varphi$, the next state $q' = \delta(q, [A])$ obtained after running the trace fragment $[A]$ on $A_\varphi$. As we previously observed the total running time of this simple algorithm is $O(|A_\varphi| + |G| \cdot |A_\varphi| D)$ (where $D$ is the time to compute the transition function) which, based on the size of the smallest deterministic automaton, is $O(|G|\omega|^k|\cdot |\varphi|)$, and thus not tractable even for moderately sized formulas $\varphi$. The algorithm we propose for formulas in LTL[F, G, X] works on the same automata-theoretic paradigm but with modifications that lead to a polynomial running time. In this section we outline some of the ideas that help us achieve this polynomial time.

Backwards automaton. The first observation of how our algorithm relies on is that for the fragment $\text{LTL}[F, G, X]$, there is a deterministic automaton that works backwards and only suffers an exponential blow-up (instead of double exponential in the forward automaton). That is, for the input formula $\varphi \in \text{LTL}[F, G, X]$, we construct an automaton $A_\varphi'$ such that for any trace $\tau$, $\tau \models \varphi$ iff $\tau'$ is accepted by $A_\varphi'$; for an execution $\tau = e_0e_1 \cdots e_k$ its reverse is $\tau' = e_k \cdots e_1$. The algorithm for analyzing the SLP $G$ with this automaton is also straightforward, and proceeds as if the grammar $G$ is reversed (every rule of the form $A \rightarrow BC$ becomes $A \rightarrow CB$). But, this by itself is not enough if we are using the exhaustive paradigm compute $\delta(q, A)$ for all $q$ and $A$ because $|A_\varphi'| = O(2^{|\varphi|})$. Thankfully, the backwards automaton $A_\varphi'$ enjoys a special structure that we exploit, in conjunction with the next observation, to get our efficient algorithm.

Bounding running time with visited states. We next observe that, instead of computing $\delta(q, [A])$ for all pairs of state $q$ and non-terminal $A$ in the automata-theoretic algorithm, we can afford to only compute $\delta(q, \cdot)$ for states that are actually visited (instead of all states). Consider the production rule corresponding to the starting non-terminal $S \rightarrow UV$, where $S$ is the starting non-terminal of the input SLP grammar $G$. Our final goal is to compute the state $q = \delta(q_0, [S])$. We remark that this can be computed as the composition $q = \delta(q', [V])$, where $q' = \delta(q_0, [U])$. If this is the only rule that $V$ occurs in, we only need to compute $\delta(q', [V])$ (instead of computing $\delta(q, [V])$ for every state $q$). Notice that this intermediate state $q'$ would, in fact, be visited when running the automaton $A$ on the uncompressed trace $[S]$; this is precisely the state reached after running the prefix $[U]$ of $[S]$. In fact, this observation can be generalized so that we only compute $\delta(q, A)$ for those states $q$ that are ever visited when analyzing the uncompressed trace.

We formalize this as follows. For a trace $\tau$ and an automaton $A$ with initial state $q_0$, let $\nu(A, \tau) = \{\delta(q_0, r[i]): 0 \leq i < |\tau|\}$ be the states of $A$ that are visited when running $\tau$ on $A$. Then, we can compute $\delta(q_0, S)$ by only computing $\delta(q, A)$ for every non-terminal $A$ and state $q \in \nu(A, [G])$. This gives us an upper bound of $O(|G| \cdot |\nu(A, [G])| \cdot |D|)$ which is a better bound than $O(|G| \cdot |A| \cdot |D|)$; as always $D$ denotes the time to compute the transition function.

Bounding number of states visited. Our third important observation is that any run of automaton $A_\varphi'$ (for $\varphi \in \text{LTL}[F, G, X]$) satisfies a “monotonicity” property. This property allows us to bound the number of states visited on any input to $|\varphi||\Sigma|^k$, where $k$ is what we call the nesting depth of $X$ in $\varphi$; a precise definition will be presented in Section 4. This combined with some other observations gives us a running time of $O(|G| |\varphi|^2 |\Sigma|^4)$.

Further improvements. Our algorithm for analyzing compressed executions runs in strictly polynomial time (and thus does not have the exponential dependence due to the factor $|\Sigma|^k$). We achieve this by performing an involved fine grained analysis of the running time of the algorithm. Further, we also make use of the monotonicity property outlined in the previous paragraph to optimize the space usage of the algorithm.

In Section 4, we describe the automaton construction, and the other observations about monotonicity and the number of visited states in greater detail. We finally present the algorithm for analyzing compressed executions and improvements thereof in Section 5.

4 AUTOMATON FOR $\text{LTL}[F, G, X]$

In this section, we present the construction of the deterministic backwards automaton $A_\varphi'$ such that $A_\varphi'$ accepts a trace $\tau'$ if and only if $\tau \models \varphi$; recall that $\tau'$ is the reverse of execution $\tau$. The main advantage of this construction is that the size of $A_\varphi'$ is only exponential in $|\varphi|$ (as opposed to doubly exponential). In addition, $A_\varphi'$ has a special structure that ensures that the number of states visited by $A_\varphi'$ on any input $\tau'$ bounded by a small number.

