Briefings in Bioinformatics, 2024, pp. 1–14

doi:
Advance Access Publication Date: Day Month Year
Paper

PAPER

Formal Verification of Bioinformatics Software using Model Checking and Theorem Proving

Hansika Weerasena[1],* Aruna Jayasena[1],* Christina Boucher[1] and Prabhat Mishra[1]

[1]Department of Computer & Information Science & Engineering, University of Florida, USA
*Corresponding author. hansikam.lokukat@ufl.edu
FOR PUBLISHER ONLY Received on Date Month Year; revised on Date Month Year; accepted on Date Month Year

Abstract

Motivation: While there is explosive growth in the creation of biological data, researchers rely on ad-hoc verification methods such as testing with small simulated datasets. Due to their importance in biology and biomedicine, there is a critical need to verify these algorithms as well as their implementations to ensure that the results and conclusions are trustworthy. In this paper, we explore an effective combination of model checking and theorem proving of bioinformatics software, including BiopLib [1], BWA [2], Jellyfish [3], SDSL [4], Dashing [5], SPAdes [6], and MUMmer [7].
Results: We provide results for model checking for bioinfomatics software libraries and theorem proving for specific properties. Our model checking framework found several potential flaws in the two tools (BiopLib [1] and BWA [2]), as shown in Table 12. We have also detected several failing cases in Succinct Data Structures Library (SDSL).
Availability: Our implementation is open source and available at `https://github.com/UFESL/BioInfoVerify`.

Key words: Succinct data structures, Burrows–Wheeler transform, positional Burrows–Wheeler transform, Pattern matching, Formal verification, Model Checking, Property Checking, Theorem Proving

Introduction

Advances in high-throughput sequencing of DNA and other biological samples provide holistic investigatory capabilities to address complex problems in biomedical science. Numerous advancements in biology and health-related sciences have dramatically changed due to the development of technology that is capable of generating biological data and the accompanying software tools to analyze the data. Although advances in bioinformatics have focused primarily on the development of efficient and scalable tools to process vast amounts of biological data, ensuring the accuracy and reliability of these tools remains a critical but underexplored challenge. The accuracy of bioinformatics software is traditionally evaluated using simulated data, which provides a "ground truth" for comparison. However, most validation methods in the bioinformatics domain are ad hoc, relying on testing a limited set of scenarios. This lack of complete validation has led to instances in which published findings are later contradicted, resulting in the retraction of numerous articles; for example, several articles related to cancer microbiome research have been invalidated due to flawed analysis [8]. These issues highlight the urgent need for rigorous verification of bioinformatics algorithms to ensure the trustworthiness of results.

Addressing this need requires a closer examination of the existing validation methodologies, which can generally be categorized into two main types: simulation-based validation and formal methods. Simulation-based validation employs random or constrained random tests to verify software implementations. In bioinformatics, validation is often complicated by the absence of a golden reference, which is a standard reference genome for the species of interest, such as the GRCh38 (hg38) human genome assembly from the Genome Reference Consortium. In these cases, alternative methods such as metamorphic testing [9, 10] are used to assess software correctness. However, these methods are inherently limited in scalability and cannot exhaustively cover all possible input sequences for large software designs.

In contrast, formal methods offer promising solutions for specific application scenarios, such as verifying distributed AI [11], selecting appropriate model checkers [12], validating sequencing workflows [13], adhering to regulatory standards [14], and analyzing bioinformatics workflows for respiratory tract infections [15]. Despite their potential, these approaches are often narrowly focused and do not provide a comprehensive framework applicable to the diverse range of bioinformatics algorithms and implementations.

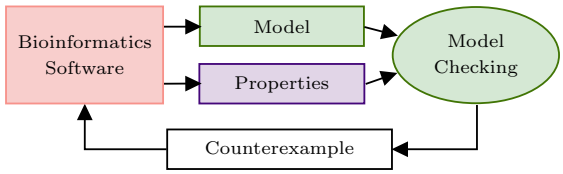

Fig. 1. Overview of model checking.

In this paper, we present a comprehensive and scalable verification framework using an effective combination of model checking and theorem proving to ensure the correctness of bioinformatics software. As shown in Figure 1, model checking provides a mechanism to verify whether a system meets a given specification by checking against a set of properties. In case the system does not hold the given property, it provides a counterexample that can reproduce the violation of the property. While model checking is a promising solution for verifying small systems, it can lead to state space explosion when dealing with large systems or complex scenarios. In contrast, theorem proving can deal with large systems since it relies on logical reasoning rather than state-space enumeration, as shown in Figure 2.

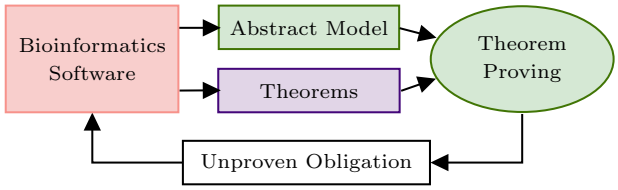

Fig. 2. Overview of theorem proving.

This paper combines the advantages of both model checking and theorem proving for effective validation of tools used by biomedical applications, as shown in Figure 5. We also perform a static analysis of bioinformatics software to detect common software vulnerabilities. Specifically, this paper makes the following major contributions:

- We derive properties for different application-specific data structures and algorithms used in bioinformatics software.
- We propose model checking to formally verify bioninformatics software libraries against expected behaviors (properties).
- We propose theorem-proving based formal verification of expected behaviors in bioinformatics applications.
- The experimental results across various bioinformatics libraries and software demonstrate that an effective combination of model checking and theorem proving can identify critical flaws in bioinformatics software.

Background

Verification is the process of ensuring that software and systems behave according to their specified requirements. There are several ways to perform verification: simulation-based validation and formal verification. Simulation-based validation is scalable, yet incomplete, due to limited state exploration. Specifically, simulation-based validation faces the challenge of exponential input space complexity, which can make it difficult to achieve comprehensive coverage, particularly for rare or edge-case scenarios. In contrast, formal verification provides a higher degree of assurance by mathematically proving system properties and exhaustively exploring possible system states, making it well-suited for applications where completeness and correctness are critical. We first provide an overview of two popular formal verification methods, model checking and theorem proving. Next, we discuss the fundamental challenges in verifying bioinformatics software.

Model Checking

Formal verification involves mathematically proving that a software implementation adheres to its specification. Model checking [16], also known as property checking, is a formal verification strategy that ensures that a software implementation satisfies its intended behaviors (properties). This approach helps uncover issues that might be overlooked in standard test cases by focusing on invariant behaviors that should hold true for all inputs. For example, in bioinformatics, a property may validate that a data structure should always maintain sorted order after insertions.

Model checking verifies whether a system meets a given specification by systematically exploring all possible states of the system. Model checking provides exhaustive coverage by analyzing every potential state of the system. For example, in verifying a sequence alignment algorithm, model checking would ensure that the algorithm produces the correct alignment even for edge cases, such as sequences with highly repetitive regions. This exhaustive state exploration helps identify hidden errors that might not be discovered through traditional testing. SAT-based Bounded Model Checking (BMC) [17] is ideal to check the safety properties. When checking multiple related properties, the checker can learn between bounds in a property and between related properties to reduce the search space [18].

Table 1. Basic logical operators

	Operation	Example
$\neg$	not	$\neg P$ is false if P is true.
$\wedge$	and	$P \wedge Q$ is true if both P and Q are true. False otherwise.
$\vee$	or	$P \vee Q$ is true if either P or Q is true. False otherwise.
$\rightarrow$	implication	$P \rightarrow Q$ is false when P is true and Q is false. True otherwise
$\Leftrightarrow$	equivalent	$P \Leftrightarrow Q$ is true if P and Q have same truth values. False otherwise.

To conduct model checking, the system specification should be represented as a set of properties. Property generation involves translating high-level goals into temporal logic, often using Linear Temporal Logic (LTL). Typically, specification of design (i.e., software) is used to generate these properties. This section outlines temporal logic as well as its variations that are utilized to describe the properties.

Temporal Logic: Temporal logic is widely used to express different types of properties. For example, propositional logic allows us to reason about the truth or falsehood of logical expressions. A proposition is evaluated to be either true or false. Multiple propositions can be connected using logical operators to form a propositional formula. Table 1 shows a few commonly used logical operators. The table is sorted by the order of precedence, with '$\neg$' having the highest precedence.

Table 2. Basic temporal operators in LTL [16]

	Semantics	Description
$X\ p$	next	p is true in the next state of the path.
$G\ p$	always	p is true at every state on the path.
$F\ p$	eventually	p is true at some future state on the path.
$p\ U\ q$	until	q is true at some future state, and at every preceding state on the path, p is true.

Linear Temporal Logic (LTL): LTL extends the propositional logic by introducing the notion of timing [19]. It is used to describe a sequence of transitions along a path. Some of the temporal operators that are used to describe the transitions are given in Table 2, where p and q are propositions which can

contain both logical and temporal operators. For example, $F\ p$ represents that eventually, p will be true at some future state, as shown in Figure 3(a), where each node represents a state. Similarly, Figure 3(b) shows $p\ U\ q$ which represents that p is true in every preceding state before q is true.

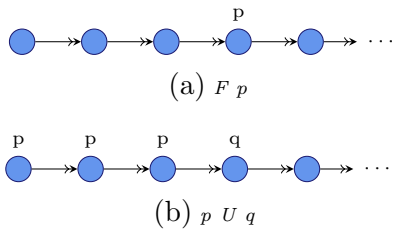

Fig. 3. State diagram of two example temporal operators in LTL.

Computational Tree Logic (CTL): Computation tree logic (CTL) [20] is a branching-time logic, where the future of a state is not determined. To deal with the many possible paths in the future, CTL introduces two types of path qualifiers, as shown in Table 3. While "Ap" represents that all possible execution paths from the current state should have the property p, "Ep" represents that there exists at least one path such that p is true. The temporal operator in CTL is composed of a path qualifier and a temporal operator from LTL. For example, $AF\ p$ represents that for all paths from the current state, there must exist one state where p is true, as shown in Figure 4(a). Figure 4(b) shows the state diagram of $EG\ p$, which means that there exists at least one path where p is true for each state along the path.

Table 3. Path quantifiers in CTL [21]

	Description
$A\ p$	p is true in **all** the paths starting from the current state.
$E\ p$	p is true in **some** path starting in the current state.

CTL has the restriction that each LTL operator must be immediately preceded by a path qualifier, e.g., $EG(AF\ p)$. CTL* is a super-set of both LTL and CTL, and removes this restriction. Although CTL* offers better expressiveness than LTL and CTL, there are still some limitations in CTL*, such as its lack of support for expressing quantitative constraints over positions or occurrences of events along a path, a feature known as the ability of counting. For example, it cannot express that p should be true for two states before q is true in a path. Researchers have proposed QCTL (quantified CTL) [22] that adds the ability of counting. There are different variations of temporal logic to solve different problems, but they are beyond the scope of this paper.

Theorem Proving

Theorem proving is a rigorous approach where behaviors (properties) of a system are expressed as logical theorems, and these theorems are formally proven using mathematical reasoning and proof techniques. Unlike testing, which checks for correctness over a subset of inputs, theorem proving ensures correctness across all possible inputs and states. For example, in seeding alignment in bioinformatics, theorem proving can verify that all maximal exact matches between a read and a reference genome are identified, providing guarantees that are critical for high-stakes applications.

The theorem proving process begins with a *formal specification* of an algorithm, which is a detailed mathematical

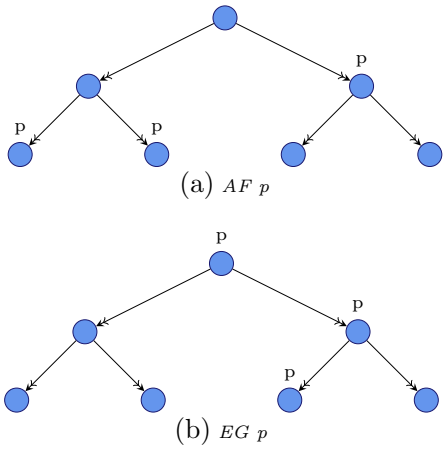

Fig. 4. State diagram of two example temporal operators in CTL.

description of the algorithm. These theorems typically represent safety or liveness properties, or application-specific requirements, that the system should satisfy. Safety properties assert that "nothing bad happens" by ensuring that the system avoids undesirable states such as data races, deadlocks, or invalid memory access. Liveness properties, on the other hand, assert that "something good eventually happens" by guaranteeing that the system makes progress such as completing tasks, responding to inputs, or eventually reaching a desired state. A *proof assistant* is then utilized, which is a software tool designed to help developers construct and check the validity of logical arguments within the formal specification. The proof assistant aids in the *generation of proof obligations*, which are essentially conditions that need to be proven true for the properties to hold for the given formal specification. Subsequently, the *verification of proof obligations* takes place, wherein each generated obligation must be verified. If all the proof obligations are successfully verified, the system is deemed to have been *verified* and thus meets the defined specifications and properties. However, if any obligation is not verified, it leads to the system being marked as *not verified*. Our proposed framework guarantees correctness through formal, machine-checked proofs

Need for Verifying Bioinformatics Software

Computational biology pipelines typically involve two major steps: bioinformatics-based data analysis and subsequent biological interpretation or further experiments informed by the results. Even subtle software bugs, such as memory errors, pointer overflows, or crashes during read alignment, can silently propagate incorrect outputs in bioinformatics softwares. These errors often go unnoticed until they compromise entire experimental analyses, leading biologists to pursue false leads, redesign experiments, or invalidate results after costly delays. By providing formal machine-checked proofs of correctness, our framework eliminates such hidden vulnerabilities, ensuring that the foundational tools operate as intended in all input scenarios. This reliability is critical to avoid inefficiencies and wasted resources in biomedical laboratories, where researchers are heavily dependent on the accuracy of computational tools. Ultimately, our approach improves trust in bioinformatics software by preventing implementation-level flaws that could otherwise distort scientific interpretation and outcomes.

Challenges in Verifying Bioinformatics Software

Due to the diverse and complex nature of bioinformatics algorithms, traditional testing methods often fall short. State space explosion, where the number of potential states to explore becomes unmanageable, is a significant hurdle in applying model checking effectively. This challenge is especially pronounced in bioinformatics, where algorithms often process vast datasets and intricate workflows.

Another challenge is the scalability of verification tools when applied to real-world bioinformatics software. Complexities such as data dependencies, algorithmic variability, and interaction with external data sources can make the verification process difficult. Optimization techniques, such as abstraction and property decomposition, are essential to mitigate these issues by reducing the verification complexity and focusing on the most critical aspects of the system.

Bioinformatics software often requires handling noisy data and accommodating biological variation, making it challenging to define precise properties that hold universally. These unique requirements necessitate tailored verification approaches to ensure comprehensive and reliable verification outcomes.

Related Work

The existing validation approaches can be broadly divided into two categories: simulation-based validation and formal methods. We first briefly outlined the related efforts in these two categories. Next, we discuss their limitations to highlight the need for our proposed framework.

Simulation-based Validation: Yang et al. examined two important aspects that are central to bioinformatics analysis – software scalability and validity [23]. Specifically, this paper highlighted the difficulty of determining the correctness of bioinformatics software due to the large input space. Roy et al. provided guidelines for validating next-generation sequencing bioinformatics pipelines [24] including practical guidance for laboratories regarding NGS bioinformatics pipeline design, development, and operation. Bogaerts et al. presented a validation strategy that focused specifically on the exhaustive characterization of the bioinformatics analysis of a whole-genome sequencing workflow [13]. Several approaches have utilized metamorphic testing to verify bioinformatics applications. Metamorphic testing is a software testing technique that validates programs by identifying and exploiting expected relationships between inputs and outputs, even in the absence of a golden reference. Chen et al. presented a framework for validation of bioinformatics programs using metamorphic testing [9]. It verifies the correctness of test outputs by checking if they conform to domain-specific properties. Similarly, Giannoulatou et al. utilized metamorphic testing for validation of bioinformatics software without a golden reference model [10].

Formal Verification: Kugler outlined the challenges in applying formal verification in engineered biological systems [25]. Pereira et al. outlined application of model checking to verify distributed artificial intelligence (AI) in bioinformatics [11]. This paper only focuses on verifying a distributed system in biology that uses AI. Bakir et al. introduced a method for automatic selection of a model checker for verification of biochemical models [12]. By using statistical model checking tools like PRISM, PLASMA-Lab, and Ymer on various biological models and using machine learning techniques, the authors developed a predictive system with over 90% accuracy on model selection. Note that they did not consider any bioinformatics applications. SoRelle et al. provided a step-by-step guide to navigate the regulatory and validation standards of implementing a bioinformatics pipeline as a part of a new clinical next-generation clinical assays [14]. Jin et al. outlined bioinformatics analysis and verification of key genes associated with recurrent respiratory tract infections [15]. Cooper et al. developed a software program that implements multiple strategies for the selection of verification candidates [26].