Before presenting the construction, we will introduce some conventions and notations that we will use in the rest of this paper. First, as outlined in Section 2, we can assume that $\text{LTL}[F, G, X]$ formulas are in normal form given by Equation (1), i.e., $X$ operators have been pushed as far inside as possible. For any formula $\varphi \in \text{LTL}[F, G, X]$, we will use $\text{sub}(\varphi)$ to denote the set of sub-formulas of $\varphi$. When defining our automata, we will consider a special subset of sub-formulas that we call $\text{FGXsub}(\varphi)$. These are sub-formulas $\psi$ of $\varphi$ whose topmost operator is either $F$, $G$, or $X$, and if the top operator of $\psi$ is $X$, then $\psi$ has a sub-formula with topmost operator $G$.

We formally define this set next.

Definition 1. For a formula $\varphi \in \text{LTL}[F, G, X]$, $\text{FGXsub}(\varphi)$ is the set of sub-formulas defined inductively as follows.

\[
\begin{align*}
\text{FGXsub}(a) &= \emptyset \\
\text{FGXsub}(\lnot \varphi) &= \text{FGXsub}(\varphi) \\
\text{FGXsub}(\varphi_1 \circ \varphi_2) &= \text{FGXsub}(\varphi_1) \cup \text{FGXsub}(\varphi_2), \\
\text{FGXsub}(M\varphi) &= \text{FGXsub}(\varphi), \\
\text{FGXsub}(X\varphi) &= \text{FGXsub}(\varphi), \\
\text{FGXsub}(X\varphi) &= \text{FGXsub}(\varphi), \\
\varphi &\in \text{LTL}[X]
\end{align*}
\]
Let us look at examples to highlight these definitions.

**Example 5.** Consider the formula $\varphi = \neg n \land G(n \implies \neg Xn)$ from Example 2. The set of its sub-formulas is $\text{sub}(\varphi) = \{ \varphi, \neg n, G, (\neg n) \lor \neg (Xn), (\neg n) \land \neg Xn, Xn \}$. Similarly, the set of its FGX-sub-formulas is $\text{FGXsub}(\varphi) = \{ \{ (\neg n) \lor \neg (Xn) \} \}$. Notice that $Xn \not\in \text{FGXsub}(\varphi)$ even though its topmost operator is $X$. This is because it does not have a $G$-sub-formula in its scope.

As is standard in automata constructions for LTL, our automaton $A^\varphi$ for $\varphi$ will track the truth of sub-formulas of $\varphi$ as it processes the input. Instead of tracking the truth of all sub-formulas, our automaton will only track the truth of sub-formulas in $\text{FGXsub}(\varphi)$. Since $\text{FGXsub}(\varphi)$ is smaller than $\text{sub}(\varphi)$ (as illustrated by Example 5), this results in smaller automata and better performance in practice. But this is not our only reason for tracking fewer sub-formulas. As we will show towards the end of this section, tracking the truth of fewer sub-formulas reveals that every run of the automaton is “monotonic”, which can then be exploited to argue that the number of states visited in the run of $A^\varphi$ on any string can be bounded.

Let us fix $\varphi \in \text{LTL}[F, G, X]$. The states of our automaton $A^\varphi$ will keep track of which sub-formulas in $\text{FGXsub}(\varphi)$ are true and which ones are not, on the input seen so far. Thus a state is essentially a *valuation* $h$ : $\text{FGXsub}(\varphi) \rightarrow \{ \top, \bot \}$ over $\varphi$. We use $\text{Val}_\varphi$ to denote the set of all valuations over $\varphi$.

While keeping track of the truth of sub-formulas is necessary, it is not sufficient. In order to determine truth of formulas in $\text{LTL}[X]$ like $Xa$, the automaton will additionally also keep track of the last few events seen, in its control state. How many events need to be tracked depends on the number of $X$ operators that appear in $\text{LTL}[X]$ sub-formulas of $\varphi$. Recall that we are assuming that $X$s have been pushed as far as possible in $\varphi$.

For $\psi \in \text{LTL}[X]$, define $X\text{depth}(\psi)$ to be the nesting depth of $X$ operators in $\psi$. And more generally, for $\varphi \in \text{LTL}[F, G, X]$, we define $X\text{depth}(\varphi) = \max(X\text{depth}(\psi)) \mid \psi \in \text{sub}(\varphi) \cap \text{LTL}[X]$. For example, $\text{LTL}[X]$ sub-formulas of $XG(h \implies (XXn))$ are $h$ and the sub-formulas of $XXXn$. Thus, $X\text{depth}(XG(h \implies XXXn)) = 3$. On the other hand, since the only $\text{LTL}[X]$ sub-formulas of $h \land (XGn)$ are $h$ and $n$, $X\text{depth}(h \land (XGn)) = 0$.

To compute the next state $h'$ obtained after reading a symbol $e$ in state $h$, the automaton needs to update the truth of all sub-formulas in $\text{FGXsub}(\varphi)$. It turns out that we can, in fact, compute the truth of all sub-formulas in $\text{sub}(\varphi)$ (and thus the valuation $h'$) solely by looking at $h, e$, the formula $\varphi$ and the last events seen in the trace, where $k = X\text{depth}(\varphi)$. This definition (of how truth of sub($\varphi$) is updated) is critical not only in defining the automaton but also in stating its correctness. We present this definition before giving the formal definition of $A^\varphi$. In this definition, we will use $\Sigma^{\leq k}$ to denote the set of all sequences over $\Sigma$ of length at most $k$.

**Definition 2.** Let $\varphi \in \text{LTL}[F, G, X]$, $h \in \text{Val}_\varphi$ and $b \in \Sigma^{\leq k}$. For any event $e \in \Sigma$, $\text{post}(h, b, e) : \text{sub}(\varphi) \rightarrow \{ \top, \bot \}$ is defined inductively as follows.