Although existing approaches provided important guidelines for verification, they demonstrated applicability on very specific application scenarios. *To the best of our knowledge, there are no comprehensive formal verification framework that can be applied on a wide variety of bioinformatics algorithms and implementations.*

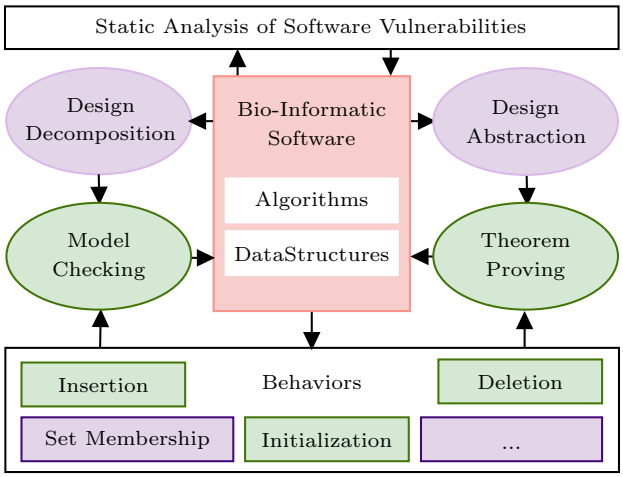

Fig. 5. Overview of the proposed formal verification framework that consists of model checking and theorem proving.

Formal Verification Framework

Figure 5 provides an overview of our proposed framework. Our main contribution is the verification of bioinformatics implementations through an integrated framework that leverages model checking and theorem proving for verifying specific properties. We first describe various steps during model checking of bioinformatics libraries. Next, we discuss theorem proving of bioinformatics applications.

Model Checking of Bioinformatics Software

Figure 5 shows three major components in model checking based formal verification: model (software), property (expected behavior), and model checking. We first outline the model using Kripke structure, which represents the behavior of systems in terms of states and transitions. Next, we describe how to derive properties to cover various functionalities. Finally, we discuss learning and decomposition methods to reduce the model checking complexity.

Modeling of Bioinformatics Software

A Kripke structure is a mathematical framework used to model the behavior of a system in terms of states, transitions, and properties associated with those states. A Kripke structure is formally defined as $K = (S, I, R, L)$ where:

- S is the set of states representing possible stages or steps in the software.
- $I \in S$ is the set of initial states.
- R is the binary relation representing transitions between states, denoting the actions or events in the algorithm.
- L is the labeling function that assigns labels to states, providing additional information about the properties or conditions associated with each stage of the process.

A Kripke model was created for each software evaluated to conduct model checking.

Property Generation

To conduct model checking, properties must be derived from system specifications. This involves translating high-level goals into formal temporal logic, often using LTL, as discussed in Section 2. Bioinformatics applications frequently rely on specialized data structures such as Wavelet Trees and De Bruijn Graphs. Properties for model checking are generated based on the functionalities provided by these data structures. The remainder of this section covers widely used data structures in bioinormatics libraries and derives the corresponding properties to cover their functionalities.

Radix Trees: Radix trees [27] (or prefix trees) are space-optimized tree structures that store keys as common prefixes, making them suitable for tasks like indexing and searching genomic data. In bioinformatics, they are used for efficient storage and retrieval of sequences, such as k-mers or motifs. Table 4 illustrates the properties generated for verifying the behavior of radix tree implementations. For example, the radix tree should implement correct logic for prefix matching, which allows rapid identification of all sequences that share a common prefix. When performing a prefix match, if a prefix is matched, it should return all sequences that start with that prefix.

Bloom Filters: Bloom filters [27] are probabilistic data structures used to test whether an element is a member of a set, with a small probability of false positives but no false negatives. In bioinformatics, they are widely used for tasks like k-mer membership queries, error correction, and indexing large genomic datasets. Table 5 illustrates the properties that are used to verify the implementation of the bloom filters. For example, the bloom filter should always set the corresponding bits correctly for all hash functions during an insertion.

FM-index: FM-index [28] is a compressed suffix array data structure that enables efficient substring searches while using minimal memory. It combines the Burrows-Wheeler Transform (BWT) [29], a reversible text transformation that reorganizes a string into runs of similar characters to facilitate compression, with auxiliary data structures such as occurrence counts and rank queries to enable fast exact or approximate pattern matching. In bioinformatics, FM-index is widely used in tools like BWA for genome alignment and sequence mapping. Table 6 illustrates the properties generated for verifying the behavior of FM-index implementations. For example, property P_3 ensures that the FM-index can efficiently count how many times a specific character appears in the indexed text up to a given position. Similarly, property P_3 checks that the FM-index can find all starting positions of a given substring (or pattern) in the original text. It ensures accurate and efficient pattern-matching functionality.

Wavelet Trees: Wavelet trees [27] are hierarchical data structures that provide compact and efficient representations of strings and sequences by organizing them based on their alphabet. They support operations like rank, select, and access in logarithmic time, making them ideal for tasks such as compressed text indexing, pattern matching, and sequence analysis. In bioinformatics, wavelet trees are used in applications like FM-indexes to enable efficient substring queries and genome compression. Table 7 illustrates the properties generated for verifying the behavior of wavelet tree implementations. For example, property P_4 ensures that correctness of finding the position of the k-th occurrence of a specific character in the sequence. Similarly, property P_8 confirms that the wavelet tree efficiently compresses a sequence, storing it in a space proportional to the product of the sequence length and the logarithm of the alphabet size.

Interval Graphs: Interval graphs [27] are graphical representations where each vertex corresponds to an interval on a real line, and an edge exists between two vertices if their intervals overlap. They are particularly suited for modeling relationships between overlapping genomic features, such as exons, regulatory regions, or read alignments. In bioinformatics, interval graphs are used in applications like genome annotation, synteny analysis, and detecting overlaps in sequencing data. Table 8 illustrates the properties generated for verifying the behavior of interval graph implementations.

Table 4. Properties for Radix Trees

Functionality	**Property**
Insertion	$P_1 : \forall \text{seq} \in \text{Seq} : G\,(\text{Insert}(\text{seq}) \rightarrow \exists n \in \text{Tree} : n.\text{seq} = \text{seq} \wedge \forall m \in \text{Tree} : m \neq n \implies m.\text{seq} \neq \text{seq})$
Search	$P_2 : \forall \text{seq} \in \text{Seq} : G\,(\text{Search}(\text{seq}) \rightarrow (\text{R} = \text{seq} \implies \exists n \in \text{Tree} : n = \text{seq}) \wedge (\text{R} = \emptyset \implies \forall n \in \text{Tree} : n \neq \text{seq}))$
Prefix Match	$P_3 : \forall \text{pr} \in \text{Pref} : G\,(\text{match}(\text{pr}) \rightarrow \forall n \in \text{Tree} : (n.start = \text{pr} \implies n.\text{seq} \in \text{R}) \wedge (n.start \neq \text{pr} \implies n.\text{seq} \notin \text{R}))$
Deletion	$P_4 : \forall \text{seq} \in \text{Seq} : G\,(\text{Delete}(\text{seq}) \rightarrow \forall n \in \text{Tree} : n.\text{seq} \neq \text{seq} \wedge (\exists m \in \text{Tree} : m.\text{seq} = invalid(\text{seq})))$
Integrity Check	$P_5 : G\,(\forall n \in \text{Tree} : \text{ValidNode}(n) \wedge \text{Integrity}(n) \wedge (\text{Path}(n) \text{ is consistent with parent nodes}))$

Table 5. Properties for Bloom Filters

Functionality	**Property**
Initialization	$P_1 : G\,(\text{Initialize}() \rightarrow \forall i \in [0, m) : \text{BF}[i] = 0 \wedge \text{Count} = 0)$
Insertion	$P_2 : \forall \text{seq} \in \text{Seq} : G\,(\text{Insert}(\text{seq}) \rightarrow \forall k \in [1, K] : \text{BF}[h_k(\text{seq})] = 1)$
Membership Query	$P_3 : \forall \text{seq} \in \text{InsertedSeq} : G\,(\text{Query}(\text{seq}) \rightarrow \forall k \in [1, K] : \text{BF}[h_k(\text{seq})] = 1)$
Membership Query	$P_4 : \forall \text{seq} \notin \text{InsertedSeq} : G\,(\text{Query}(\text{seq}) \rightarrow (\forall k \in [1, K] : \text{BF}[h_k(\text{seq})] = 1 \implies FP))$
Capacity	$P_5 : G\,(\text{Count} \leq \text{MaxCapacity} \wedge (\text{Count} = \text{InsertedSeq.Size}))$
Error Rate	$P_6 : G\left(\text{ErrorRate} = \left(1 - \left(1 - \frac{1}{m}\right)^{K \cdot \text{Count}}\right)^K\right)$
Non-Membership Assurance	$P_7 : \forall \text{seq} \notin \text{InsertedSeq} : G\,(\text{Query}(\text{seq}) \wedge \exists k \in [1, K] : \text{BF}[h_k(\text{seq})] = 0 \rightarrow \text{NotMember})$