- $\text{post}(h, b, e)(a) = (a = e)$
- $\text{post}(h, b, e)(\neg \phi) = \neg \text{post}(h, b, e)(\phi)$
- $\text{post}(h, b, e)(\psi_1 \land \psi_2) = \land \text{post}(h, b, e)(\psi_1), \text{post}(h, b, e)(\psi_2)$, if $\land \in \{ \land, \lor \}$
- $\text{post}(h, b, e)(G\psi) = h(G\phi) \land \text{post}(h, b, e)(\phi)$
- $\text{post}(h, b, e)(F\psi) = h(F\phi) \lor \text{post}(h, b, e)(\phi)$
- $\text{post}(h, b, e)(X\phi) = \begin{cases} (b \models_f \phi) \land (h \in \text{LTL}[X] \implies X\phi) & \text{otherwise} \\ (h(\phi) \in \text{Val}_\varphi \land b \in \Sigma^{\leq k}) & \end{cases}$

Having outlined the basic intuition behind the construction of $A^\varphi$, we are ready to present its formal definition. In the following, for a function $f : A \rightarrow B$ and set $C \subseteq A$, we denote by $f \mid_C$ the restriction of $f$ to the domain $C$.

**Definition 3 (Automaton for LTL[F, G, X]).** For $\varphi \in \text{LTL}[F, G, X]$ with Xdepth($\varphi$) = $k$ and event set $\Sigma$, the DFA $A^\varphi = (\Sigma, \Sigma_0, \delta, F)$ is defined as follows.

- The states in $Q$ are triples of the form $(h, b, \text{buf})$ where $h \in \text{Val}_\varphi$, $b \in \{ \top, \bot \}$ and $\text{buf} \in \Sigma^{\leq k}$. Intuitively, $h$ tracks the truth of FGX-sub-formulas of $\varphi$ while $b$ tracks whether $\varphi$ is true on the input read so far. Additionally, $\text{buf}$ stores the last $k$ symbols read by the automaton thus far.
- The initial state $q_0$ is $(h_0, \bot, \text{buf})$ where for every $G\psi, F\psi \in \text{FGXsub}(\varphi)$, $h_0(G\psi) = \bot$ and $h_0(F\psi) = \top$.
- The transition function $\delta$ is given as follows: $\delta((h, b, 	ext{buf}), e) = (\text{post}(h, b, e)(\text{FGXsub}(\varphi)) : (h, b, \text{post}(h, b, e)(\text{buf})), \text{buf})$, where $\text{buf}'$ is the prefix of length $k$ of the concatenated sequence $e \cdot \text{buf}$.
- The final states $F = \{(h, \top, \text{buf}) \mid h \in \text{Val}_\varphi \land b \in \Sigma^{\leq k}\}$.

Let us illustrate the automaton construction with an example.

**Example 6.** Consider the formula $\varphi = \neg n \land G(n \implies \neg Xn)$ from Example 2. The backwards automaton $A^\varphi$ for $\varphi$ is shown in Figure 3; the alphabet is assumed to be $\Sigma = \{ h, n \}$. The set of sub-formulas $\text{FGXsub}(\varphi) = \{ G(n \implies \neg Xn) \}$ is singleton, and thus there are 2 valuations in $\text{Val}_\varphi$. Further, $X\text{depth}(\varphi) = 1$ and thus the buffer size is at most 1. The states of $A^\varphi$ are triples $(h, b, \text{buf})$, where $h \in \text{Val}_\varphi$, $b \in \{ \top, \bot \}$ and $\text{buf} \in \{ h, n, e \}$. In the figure, we only show the first component (valuation $h$) and the third component (the buffer) of the state. Since there is only one formula in $\text{FGXsub}(\varphi)$, we write the valuation $h$ as the truth value it maps the sub-formula to. The component $b$ can be inferred from the figure $b = \top$ in a state iff the state is an accepting state (state $q_1$).

Now consider the traces $t_1 = h\cdot h\cdot n\cdot h$ and $t_2 = h\cdot h\cdot n$. Observe that the automaton rejects the trace $t_2^r = n\cdot h\cdot h\cdot n$ but accepts $t_1^r = n\cdot h\cdot h\cdot n$ as $t_1 \not\equiv_f \varphi$ but $t_2 \equiv_f \varphi$.

The correctness proof of the automaton construction in Definition 3 relies on the following technical lemma which says that the
automaton correctly computes the truth of every sub-formula. It can be proved using an easy induction on $|\tau|$ and structural induction on the formula.

**Lemma 4.1.** For $\varphi \in \text{LTL}[F, G, X]$ let $\mathcal{A}_\varphi' = (Q, \Sigma, q_0, \delta, F)$ be the DFA as given in Definition 3. For any execution $\tau = \sigma e r$ where $e \in \Sigma$, for any $\psi \in \text{sub}(\varphi)$, $\tau \models \psi$ if and only if $\varphi(\delta(q_0, \sigma), e(\psi)) = T$.

We can now state the correctness of our automaton construction.

**Theorem 4.1.** Let $\varphi \in \text{LTL}[F, G, X]$ and $\mathcal{A}_\varphi'$ be the DFA given in Definition 3. For any execution $\tau$, $\tau \models \varphi$ if and only if $\tau' \in L(\mathcal{A}_\varphi')$.

**Proof.** Let $\tau = \sigma e r$. Observe that $\tau' \in L(\mathcal{A}_\varphi')$ iff $\delta(q_0, \tau') = (h, \top)$ for some $h$. From the definition of the transition function, this is equivalent to post($\delta(q_0, \sigma'), e(\psi)) = T$. From Lemma 4.1, this is the same as $r \models \varphi$ and thereby establishing the theorem. $\square$

**Size of $\mathcal{A}_\varphi'$.** Observe that the number of states of the automaton $\mathcal{A}_\varphi'$ is $2|\text{FGXsub}(\varphi)| + |\Sigma|^k$, where $k = \text{Xdepth}(\varphi)$. Since $\text{FGXsub}(\varphi) \subseteq \text{sub}(\varphi)$ and $|\text{sub}(\varphi)| \leq |\varphi|$, we have the number of states is $O(2|\varphi| |\Sigma|^k)$.

We next argue that though $\mathcal{A}_\varphi'$ has $O(2|\varphi| |\Sigma|^k)$ states, in any run, it goes through at most $O(|\varphi| |\Sigma|^k)$ states. This is based on the observation that state changes in $\mathcal{A}_\varphi'$ are monotonic.

Consider two valuations $g, h : \text{FGXsub}(\varphi) \rightarrow \{\top, \bot\}$. We will say that $g \preceq h$ if for every $F \psi \in \text{FGXsub}(\varphi)$, if $g(F \psi) = \top$ then $h(F \psi) = \top$ and for every $M \psi \in \text{FGXsub}(\varphi)$ where $M \in \{X, G\}$, if $g(M \psi) = \bot$ then $h(M \psi) = \bot$.

**Lemma 4.2.** For $\varphi \in \text{LTL}[F, G, X]$, let $\mathcal{A}_\varphi' = (Q, \Sigma, q_0, \delta, F)$ be the DFA defined in Definition 3. Let $u \in \Sigma^*$ and let states $(h_1, b_1, \text{buf}_1)$ and $(h_2, b_2, \text{buf}_2)$ be such that $\delta((h_1, b_1, \text{buf}_1), u) = (h_2, b_2, \text{buf}_2)$. Then, $h_1 \preceq h_2$.

The proof of Lemma 4.2 follows from the definition of the transition function and induction on the length of $u$.

Lemma 4.2 establishes that once the valuation component of the state changes, you never revisit the same valuation. Since the assignment to any $\psi \in \text{FGXsub}(\varphi)$ can change at most once, the number of valuations visited in any run is bounded by $|\text{FGXsub}(\varphi)|$, thereby giving a bound on the number of states visited in any run.

**Corollary 4.1.** Let $\varphi \in \text{LTL}[F, G, X]$ be a formula over $\Sigma$. The DFA $\mathcal{A}_\varphi'$ visits $O(|\varphi| |\Sigma|^k)$ distinct states on any input trace $\tau \in \Sigma^*$.

5 MONITORING COMPRESSED TRACES AGAINST LTL[F, G, X]

We will now present our main result — an efficient algorithm to check, given an SLP $G = (T, N, S, R)$ and $\varphi \in \text{LTL}[F, G, X]$, if $[S] \models \varphi$. Our algorithm follows the template algorithm for SLPS outlined in Section 2.3. That is, we will “run” the automaton $\mathcal{A}_\varphi'$ (Definition 3) on the uncompressed trace $[S]$, without explicitly uncompressing the SLP. This can be accomplished by computing, for every state $q$ and non-terminal $A \in N$, the state $\delta(q, [A]^k)$, and then finally checking $\delta(q_0, [S]^k) \in F$, where $q_0$ is the initial state and $F$ is the set of final states of $\mathcal{A}_\varphi'$. As pointed out in Section 2.3, this runs in $O(|G||\mathcal{A}_\varphi'|D)$ time (where $D$ is the time to compute the transition function), which given the description of $\mathcal{A}_\varphi'$, is $O(|G||\varphi||\Sigma|^k)$, where $k = \text{Xdepth}(\varphi)$. Now, we can observe that it is not necessary to compute $\delta(q, [A]^k)$ for every state $q$, but only for the states visited during a run of $\mathcal{A}_\varphi'$ on $[S]^k$. This can be accomplished if we used a “on-the-fly” algorithm for the $\delta(q, [A]^k)$ computations. For such an algorithm, given the monotonicity properties of $\mathcal{A}_\varphi'$ (Lemma 4.2) and the resulting bound on the number of visited states (Corollary 4.1), we can improve the running time to $O(|G||\varphi|^2 |\Sigma|^k)$.

The main observation in this section is that this “on-the-fly” algorithm in fact has a running time that is polynomial in the size of $G$ and $\varphi$. This requires us examine this algorithm in some detail, and analyze its running time carefully.

Recall that a state of $\mathcal{A}_\varphi'$ is of the form $(h, b, \text{buf})$ where $h \in \text{Val}_{\varphi}$, $b \in \{\top, \bot\}$ is a Boolean recording the truth of $\varphi$, and $\text{buf}$ is the buffer tracking the last $k$ symbols read. Now, consider a non-terminal $A$ and state $(h, b, \text{buf})$. Let $(h', b', \text{buf}') = \delta((h, b, \text{buf}), [A]^k)$. Based on the Definition 2, we know that the value of the Boolean $b$ does not influence the values of $h'$, $b'$, and $\text{buf}'$. This Boolean $b$ is only needed to determine if the last state (i.e., $\delta(q_0, [S]^k)$) is a final state. This can alternatively be determined from the valuation $h'$ at the end and buffer $\text{buf}'$ using a function analogous to post (Definition 2); we skip giving this definition. Next $\text{buf}'$ is nothing but the prefix of length $k$ of the concatenated string $[A]^k \cdot \text{buf}$, which can be computed in an inductive manner based on the rules in the grammar. In the interests of space, we don’t give how $\text{buf}'$ can be computed but we assume we have a function updateBuffer$(A, \text{buf})$ which returns the prefix of length $k$ of $[A] \cdot \text{buf}$.