Table 6. FM-index Properties

Functionality	**Property**
Index Initialization	$P_1 : G(\text{Initialize}()) \rightarrow \exists(\text{BWT}, \text{Occ Table}) : \text{FM-index is constructed.}$
Backward Search	$P_2 : G(\text{BackwardSearch(P)}) \rightarrow [l, r] : \text{SA interval for } P.$
Rank Query	$P_3 : G(\text{Rank}(c, i)) \rightarrow \lvert\{j \mid \text{BWT}[j] = c, j \leq i\}\rvert.$
Select Query	$P_4 : G(\text{Select}(c, k)) \rightarrow j : \text{BWT}[j] = c, k^{\text{th}} \text{ occurrence.}$
LF-Mapping	$P_5 : G(\text{LF-mapping}(i)) \rightarrow \text{SA}[j], \text{ where } j \text{ is mapped from } i.$
Substring Search	$P_6 : G(\text{FindSubstring}(P)) \rightarrow \{s \mid \text{Ref}[s : s + \lvert P\rvert - 1] = P\}.$
Genome Reconstruction	$P_7 : G(\text{ReconstructGenome}()) \rightarrow \text{Concatenate(BWT, Occ)} = \text{Ref.}$
Exact Match	$P_8 : G(\text{ExactMatchQuery}(P)) \rightarrow \{s \mid \text{Ref}[s : s + \lvert P\rvert - 1] = P\}.$
Approximate Match	$P_9 : G(\text{ApproximateMatch}(P, d)) \rightarrow \{s \mid \text{Hamming}(\text{Ref}[s : s + \lvert P\rvert - 1], P) \leq d\}.$
Range Query	$P_{10} : G(\text{RangeQuery}(l, r)) \rightarrow \{(c, \text{freq}) \mid c \in \Sigma, \text{freq} = \lvert\{j \mid \text{BWT}[j] = c, l \leq j \leq r\}\rvert\}.$
Index Consistency	$P_{11} : G(\text{VerifyIndex}()) \rightarrow \text{BWT, Occ Table, and SA are consistent.}$

Table 7. Wavelet Tree Properties

Functionality	**Property**
Tree Initialization	$P_1 : G\,(\text{Initialize}() \rightarrow \text{Root} \neq \emptyset \wedge \text{TreeHeight} = \lceil \log(\lvert\text{Alphabet}\rvert) \rceil)$
Node Structure	$P_2 : G\,(\forall \text{node } n \in \text{Tree} : \text{NodeBitmap}(n) \text{ is a valid bitvector encoding substrs})$
Rank Operation	$P_3 : \forall c \in \text{Alphabet}, i \in [1, \lvert\text{Seq}\rvert] : G\,(\text{Rank}(c, i) = \text{Number of occurrences of } c \text{ in Sequence}[1 : i])$
Select Operation	$P_4 : \forall c \in \text{Alphabet}, k \in [1, \text{Count}(c)] : G\,(\text{Select}(c, k) = \text{Position of } k\text{-th occurrence of } c \text{ in Sequence})$
Access Operation	$P_5 : \forall i \in [1, \lvert\text{Seq}\rvert] : G\,(\text{Access}(i) = \text{Seq}[i])$
Genome Compression	$P_6 : G\,(\text{CompressedSize(WaveletTree)} \leq \lvert\text{Seq}\rvert \cdot \log(\lvert\text{Alphabet}\rvert))$
Genome Reconstruction	$P_7 : G\,(\text{ReconstructGenome}() \rightarrow \text{Reconstruct all characters from rank/select queries})$
Range Query	$P_8 : \forall [l, r] \subseteq [1, \lvert\text{Seq}\rvert] : G\,(\text{RangeQuery}(l, r) \rightarrow \text{Frequencies of all characters in Seq}[l : r])$
Substring Search	$P_9 : \forall \text{substr} \in \text{Seq} : G\,(\text{Findsubstr(substr)} \rightarrow \text{Return all start positions of subst})$
Balanced Tree Property	$P_{10} : G\,(\text{TreeHeight} = \lceil \log(\lvert\text{Alphabet}\rvert) \rceil \wedge \text{Balanced binary tree structure})$
Error Correction	$P_{11} : G\,(\text{CorrectErrors}() \rightarrow \text{Find and fix erroneous k-mers using rank and select operations})$
Substring Mapping	$P_{12} : \forall \text{subst} \in \text{Seq} : G\,(\text{FindPosition(subst)} \rightarrow \text{Output all start positions of subst in Seq})$

For example, property P_3 ensures correct edge creation between two vertices if their intervals overlap. It guarantees that edges are added correctly based on interval relationships. Similarly, property P_7 ensures correct calculation of how many intervals (vertices) overlap with each position in the genome.

de Bruijn Graphs: In de Bruijn graphs [27], vertices represent k-mers (substrings of length k), and edges represent overlaps of $k-1$ between k-mers. These graphs are widely used in bioinformatics for genome assembly, error correction, and sequence alignment. By breaking sequences into smaller k-mers and connecting them based on overlaps, de Bruijn graphs enable efficient reconstruction of genomes from short reads. Table 9 illustrates the properties generated for verifying the behavior of de Bruijn graph implementations. For example, property P_6 ensures that graph simplification is performed correctly by condensing all non-branching paths—continuous sequences of edges without divergence—into single edges, reducing graph complexity while preserving the original information.

HyperLogLog: HyperLogLog [30] is a probabilistic data structure designed for efficient cardinality estimation, allowing accurate approximations of the number of distinct elements in large datasets with minimal memory usage. It achieves this through the use of hash functions and compact storage of registers, making it ideal for tasks like k-mer counting, duplicate detection, and large-scale data analysis. In bioinformatics, HyperLogLog is particularly useful in applications requiring fast and memory-efficient processing of massive genomic datasets. Table 10 illustrates the properties generated for verifying the behavior of HyperLogLog implementations.

Suffix Trees: Suffix trees [27] provide an efficient way to store and search for all suffixes of a given string. They enable fast substring queries, pattern matching, and finding longest common substrings, making them a cornerstone for many string-processing algorithms. In bioinformatics, suffix trees are widely used in genome alignment, repeat detection, and motif search, offering efficient solutions for large-scale

Table 8. Properties for Interval Graphs

Functionality	**Property**
Graph Initialization	$P_1 : G\,(\text{Initialize}() \rightarrow V = \emptyset \wedge E = \emptyset \wedge \text{Intervals} = \emptyset)$
Vertex Insertion	$P_2 : \forall v \in \text{Intervals} : G\,(\text{InsertVertex}(v) \rightarrow v \in V \wedge v = [l, r] \wedge l < r)$
Edge Construction	$P_3 : \forall v, u \in V : G\,(\text{InsertEdge}(v, u) \rightarrow ([l_v, r_v] \cap [l_u, r_u] \neq \emptyset) \iff (v, u) \in E)$
Overlap Detection	$P_4 : \forall v, u \in V : G\,(\text{Overlap}(v, u) \rightarrow [l_v, r_v] \cap [l_u, r_u] \neq \emptyset)$
Graph Consistency	$P_5 : G\,(\forall v \in V : \text{IntervalsAreValid}(v) \wedge \forall (v, u) \in E : \text{Overlap}(v, u))$
Maximal Overlap Detection	$P_6 : G\,(\text{FindMaxOverlap}() \rightarrow \exists v, u \in V : \text{Maximize}(\lvert[l_v, r_v] \cap [l_u, r_u]\rvert))$
Genome Segment Coverage	$P_7 : G\,(\text{CalculateCoverage}() \rightarrow \forall p \in \text{Genome} : \text{Coverage}(p) = \lvert\{v \in V : p \in [l_v, r_v]\}\rvert)$
Minimum Coloring	$P_8 : G\,(\text{ColorGraph}() \rightarrow \text{Minimize the number of colors } \forall (v, u) \in E : \text{Color}(v) \neq \text{Color}(u))$