**Algorithm 1:** Compute state of automaton $\mathcal{A}_\varphi'$ after reading the string $[A]^k$.

1. function postState$(A, h, \text{buf})$
2. visited ← visited $\cup \{(A, \text{buf})\}$
3. if $A \to e$ then
4. if $(e, \text{buf}) \in \text{visited}$ then return $h$
5. else /* $(e, \text{buf}) \notin \text{visited}$ */
6. visited ← visited $\cup \{(A, \text{buf})\}$
7. $h' \leftarrow \delta((h, \top, \text{buf}), e)$
8. if $h' \neq h$ then visited ← visited $\cup \{(A, \text{buf})\}$ return $h'$
9. else /* $A \to BC */
10. if $(C, \text{buf}) \in \text{visited}$ then $h' \leftarrow h$
11. else /* $(C, \text{buf}) \notin \text{visited}$ */
12. $h' \leftarrow \text{postState}(C, h, \text{buf})$
13. $\text{buf}' \leftarrow \text{updateBuffer}(C, \text{buf})$
14. if $(B, \text{buf}') \in \text{visited}$ then return $h'$
15. else /* $(B, \text{buf}') \notin \text{visited}$ */
16. $h'' \leftarrow \text{postState}(B, h', \text{buf}')$
17. return $h''$
18.

The critical function is really the computation of $h'$ given non-terminal $A$, valuation $h$ and buffer $\text{buf}$. A pseudocode for this function postState is given in Algorithm 1. We will call postState
with arguments $A, h, \text{buf}$ but only once. After the first call we will memo-
ize this result, and if in subsequent computations, there is a need to compute $\text{postState}(A, h, \text{buf})$ we will use the stored result. 
The data structure storing these previously computed $\text{postState}$ re-

ets is called visited in Algorithm 1. Observe that monotonicity properties of $A^*_\Sigma$ (Lemma 4.2) mean that if the valuation $h$ in the 
state changes during an execution, the automaton never returns to the 
same valuation again. Hence, visited just stores the previous 
calls for the current valuation $h$; as soon as the valuation changes, 
we reset the data structure visited because the previous calls to 
$\text{postState}$ will never be repeated as $h$ has changed. Moreover, 
this means that visited only stores pairs $(A, \text{buf})$ when a call to 
$\text{postState}(A, h, \text{buf})$ returns the valuation $h$.

In line 2, we record the fact that we have made a call $\text{postState}(A, h, \text{buf})$ by adding $(A, \text{buf})$ to visited. The computation then proceeds based on the rule for the non-terminal $A$. If $A \rightarrow e (e \in \Sigma)$ is the rule, then we return $h$ if we have computed it before (line 4) or find the new valuation by computing the transition function $\delta$. 

Note that visited is set to $\emptyset$ if the valuation changes (line 8). 
On the other hand, if the rule is $A \rightarrow BC$ (lines 10 through 17), 
then we compute the result by “running” $C$ and then $B$.

**Running time.** The running time for each call to $\text{postState}$ is 
dominated by either the time taken for line 7 or for line 14. This is 
because if we make recursive calls to $\text{postState}$ (lines 13 and 17) that 
time can be ascribed to those recursive calls. Line 7 takes at most 
time $O(|\varphi|)$ while line 14 takes $O(k)$ time (recall $k = X\text{depth}(\varphi)$). 
Thus, each call to $\text{postState}$ takes $O(|\varphi|)$ time. 

The number of possible calls to $\text{postState}$ is at most the number of 
triples $(A, h, \text{buf})$ which is $|G||\varphi||\Sigma|^k$. Thus, the total running time can be bounded by $O(|G||\varphi|^2|\Sigma|^k)$. This bound has an exponential dependence on 
k = $O(|\varphi|)$ and we will show that this can be improved.

The key to improving the bound is to do a more careful count of the 
number of $\text{postState}$ calls. Monotonicity (Lemma 4.2) ensures that 
there are at most $|\varphi|$ different valuations $h$. Therefore, for any 
fixed valuation $h$, we will try to bound the number of pairs $(A, \text{buf})$ that 
can arise as arguments in a call to $\text{postState}$ with $h$ as the valuation. 

Our observation is that this is much less than $|G||\Sigma|^k$. This 
is because if $(A, h, \text{buf})$ is an argument to $\text{postState}$, then $[A] \cdot \text{buf}$ must be a substring of $[S]$. Let us fix the uncompressed string, i.e., 
$[S]$, to be $r$. As a first step towards counting such pairs $(A, \text{buf})$, 
we define the notion of when a non-terminal $C$ is responsible for 
generating the pair $(A, \text{buf})$.

**Definition 4.** A non-terminal $C$ is said to be responsible for a 
strings $r[i : j]$ of $r$ if $C$ is the label of the lowest internal node 
of the parse tree for $r$ that has $r[i : j]$ as a substring.

Similarly, $C$ is responsible for pair $(A, \text{buf})$ if $C$ is responsible for some occurrence of the string $[A] \cdot \text{buf}$ (which is a substring of $r$).

Observe that all nodes labeled $C$ are responsible for the same 
set of pairs $(A, \text{buf})$. This is because such pairs are completely 
determined by the parse tree with root labeled $C$. Moreover, there is 
some non-terminal that is responsible for each pair $(A, \text{buf})$. Thus, 
we can upper bound the number of pairs $(A, \text{buf})$ by counting the 
number of pairs each non-terminal $C$ is responsible for. Lemma 5.1 
presents one such bound, and its proof is presented in the Appendix.

**Lemma 5.1.** A non-terminal $C$ is responsible for at most $O(H(C) + k)$ pairs; here $H(C)$ is the height of the parse tree whose root is labeled $C$.

Taking $H(G)$ to denote the height of the grammar (or $H(S)$), we 
can use Lemma 5.1 to get the following bound on the running time.

**Theorem 5.1.** Given an SLP $G$ with start symbol $S$ and formula $\varphi \in \text{LTL}[F, G, X]$, the problem of determining if $[S] \models_f \varphi$ can be solved in time $O(|G||(H(G) + k)|\varphi^2)$.

Theorem 5.1 follows from observing that Lemma 5.1 shows that the 
number of calls to $\text{postState}$ is bounded by $O(|G||(H(G) + k)|\varphi|)$ 
and the running time of each call to $\text{postState}$ is at most $O(|\varphi|)$.

## 6 EXPERIMENTAL EVALUATION

In this section, we gauge the feasibility of our proposed approach of 
monitoring execution traces by compressing them. For this, we 
evaluated the efficiency of our algorithm by comparing with the 
standard approach of monitoring traces without compressing them. 
The goal of our evaluation is two-folds:

1. **Compression ratios.** The asymptotic runtime of the algorithm 
   we propose varies quadratically with the size of the compressed 
   trace (Theorem 5.1). As a result, any speed up 
   (over the standard approach of directly analyzing uncompressed 
   traces) will evidently only be because of good compression ratios. We, 
   therefore, want to evaluate whether execution traces from real world software projects can be 
   compressed efficiently.

2. **Performance of algorithm.** Our next goal is to understand 
   how the running time varies with the size of the compressed 
   trace (SLP) in practice. Further, in order to evaluate the prac-
   tical feasibility of our approach, we want to evaluate whether our 
   algorithm for analyzing compressed traces performs better 
   than the standard approach of analyzing (uncompressed) 
   traces directly, by a good margin. Finally, we want to under-
   stand how the speed up varies with factors such as compression 
   ratio.

We next describe our implementation and experimental setup 
(Section 6.1) and then discuss our evaluation results (Section 6.2).

### 6.1 Implementation and Setup

The broad outline of our experimental setup is as follows. For our 
set of benchmark programs, we extract execution traces using an 
off-the-shelf logging tool. We then compress these traces as straight 
line programs (SLPs) and analyze the SLPs thus generated using our 
algorithm detailed in Section 5. We also compare the running 
time of our algorithm with the time it takes to analyze the original 
uncompressed traces against finite state automata corresponding to 
our LTL specs.

**Implementation.** We implemented our algorithm for analyzing compressed traces against LTL[F, G, X] in our tool ZipMOP. Zip-
MOP is primarily implemented in Java (in about 500 LoC). We use JavaMOP [1, 11, 33] for extracting execution traces. JavaMOP is a runtime verification framework that can monitor Java programs 
against formal specifications written in one of various specification 
languages, including LTL. JavaMOP instruments a Java program
under test to add monitoring code to identify what events to monitor, and what action to perform at each event. For our experimental evaluation, we obtained execution traces by modifying JavaMOP so that it logs events to a file. To analyze uncompressed traces against LTL[F, G, X] properties, use Rabinizer-4.0 [24] publicly available at [2]. Rabinizer-4.0 is a state-of-the-art tool for translating LTL formulae into automata. For each of the LTL properties we consider, we obtain deterministic finite automata using Rabinizer-4.0 and check if this automata accepts the trace in consideration.

**Benchmarks and Traces.** Our subjects are open source repositories obtained from GitHub. We obtained GitHub repositories from a prior empirical study on GitHub projects [28] as well as independently from GitHub based on their popularity score (measured by GitHub stars). We use JavaMOP to instrument these repositories so that all events of interest (those that occur in any of the LTL specs) are logged. We then generated traces by running all test classes of these repositories. We chose the top 100 traces based on the trace lengths. The minimum, maximum and average trace lengths in this set are 52.6M, 1.03B and 209M. The overall distribution is given in Figure 4a.

**LTL Specifications.** Our LTL properties are also obtained from the suite of properties collected in the empirical study [28]. Most of these properties specify the expected usage of different data structures and APIs used in these software projects and are expressed in many different formalisms (regular expressions, ERE, LTL, FSM, etc.). An example property is the property $\neg n \land G(n \implies \neg Xn)$ from Example 2 (in Section 2.2), that specifies how an iterator of the Set collection must be used — every call to next(τ) (denoted by ‘n’) must be immediately preceded by a call to hasNext(τ) (denoted by ‘h’). Another such property is $\varphi = G(cr \implies f(cl))$ which states that a resource (such as a buffered stream) must be eventually closed (‘cl’) every time it is created (‘cr’). We selected a total of 10 properties that were expressible in the fragment LTL[F, G, X]. We list them in our Appendix.