Table 9. Properties De Bruijn Graphs

Functionality	**Property**
Graph Initialization	$P_1 : G\,(\text{Initialize}() \rightarrow V = \emptyset \wedge E = \emptyset \wedge \text{Kmers} = \emptyset)$
K-mer Insertion	$P_2 : \forall km \in \text{Kmers} : G\,(\text{InsertKmer}(km) \rightarrow km \in V \wedge k\text{-1 suff/prefix create edges in } E)$
Edge Consistency	$P_3 : \forall u, v \in V : G\,((u, v) \in E \iff \text{Suffix}(u) = \text{Prefix}(v))$
Genome Reconstruction	$P_4 : G\,(\text{ReconstructGenome}() \rightarrow \text{Concatenate all paths covering Kmers to reconstruct Genome})$
Error Correction	$P_5 : G\,(\text{CorrectErrors}() \rightarrow \forall k\text{-mers with low coverage: Remove from graph})$
Graph Simplification	$P_6 : G\,(\text{SimplifyGraph}() \rightarrow \text{Condense all non-branching paths into single edges})$
Loop Detection	$P_7 : G\,(\text{DetectLoops}() \rightarrow \exists v \in V : \text{Path from } v \text{ back to itself exists})$
Contig Construction	$P_8 : G\,(\text{ConstructContigs}() \rightarrow \text{Output all non-branching paths covering unique sequences})$
Graph Coverage	$P_9 : G\,(\text{CalculateCoverage}() \rightarrow \forall k\text{-mers} : \text{Coverage}(k\text{-mer}) = \text{Frequency in original reads})$
Subgraph Extraction	$P_{10} : G\,(\text{ExtractSubgraph}(S) \rightarrow \forall u, v \in S : u, v \in V \wedge (u, v) \in E)$

Table 10. Hyperloglog Properties

Functionality	**Property**
Initialization	$P_1 : G\,(\text{Initialize}() \rightarrow \forall i \in [0, m) : \text{Registers}[i] = 0 \wedge \text{Count} = 0)$
Insertion	$P_2 : \forall \text{seq} \in \text{Seq} : G\,(\text{Insert}(\text{seq}) \rightarrow \text{Registers}[h(\text{seq})] = \max(\text{Registers}[h(\text{seq})], \rho(h(\text{seq}))))$
Estimate Cardinality	$P_3 : G\left(\text{CardinalityEstimate}() = \alpha_m \cdot m^2 \cdot \left(\sum_{i=0}^{m-1} 2^{-\text{Registers}[i]}\right)^{-1}\right)$
Merge Operation	$P_4 : G\,(\text{Merge}(HLL_1, HLL_2) \rightarrow \forall i \in [0, m) : \text{Registers}[i] = \max(\text{Registers}_1[i], \text{Registers}_2[i]))$
Accuracy Bound	$P_5 : G\left(\text{RelativeError} \leq \frac{1.04}{\sqrt{m}}\right)$
Hash Function Validity	$P_6 : \forall \text{seq} \in \text{Seq} : G\,(h(\text{seq}) \in [0, m) \wedge \rho(h(\text{seq}))\text{ is deterministic})$
Union Operation	$P_7 : G\,(\text{Union}(HLL_1, HLL_2) \rightarrow \forall i \in [0, m) : \text{Registers}_{\text{union}}[i] = \max(\text{Registers}_1[i], \text{Registers}_2[i]))$

sequence analysis. Table 11 illustrates the properties generated for verifying the behavior of suffix tree implementations.

Model Checking Techniques

A model checker [16], accepts the design (software) and the property as inputs, and informs if the implementation satisfies the property. If model checking fails, it will produce a counterexample that can be utilized to fix the implementation. When the design is large or the property is complex, the model checker is expected to fail due to state space explosion. There are various approaches in the literature to address the challenges of state space explosion. In this section, we outline three popular approaches that we have utilized in our framework.

SAT-based Bounded Model Checking: In this work, we utilized SAT-based bounded model checking (BMC) for verifying bioinformatics related properties. BMC tries to find a counterexample for a given property within a given bound [17]. For a design D and a property p with a bound of k, BMC encodes the design for satisfiability solving as:

$$BMC(D, p, k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i)$$

Here, the design (D) is unrolled for k cycles, $I(s0)$ denotes the initial state of the design, state transition from s_i to s_{i+1} is represented by $R(s_i, s_{i+1})$ while $p(s_i)$ monitors the property p status during state s_i. The equation will be converted into Conjunctive Normal Form (CNF) and SAT solvers can be used to find a suitable assignment. If CNF finds an assignment that means the p does not hold within the given bound with k cycles, otherwise, it can be concluded that p holds up to k ($D \vDash_k p$).

Design and Property Decompositions: A promising avenue to reduce the property checking complexity is to exploit design and property decompositions. Note that it can lead to false positives unless specific care is taken in bioinformatics applications. For example, consider a property $E(s\&d)$ where the variable s is only relevant for the sequencing module and variable d is only relevant for the variant detection module. We can check s on sequencing module and d on detection module. Unless they are satisfied exactly at the same time in both sequencer and detector, it would be a false positive.

Abstraction and Refinement Techniques: We explored both property and design specific abstraction techniques to reduce the model checking complexity. To take advantage of property-specific abstraction, we utilized conditional slicing [31] as well as counter-example guided abstraction refinement [32, 33]. Conditional slicing refers to a program analysis technique that extracts parts of a program relevant to a given condition,

Table 11. Properties for Suffix Trees

Functionality	**Property**
Tree Initialization	$P_1 : G\,(\text{Initialize}() \rightarrow \text{Root} \neq \emptyset \wedge \text{No edges or nodes except Root})$
Suffix Insertion	$P_2 : \forall \text{suffix} \in \text{Suffixes(Seq)} : G\,(\text{Insert}(\text{suffix}) \rightarrow \exists \text{path from Root to Leaf matching suffix})$
Edge Labels	$P_3 : G\,(\forall \text{node} \in \text{Tree} : \text{Labels on edges from node are unique prefixes of suffixes})$
Substring Query	$P_4 : \forall \text{substr} \in \text{Seq} : G\,(\text{Query}(\text{substr}) \rightarrow \exists \text{path from Root matching substr} \iff \text{substr} \in \text{Seq})$
Lowest Common Ancestor	$P_5 : \forall u, v \in \text{Leaves} : G\,(\text{LCA}(u, v)\text{ represents the longest common prefix of } u \text{ and } v)$
Segment Matching	$P_6 : \forall \text{seg} \in \text{Genome} : G\,(\text{FindSegment}(\text{seg}) \rightarrow \exists \text{path matching seg} \iff \text{seg} \in \text{Genome})$
Suffix Link Consistency	$P_7 : G\,(\forall \text{internal node } n \in \text{Tree} : \text{SuffixLink}(n)\text{ points to node representing suffix of n's string})$
Overlap Detection	$P_8 : G\,(\text{DetectOverlaps}() \rightarrow \forall \text{Seq } A, B : \text{Find max overlap between A and B})$

typically a property to be verified, while omitting unrelated code. Similarly, we utilized design-specific slicing (cone of influence), which extends this concept by leveraging structural and data dependencies in the design, to further reduce the verification complexity.

Theorem Proving of Bioinformatics Software

The theorem proving process begins with a *formal specification* of the bioinformatics algorithm, which is a detailed mathematical description of the algorithm. We conduct a design abstraction to prepare a abstract mathematical description of the algorithm. Then, we utilize this framework to verify properties (theorems) derived using specification. These theorems are usually safety, liveness, or application-specific requirements that the system should satisfy. A *proof assistant* is then utilized, which is a software tool designed to help developers construct and check the validity of logical arguments within the formal specification. The proof assistant aids in the *generation of proof obligations*, which are essentially conditions that need to be proven true for the properties to hold for the given formal specification. Subsequently, the *verification of proof obligations* takes place, wherein each generated obligation must be verified. If all the proof obligations are successfully verified, the system is deemed to have been *verified* and thus meets the defined specifications and properties. However, if any obligation is not verified, it leads to the system being marked as *not verified*. The system's formal specification, properties, or the proofs need to be modified to correct any issues, and the verification process is then repeated. Our approach guarantees correctness through formal, machine-checked proofs.

As shown in Figure 5, our theorem proving framework has three important components: design abstraction, construct theorems, and theorem proving. We use Rosette [34], a formal verification language, to develop the abstract model and to construct theorems. We apply theorem proving to verify two bioinformatics operations used in two applications: input validation for DNA sequencing and maximal exact matches for seeding alignment.

Input Validation of Sequence Data

Each of the software we evaluated was developed to handle large data and thus, cannot easily be reduced to a finite state model, making model checking less feasible due to state explosion. Theorem proving allows for symbolic reasoning, which can prove properties for entire classes of inputs rather than specific instances, providing a more general result. Input Validation is the process of determining the exact order of nucleotides (A, T, C, G) in the input sequences. The example Rosette model in Listing 1 includes two key functions in validating the input: append, which adds a nucleotide to a sequence, and read, which retrieves the sequence while ensuring it is not empty.