**Setup.** We compare the running times of our algorithm over compressed traces to the time for analyzing the corresponding uncompressed traces against our LTL[F, G, X] specifications. After obtaining traces from our benchmark projects (using JavaMOP), we compress these traces using the Sequitur algorithm [35] which runs in linear time in the size of the uncompressed trace. We use a publicly available implementation [3] of Sequitur. For the uncompressed traces, we use Rabinizer-4.0 to generate a deterministic finite state automaton for each property. For every property, Rabinizer-4.0 generates a Rabin automata, which is essentially a finite state machine, together with an acceptance condition for deciding membership of infinite words. For our purpose, however, we transform the acceptance condition of these automata so that they are suitable for analyzing finite traces. We manually encoded these automata in Java. All our experiments were conducted over a 2.6GHz 64-bit Linux machine.

### 6.2 Evaluation Results

**Size of Compressed Traces and Compression Ratios.** While the uncompressed traces have lengths varying from 50M to 1.03B, the sizes of the compressed traces (SLPs) all lie between 54k and 1.8M. The average size of the SLPs is approximately 329k and the overall distribution is presented in Figure 4b. The compression ratios of each trace was observed to be at least 277. The maximum and average compression ratios are 1016 and 641, and the distribution is as shown in Figure 4c. The significant compression ratios hint that most open source projects generate execution traces which have a lot of repetitions which can be effectively represented in a compressed format. A plausible explanation is that many of these test cases repeatedly manipulate collection objects (such as lists or sets), giving rise to long repetitive patterns, making them amenable to effective compression.

**Running times.** In Figure 5a, we plot the running time (in seconds) for every compressed trace. These times are averaged over the running time of ZipMOP across all the 10 LTL[F, G, X] properties we consider. Further, in order to ensure fair comparison with the analysis over uncompressed traces, we exclude the time to read (uncompressed or compressed) trace files in memory — including I/O times would penalize the uncompressed analysis more heavily as they work over larger files. Observe that all the times are within 0.5 second (excluding I/O time). Also observe that, as expected, the times increase with the size of the compressed trace (SLP). In fact, we can see that the time increases linearly with the size of the SLP, despite the worst case dependence of $|G|^2$ as in Theorem 5.1 ($H(G)$ can be $O(G)$ in worst case).

**Speed-up over analysis of uncompressed traces.** We now compare how the running time over compressed traces compare with the running time of analyzing uncompressed trace logs. Figure 5b shows the speed up $\frac{\text{Time to analyze uncompressed traces}}{\text{Time to analyze SLP}}$, where as before both the numerator and denominator are average times...
over all LTL specs. Further, both the times exclude I/O time as including it would penalize the uncompressed analysis much more heavily. The maximum, minimum and average speed ups are 90×, 15× and 34×. The high speed up shows the power of compression in analyzing trace logs as compared to uncompressed versions.

In Figure 6a, we show how the speed up varies with the compression ratio. As expected, our algorithm would perform better (as against the uncompressed analysis) when the compression ratio is high. This is because the time to analyze an uncompressed trace $\tau$ is $O(|\tau|)$ (time to check membership in a finite automaton) and the time to analyze a compressed trace $G$ using our algorithm (Section 5) is proportional to $O(|G|)$ and the speed-up thus increases with the ratio $O(|\tau|/|G|)$.

In Figure 6b, we analyze the efficiency of the algorithm, defined as $\eta = \frac{\text{Speed up}}{\text{Compression ratio}}$. The efficiency factor intuitively captures how well the speed up over uncompressed traces be explained using the compression ratio. We observe that the efficiency values are in the range 0.04 to 0.11, and this is likely because of the constant multiplicative factors involved in the running time of our algorithm for checking compressed traces. Further, the efficiency factor increases (almost) monotonically with the size of the compressed format, implying that higher compression ratios are more effective when the compressed traces are themselves large.

7 RELATED WORK

We show that the problem of checking if an execution trace, when compressed as an SLP (lossless grammar based compression scheme) can be solved in polynomial time for a fragment of LTL. From a theoretical standpoint, the work that is closest to ours is the intractability result (PSPACE-hardness) due to Marky and Schnoebelegen [32] for full LTL, where notably the hardness arises from the use of arbitrarily nested until operators in LTL formulae. Lohrey [29] surveys algorithmic and complexity-theoretic questions in checking problems such as language membership and pattern matching over strings compressed as SLPs. More generally, there is a rich history of studying algorithms and complexity of various problems such as graph reachability where the input is presented succinctly, starting from the work of Galperin and Wigderson [15]. Typically, graph problems that are tractable on the uncompressed input become intractable when posed over their compressed representation [7, 8, 12, 14, 30, 36, 41].

The problem of checking if an execution trace satisfies certain property is the central question in the area of runtime verification [17, 18, 20, 27, 34]. Amongst various different formalisms to state properties of interests, LTL [37] remains a popular choice. Havelund and Rosu [19] proposed a dynamic programming based algorithm for analyzing execution traces against LTL specifications, without explicitly constructing a finite state automaton corresponding to the input formula, unlike in model checking [6]. JavaMOP [11, 33] provides a monitoring oriented framework for monitoring various kinds of properties including LTL, finite state machines, extended regular expressions, etc...

The idea of using compact representations in the context of software engineering dates back to Larus’s whole program paths [26], which was extended to the case of concurrent programs [16]. Wang and Roychoudhury [40] propose techniques for dynamic slicing over traces compressed as straight line programs. Recently, Kini et al [23] propose polynomial time algorithms for detecting data races over concurrent program executions compressed as SLPs.