Listing 1. Rosette example for appending and reading a sequence.

```
(define (append sequence nucleotide)
  (match nucleotide
    ['A (string-append sequence 'A')]
    ['T (string-append sequence 'T')]
    ['G (string-append sequence 'G')]
    ['C (string-append sequence 'C')]
    ['N (string-append sequence 'N')]))
(define (read sequence)
  (if (not (empty? sequence))
      sequence
      (error 'Sequence is empty')))
```

Listing 2 shows a theorem to check the correctness of appending. The theorem checks that for every sequence and every valid nucleotide, the append operation followed by the read operation results in a sequence that is the original sequence with the nucleotide appended to it. This ensures the operations preserve the integrity of the input and validates that the model behaves correctly for all possible inputs.

Listing 2. Checking correctness of appending for validation of the input.

```
(define (theorem1 sequence nucleotide)
  (equal? (read (append sequence nucleotide))
          (string-append sequence nucleotide)))
(assert (for/all ([sequence (in-list '( 'A' 'T' 'C'
    'G' 'N' 'ATCGN'))])
                ([nucleotide (in-list '( 'A' 'T' 'C'
                      'G' 'N'))])
          (theorem1 sequence nucleotide)))
```

Finding Maximal Exact Matches

Maximal Exact Matches (MEMs) are frequently used to find seeds for read alignment. MEMs are exact alignments between a read and the reference genome that cannot be extended on the left or right. Here, we want to guarantee that all MEMs between a read and a reference genome are identified. The Succinct Data Structures Library (SDSL) [4] provides algorithms for MEM finding. Here, we model it on Rosette. Specifically, we need to check $G(matchFound \rightarrow (isMaximal \wedge \neg canExtendLeft \wedge \neg canExtendRight))$, where: $isMaximal$ is defined as $\forall i, j : (i < \text{start}(M) \vee j > \text{end}(M)) \rightarrow \neg\text{ValidMatch}(i, j)$. Similarly, $canExtendLeft$ is defined as $\exists c \in \Sigma$: Reference$[\text{start}(M) - 1] = c \wedge \text{Read}[k] = c$. Likewise, $canExtendRight$ is defined as $\exists c \in \Sigma$: Reference$[\text{end}(M) + 1] = c \wedge \text{Read}[k] = c$. The variable c denotes a nucleotide, and the index k maps positions in the read sequence to positions in the reference genome. Listing 3 formalizes this intuition as a theorem in Rosette, verifying the correctness by ensuring all MEMs are identified and cannot be extended to the left or right.

Listing 3. Theorem in Rosette for proving correctness of MEM finding.

```
(define (is-maximal match reference read)
  ;; A match is maximal if no valid match exists
      outside its bounds
  (forall ([i (in-range (length reference))]
           [j (in-range (length reference))])
    (implies (or (< i (match-start match)) (> j (
        match-end match)))
             (not (valid-match i j reference read)))
             ))

(define (can-extend-left match reference read)
  ;; A match cannot be extended to the left
  (exists ([c (in-list '(A T C G N))])
    (and (= (substring reference (- (match-start
        match) 1) (match-start match)) c)
         (= (substring read 0 1) c))))

(define (can-extend-right match reference read)
  ;; A match cannot be extended to the right
  (exists ([c (in-list '(A T C G N))])
    (and (= (substring reference (match-end match)
        (+ (match-end match) 1)) c)
         (= (substring read (- (length read) 1)) c))
             ))
```

Table 12. Static Analysis of BiopLib [1] and BWA [2] using CBMC [35]

Function	# of Total Properties	# of Failed Properties	Line Coverage	Potential Flaws Detected
blVecDist [1]	53	28	100%	Pointer Arithmetic, Overflow
blFreePDBStruct [1]	45	0	100%	-
re_split [2]	190	7	100%	Overflow
bwa_gen_cigar2 [2]	425	164	89%	Divide by Zero, Overflow, Pointer Arithmetic

```
(define seeding-alignment-theorem
  (assert
    ;; A match is correct if it is found and
        satisfies all three properties
    (forall ([match match-type?]
             [reference string?]
             [read string?])
      (implies (match-found match)
               (and (is-maximal match reference read
                    )
                    (not (can-extend-left match
                         reference read))
                    (not (can-extend-right match
                         reference read)))))))
```

The details of the model and the additional theorems for validating the input, and finding MEMs, along with their respective results, are discussed in the following section.

Experiments

To establish the feasibility of the proposed formal verification framework, we have done experiments for both model checking and theorem proving. We have developed three types of experiments. (1) We perform static analysis of bioinformatics software to detect typical software bugs and vulnerabilities. (2) We provide initial results for model checking of bioinformatics software, including BiopLib [1], BWA [2], Jellyfish [3], SDSL [4], Dashing [5], SPAdes [6], and MUMmer [7]. (3) We also discuss theorem proving results for two bioinformatics applications, including validating the input, and MEM finding.

Experimental Setup

Bioinformatics applications often involve the analysis of complex biological data through multiple stages such as read alignment and variant detection. Methods like BiopLib and BWA are used for these processes, where correctness is essential for reliable results. BiopLib [1] is a reliable PDB parser that handles alternate occupancies and deals with compressed PDB files and PDBML files. BWA [2] is a program for aligning reads against a reference genome. For checking implementation issues in these two libraries, we used CBMC [35], a bounded model checker for C programs which uses static analysis and bounded model checking to test potential bugs and vulnerabilities in a design. RapidCheck [36] is a property-based testing library for C++ that can be used to perform property-based verification. We have applied RapidCheck to perform model checking in five popular bioinformatics software libraries (BWA [2], Jellyfish [3], Dashing [5], SPAdes [6], and MUMmer [7]). To model algorithms, theorems, and conduct theorem proving, we use the Rosette [34] formal verification language. The Rosette model has assertions, symbolic variables, and solver-aided functions. The correctness of these elements is verified using Rosette's symbolic execution engine, which uses Z3 [37] SMT solver.

Static Analysis Results

We conduct static analysis to detect generic vulnerabilities in bioinformatics software. For this, we utilize CBMC [35] model checker for bounded model checking, which explores the state space to ensure properties hold under various conditions. We found several potential flaws in the two methods (BiopLib [1] and BWA [2]), as shown in Table 12. Consider the function *get_seq* in Figure 6, it can lead to memory allocation error for a specific input. For example consider *l_pac* = 2, *beg* = 7 and *end* = 8. For this input, line 4 will get activated because *l_pac*<<1 is 4 and *end* is greater than 4. Then *end* will get the value 4, which is less than *beg*. Since the *beg* value satisfy the condition in line 5, the program will execute the if path and line 8 will cause a memory allocation program failure due to the negative value resulted by (*end* - *beg*).

```
1 void *get_seq(int64_t l_pac,int64_t beg,int64_t end){
2  uint8_t *seq = 0;
3  if (end < beg) end ^= beg, beg ^= end, end ^= beg;
4  if (end > l_pac<<1) end = l_pac<<1;
5  if (beg >= l_pac || end <= l_pac) {
6     int64_t k, l = 0;
7     *len = end - beg;
8     seq = malloc(end - beg); ...
```

Fig. 6. An example code block in BWA [2]

Model Checking Results

For model checking, we have constructed properties for basic data structures (radix trees, bloom filters, hyperloglog, interval graphs, suffix trees, de Brujin graphs, wavelet trees and FM indexes) as outlined in Table 4-11. Table 13 shows the model checking results for verifying five bioinformatics methods (BWA, Jellyfish, Mash, SPAdes, and MUMmer). In this setup, first we used the Model Checking to find violations, and once the tool returned a counter-example that violates the property, we tested the property with 100 mutations to the counter-example to exactly identify the failure condition. For example, one of the main data structures in SDSL is the wavelet tree to support efficient `rank` and `select` operations. Our model checking targeted these underlying data structures and showed that for the `select` operation to behave correctly, the j-th occurrence of a character must exist, i.e., the character must appear at least j times in the sequence. Otherwise, the system will crash due to a failure of the assertion. Since users may not know how many times any item is repeated, it would be better to return a NULL object if an occurrence is not detected without crashing the system. Our proposed framework was able to identify this scenario in SDSL with the evaluation of the property P_4 corresponding to the wavelet trees in Table 13.

Theorem Proving Results

Table 14 summarizes the theorem proving results for validating the input as well as finding MEMs.

Our abstract model for validating the input, implemented in Rosette, consists of 178 lines of code and covers key functionalities, including appending nucleotides, reading sequences, generating complementary and reverse-complementary strands, calculating alignment scores, detecting mutations, assembling fragments, calculating GC content, generating k-mers, trimming sequences, and performing quality control. We have constructed 10 theorems using 154 lines of Rosette code. We briefly describe these theorems as follows:

Appending Theorem: This theorem ensures that appending a valid nucleotide (`A`, `T`, `G`, `C`) to an existing DNA sequence increases its length by one. This validates that the operation preserves the integrity of the sequence while expanding it as expected.