8 CONCLUSIONS

We propose the use of compression as an algorithmic paradigm to improve the efficiency of checking if execution traces conform to specifications written in LTL (linear temporal logic). While
this problem is intractable (PSPACE-hard) in general, we establish a polynomial time algorithm for the rich fragment LTL$[F, G, X]$ whose formulae do not include the U operator of full LTL. Our polynomial time algorithm leverages a monotonicity property in the automata theoretic representation of formulae in LTL$[F, G, X]$. On a comprehensive benchmark suite of open source Java projects, our evaluation confirms that execution traces can be effectively compressed and that the membership problem of traces can be efficiently decided over compressed formats (straight line programs), without decompressing them, resulting into significant speed ups when compared to analysis over uncompressed traces.
A PROOF OF LEMMA 5.1

In this section, we give the proof for Lemma 5.1. The definition of responsibility implies that a non-terminal $C$ is responsible for a pair $(A, buf)$ if $C$ is responsible for buf and buf immediately follows $A$. Let us assume without loss of generality that $C \rightarrow C_1C_2$ and $|C_1| \geq k - 1$, $|C_2| \geq k - 1$ (if $|C_1| < k - 1$ or $|C_2| < k - 1$, $C$ will only be responsible for less pairs). Let $\beta_1, \beta_2, \ldots, \beta_k$ be the rightmost $k$ leaves of the parse tree of $C_1$. Leaves are terminals, and we can view buffers as a sequence of consecutive leaves. For each of these leaves we can define a corresponding buffer such that buf$_i$ is the string composed by the $\beta_{i+1}, \ldots, \beta_k$ and the leftmost $i$ leaves of $C_2$. The first observation is that $C$ is already responsible for each pair of $(\beta_i, \text{buf}_i)$ and these count as $k$ pairs. The second observation is that if $C$ is responsible for a pair $(A, \text{buf})$ then buf must be one of buf$_1, \text{buf}_2, \ldots, \text{buf}_k$ and this means the rightmost leave of the parse tree of $A$ must be one of the $k$ leaves. This implies that the position of each node $A$ determines whether $C$ is responsible for some pairs containing $A$. All we need to count is the number of nodes (from now on we assume that every instance of node is in the scope of the parse tree of $C_1$) whose parse trees’ right most leaves are one of these $k$ leaves. This is equivalent of counting the number of nodes who are ancestors of these $k$ leaves because if a node $A$ is an ancestor of $\beta_i$, then its rightmost leaf must be a leaf $\beta_j$ for some $i \leq j$ and therefore $C$ is responsible for $(A, \text{buf}_j)$. Let $P$ be a path from $C_1$ to $\beta_1$, every node on the path is an ancestor of $\beta_1$ and hence count as $H(C)$ many pairs. Then it’s important to observe that every node on the right of $P$ is an ancestor of at least one of the rightmost $k$ leaves and none of the node on the left of $P$ is an ancestor of these leaves. There are exactly $k - 1$ leaves on the right of $P$ and there are at most $k - 1$ non-leave nodes in the same section. This is because what on the right of $P$ are disjoint full binary sub-trees. Each of those sub trees contains some of the $k - 1$ leaves. Therefore in total they contain no more than $k - 2$ non-leaf nodes. These count as another $k - 2$ pairs. So we have proved that $C$ can be responsible for at most $k + k - 2 + H(C) = O(H(C) + k)$ many pairs.
B LTL SPECIFICATIONS USED IN EXPERIMENTS.

Our LTL properties are obtained from the suite of properties collected in the empirical study [28]. Most of these properties specify the expected usage of different data structures and APIs used in these software projects and are expressed in many different formalisms (regular expressions, ERE, LTL, FSM, etc.). Here we present a list of properties that we used in the experiment and provide the English description for some LTL formula that is very specific.

\[
G(\text{create} \implies G(\text{update} \implies \neg X(\text{next})))
\]
The property monitors on the safe usage of iterators which requires that the iterator should not be modified during iteration.

\[
\neg \text{next} \land G(\text{hasNextFalse} \implies \neg X(\text{next}))
\land G(\text{next} \implies \neg X(\text{next}))
\]

\[
\neg \text{next} \land G(\text{next} \implies \neg X(\text{next}))
\]

\[
G(\text{create} \implies X(\text{explicit} \lor \text{implicit}))
\]
The property ensures a temporary file is explicitly deleted or scheduled to be deleted right after it is created.

\[
\neg F^* \land \text{sync} \land X(\text{createset}) \land XX(\text{asyncCreateIter} \lor (\text{syncCreateIter} \land X(\text{accessIter})))
\]
The property is looking for violations when a synchronized collection is created but an unsynchronized iterator is used for the collection or an synchronized iterator is used for the collection in an unsynchronized way.

\[
G(\text{create} \implies F(\text{close}))
\]

\[
G(\text{badset} \implies G(\text{badset}))
\]

\[
\neg G(\text{close} \lor G(\text{manipulate}))
\]
This property captures the violation when a closed object performs input or output operations.

\[
(\text{create1} \lor \text{create2} \lor \text{create3} \lor \text{create4})
\land X(G(\text{write}) \lor G(\text{read}))
\]
JavaMOP keeps track of size of the pushback buffer and record it as unsafeunread when an unread() is called but the internal pushback buffer is full. The LTL formula captures any violations that has an unsafeunread.

\[
\text{mark} \land G(\text{mark} \lor \text{reset})
\]