Reading Theorem: This theorem guarantees that reading a non-empty DNA sequence always returns the sequence itself, ensuring no alteration occurs during the read operation. It also confirms that attempting to read an empty sequence appropriately raises an error.

Complement Theorem: The complement theorem verifies that the generation of the complement of a DNA sequence adheres to the base-pairing rules, where `A` $\leftrightarrow$ `T` and `G` $\leftrightarrow$ `C`. This ensures the correctness of the complementary strand generation process.

Reverse Complement Theorem: This theorem validates that the reverse complement of a DNA sequence is equivalent to reversing the sequence and then generating its complement. It confirms the correctness of this essential operation in bioinformatics.

Alignment Theorem: The alignment theorem ensures that the alignment score between two identical sequences is equal to their length (assuming a match score of $+1$). This validates the correctness of the scoring mechanism for identical sequences.

Mutation Detection Theorem: This theorem guarantees that mutations are detected correctly by identifying positions where the reference and the input sequence differ. It ensures no false positives or missed mutations in the detection process.

GC Content Theorem: GC content refers to the percentage of the DNA sequence that is composed of the nitrogenous bases guanine (G) and cytosine (C). The GC content theorem confirms that the GC content of a sequence is correctly calculated as the percentage of `G` and `C` bases and always falls within valid bounds (0–100%).

Trimming Theorem: This theorem validates that trimming a sequence reduces its length by removing specified regions (e.g. adapters or low-quality bases) from the ends while ensuring no invalid bases are introduced.

k-mer Theorem: The k-mer theorem ensures that all k-mers (substrings of a specified length k) generated from a DNA sequence are valid substrings of the original sequence and have the correct length.

Fragment Assembly Theorem: This theorem guarantees that assembling multiple fragments into a single sequence produces a result equivalent to concatenating the fragments in order. It ensures the correctness of the fragment assembly.

Our abstract model for the seeding alignment operation, implemented in Rosette, consists of 36 lines of code and captures key functionalities such as match detection, validation, and maximality checks. The Seeding Alignment Theorem, defined using 25 lines of Rosette assertions, ensures the correctness of the operation by validating three critical properties: (1) the match is maximal, meaning no valid match exists outside its bounds, (2) the match cannot be extended to the left with a valid nucleotide, and (3) the match cannot be extended to the right with a valid nucleotide. This comprehensive theorem guarantees the integrity and accuracy of the seeding alignment.

Discussion

The vulnerabilities and errors that we identified can have significant effects on downstream analysis. For example, given the vulnerability found in BWA, if triggered during large-scale genome alignment tasks, critical runtime errors, such as segmentation faults or out-of-bounds memory access, can occur. Specifically, one of the flaws we identified in the *get_seq* function (Figure 6) could cause BWA to request a negative number of bytes during memory allocation when presented with certain read intervals. On some HPC systems, this may result in a crash; however, depending on the memory allocation and execution context, it may also lead to undefined behavior, such as accessing memory outside the allocated bounds. In automated sequencing workflows, this could cause entire read batches to be silently skipped or misaligned without triggering explicit errors, undermining the reliability of alignment results.

The consequences of such failures are concrete and serious. For example, if BWA is used as the first step in a variant calling pipeline (e.g., GATK Best Practices[1]), misaligned or omitted reads can result in the loss of true variants or the introduction of false positives. In metagenomics, where read depth and alignment coverage directly influence abundance estimation and species identification, this could distort community profiles or lead to incorrect taxonomic assignments [38]. This vulnerability

[1] https://gatk.broadinstitute.org/hc/en-us/sections/360007226651-Best-Practices-Workflows

Table 13. Model checking results for five libraries

Library	Properties	Time (s)
BWA [2]	Wavelet Tree: $[P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9, P_{10}, P_{11}, P_{12}]$	5.88
	FM-index: $[P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9, P_{10}, P_{11}]$	6.34
Jellyfish [3]	Radix Trees: $[P_1, P_2, P_3, P_4, P_5]$	3.18
	Bloom-filter: $[P_1, P_2, P_3, P_4, P_5, P_6, P_7]$	2.77
Dashing [5]	Hyperloglog: $[P_1, P_2, P_3, P_4, P_5, P_6, P_7]$	2.70
	Bloom-filter: $[P_1, P_2, P_3, P_6, P_7]$	2.36
SPAdes [6]	De Bruijn Graph: $[P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9, P_{10}]$	3.92
	Bloom-filter: $[P_1, P_2, P_3, P_6, P_7]$	2.36
	Interval Graph: $[P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8]$	4.17
MUMmer [7]	Suffix Tree: $[P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8]$	4.12

Table 14. Theorem proving results for validating the input as well as finding MEMs.

Operation	Number of lines of code in design	Number of lines of code in theorems	Verification time (seconds)
Validating Input	178	154	116
Finding MEMs	36	25	54

is especially impactful given the widespread use of BWA (cited in over 50,000 studies acc. to Google Scholar) and integrated into pipelines for population-scale datasets, such as the 1000 Genomes Project [39], TCGA [40, 41, 8], and routine hospital sequencing efforts. Our findings demonstrate the importance of proactive static analysis: Such flaws are unlikely to be discovered through empirical testing, which rarely covers boundary conditions or malformed inputs.

Our results on model checking demonstrate potential system crashes due to an assertion failure in SDSL. In particular, the wavelet tree is used in a number of bioinformatics software [42, 43, 44, 45, 46, 47, 48]. This failure is particularly problematic in practice because users cannot easily ensure that such conditions are satisfied. The ability of our framework to uncover this issue demonstrates its potential to detect subtle, user-facing bugs that could compromise downstream analyses without explicit warning.

While static analysis and model checking can uncover many of these issues, theorem proving offers a complementary and increasingly essential capability in bioinformatics, particularly for verifying algorithmic logic in alignment operations. For instance, our theorem proving experiments demonstrate how correctness properties for operations like reverse complementation, k-mer generation, and maximal exact match (MEM) detection can be encoded and formally validated using symbolic execution and satisfiability modulo theories (SMT) solvers. In the case of MEM finding, a key step in many bioinformatics software methods [49, 50], we verify not only the presence of the match but also its maximality and boundary conditions, which are critical for ensuring downstream consistency.

Importantly, the use of theorem proving provides guarantees beyond what is possible through empirical testing or run-time assertions. It enables developers to establish soundness conditions for biological invariants, such as base-pairing rules or the structure of alignment gaps, and ensure that these hold across all possible inputs. This is particularly valuable for alignment tools, where incorrect behavior may only be triggered by rare or malformed reads, and where silent failures can propagate downstream. By encoding these invariants as logical assertions, theorem proving allows developers to eliminate entire classes of subtle bugs at design time.

Limitations and future research directions: Formal verification generally operates in two distinct modes: verifying explicit algorithmic logic similar to this work and verifying machine learning-based prediction models. The latter is a fundamentally different domain known as verifiable AI [51]. Our current framework targets conventional software components with well-defined specifications. In future work, we plan to adopt concepts from verifiable AI to extend our framework for verifying modern bioinformatics applications that leverage deep learning. We also plan to collaborate with clinical laboratories to integrate our framework into real-world genomic analysis workflows, thus improving reliability, reproducibility, and regulatory compliance in clinical settings.

Key Points

- In this study, we proposed a comprehensive framework for the formal verification of bioinformatics software, demonstrating its feasibility and effectiveness through static analysis, model checking, and theorem proving.
- By applying these methods to popular bioinformatics libraries including BiopLib, BWA, Jellyfish, SDSL, Dashing, SPAdes, and MUMmer, we identified potential flaws and validated the robustness of data structures, reinforcing the need for rigorous software assurance in bioinformatics.
- This is the first study of comprehensive formal verification for bioinformatics software. Our findings highlight the importance of moving beyond ad-hoc testing and embracing formal verification to ensure reliable results in bioinformatics applications.
- Advancing formal verification techniques for bioinformatics software boosts trustworthiness and accelerates biomedical progress.

References

1. Craig T Porter and Andrew CR Martin. BiopLib and BiopTools—a C programming library and toolset for manipulating protein structure. *Bioinformatics*, 31(24):4017–4019, 2015.
2. H. Li and R. Durbin. Fast and accurate short read alignment with Burrows–Wheeler Transform. *Bioinformatics*, 25(14):1754–1760, 2009.
3. Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.
4. Succinct Data Structure Library (SDSL). https://github.com/simongog/sdsl-lite.
5. Daniel N Baker and Ben Langmead. Dashing: fast and accurate genomic distances with hyperloglog. *Genome Biology*, 20:1–12, 2019.
6. Andrey Prjibelski, Dmitry Antipov, Dmitry Meleshko, Alla Lapidus, and Anton Korobeynikov. Using SPAdes de novo assembler. *Current Protocols in Bioinformatics*, 70(1):e102, 2020.
7. Guillaume Marçais, Arthur L Delcher, Adam M Phillippy, Rachel Coston, Steven L Salzberg, and Aleksey Zimin. MUMmer4: A fast and versatile genome alignment system. *PLoS Computational Biology*, 14(1):e1005944, 2018.
8. Abraham Gihawi, Yuchen Ge, Jennifer Lu, Daniela Puiu, Amanda Xu, Colin S Cooper, Daniel S Brewer, Mihaela Pertea, and Steven L Salzberg. Major data analysis errors invalidate cancer microbiome findings. *MBio*, 14(5):e01607–23, 2023.
9. Tsong Yueh Chen, Joshua WK Ho, Huai Liu, and Xiaoyuan Xie. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC Bioinformatics*, 10(1):1–12, 2009.

10. Eleni Giannoulatou, Shin-Ho Park, D T Humphreys, and J WK Ho. Verification and validation of bioinformatics software without a gold standard: a case study of BWA and Bowtie. *BMC Bioinformatics*, 15(16):1–8, 2014.
11. Aedin Pereira, Julia Ding, Zaina Ali, and Rodion Podorozhny. Verification of distributed artificial intelligence systems in bioinformatics. *arXiv preprint arXiv:2302.04389*, 2023.
12. Mehmet Emin Bakir, Savas Konur, Marian Gheorghe, Natalio Krasnogor, and Mike Stannett. Automatic selection of verification tools for efficient analysis of biochemical models. *Bioinformatics*, 34(18):3187–3195, 2018.
13. Bert Bogaerts, Raf Winand, Qiang Fu, Julien Van Braekel, Pieter-Jan Ceyssens, Wesley Mattheus, Sophie Bertrand, Sigrid CJ De Keersmaecker, Nancy HC Roosens, and Kevin Vanneste. Validation of a bioinformatics workflow for routine analysis of whole-genome sequencing data and related challenges for pathogen typing in a european national reference center: Neisseria meningitidis as a proof-of-concept. *Frontiers in Microbiology*, 10:362, 2019.
14. Jeffrey A SoRelle, Megan Wachsmann, and Brandi L Cantarel. Assembling and validating bioinformatic pipelines for next-generation sequencing clinical assays. *Archives of Pathology & Laboratory Medicine*, 144(9):1118–1130, 2020.
15. Xiang Jin, Zhiyong Ji, Xiaodan Li, Rui Liu, and Yinghui Guan. Bioinformatics analysis and verification of key genes associated with recurrent respiratory tract infections. *International Journal of Molecular Medicine*, 42(1):514–524, 2018.
16. Edmund M Clarke. Model checking. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 54–56. Springer, 1997.
17. Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
18. Mingsong Chen and Prabhat Mishra. Property learning techniques for efficient generation of directed tests. *IEEE Transactions on Computers*, 60(6):852–864, 2011.
19. Mordechai Ben-Ari, Amir Pnueli, and Zohar Manna. The temporal logic of branching time. *Acta Informatica*, 20(3):207–226, 1983.
20. E.Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241 – 266, 1982.
21. Antti Pakonen, Cheng Pang, Igor Buzhinsky, and Valeriy Vyatkin. User-friendly formal specification languages-conclusions drawn from industrial experience on model checking. In *Proceedings of the IEEE 21st International Conference on Emerging Technologies and Factory Automation*, pages 1–8, 2016.
22. Amélie David, Francois Laroussinie, and Nicolas Markey. On the Expressiveness of QCTL. In *27th International Conference on Concurrency Theory (CONCUR 2016)*, volume 59 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:15, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
23. Andrian Yang, Michael Troup, and Joshua WK Ho. Scalability and validation of big data bioinformatics software. *Computational and Structural Biotechnology*, 15:379–386, 2017.
24. Somak Roy, Christopher Coldren, Arivarasan Karunamurthy, Nefize S Kip, Eric W Klee, Stephen E Lincoln, Annette Leon, Mrudula Pullambhatla, Robyn L Temple-Smolkin, Karl V Voelkerding, Chen Wang, and Alexis Carter. Standards and guidelines for validating next-generation sequencing bioinformatics pipelines: a joint recommendation of the association for molecular pathology and the college of american pathologists. *The Journal of Molecular Diagnostics*, 20(1):4–27, 2018.
25. Hillel Kugler. Formal verification for natural and engineered biological systems. In *Proceedings of Formal Methods in Computer Aided Design (FMCAD)*, pages 1–1, 2020.
26. Christopher I Cooper, Delia Yao, Dorota H Sendorek, Takafumi N Yamaguchi, Christine P'ng, Kathleen E Houlahan, Cristian Caloian, Michael Fraser, SMC-DNA Challenge Participants, Kyle Ellrott, Adam Margolin, Robert Bristow, Joshua Stuart, and Paul Boutros. Valection: design optimization for validation and verification studies. *BMC Bioinformatics*, 19:1–11, 2018.
27. Camille Marchet, Christina Boucher, Simon J Puglisi, Paul Medvedev, Mikaël Salson, and Rayan Chikhi. Data structures based on k-mers for querying large collections of sequencing data sets. *Genome Research*, 31(1):1–12, 2021.
28. Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 390–398. IEEE, 2000.
29. Giovanni Manzini. An analysis of the Burrows—Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
30. Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Proceedings of the Conference on Analysis of Algorithms (AoA)*, page 1365–8050. Discrete Mathematics and Theoretical Computer Science, 2007.
31. Shobha Vasudevan, E Allen Emerson, and Jacob A Abraham. Efficient model checking of hardware using conditioned slicing. *Electronic Notes in Theoretical Computer Science*, 128(6):279–294, 2005.
32. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV)*, pages 154–169. Springer, 2000.
33. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.
34. Rosette Language Guide. https://docs.racket-lang.org/rosette-guide/index.html.
35. Daniel Kroening and Michael Tautschnig. CBMC–C Bounded Model Checker: (Competition Contribution). In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 389–391. Springer, 2014.
36. RapidCheck. https://github.com/emil-e/rapidcheck.
37. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340. Springer, 2008.
38. Jonathan E Bravo, Ilya Slizovskiy, Nathalie Bonin, Marco Oliva, Noelle Noyes, and Christina Boucher. The

TELCoMB Protocol for High-Sensitivity Detection of ARG-MGE Colocalizations in Complex Microbial Communities. *Current Protocols*, 4(10):e70031, 2024.

39. Xiangqun Zheng-Bradley, Ian Streeter, Susan Fairley, David Richardson, Laura Clarke, Paul Flicek, and 1000 Genomes Project Consortium. Alignment of 1000 Genomes Project reads to reference assembly GRCh38. *GigaScience*, 6(7):gix038, 2017.
40. Alexandra R Buckley, Kristopher A Standish, Kunal Bhutani, Trey Ideker, Roger S Lasken, Hannah Carter, Olivier Harismendy, and Nicholas J Schork. Pan-cancer analysis reveals technical artifacts in TCGA germline variant calls. *BMC Genomics*, 18:1–15, 2017.
41. Alexandra Anke Baumann, Olaf Wolkenhauer, and Markus Wolfien. TCGADownloadHelper: simplifying TCGA data extraction and preprocessing. *Frontiers in Genetics*, 16:1569290, 2025.
42. Jarno N Alanko, Elena Biagi, Simon J Puglisi, and Jaakko Vuohtoniemi. Subset wavelet trees. In *Proceedings of the 21st International Symposium on Experimental Algorithms (SEA)*, pages 4–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2023.
43. Z Iqbal, S Maciuca, C del Ojo Elias, and G McVean. A natural encoding of genetic variation in a burrows-wheeler transform to enable mapping and genome inference. In *Proceedings of the Workshop on Algorithms in Bioinformatics (WABI)*. Springer, 2016.
44. Sanjeev Kumar, Suneeta Agarwal, and Ranvijay. Fast and memory efficient approach for mapping NGS reads to a reference genome. *Journal of Bioinformatics and Computational Biology*, 17(02):1950008, 2019.
45. Martin D Muggli, Simon J Puglisi, and Christina Boucher. A succinct solution to rmap alignment. In *Proceedings of the 18th International Workshop on Algorithms in Bioinformatics*, pages 12–1. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2018.
46. Christina Boucher, Alex Bowe, Travis Gagie, Simon J Puglisi, and Kunihiko Sadakane. Variable-order de Bruijn graphs. In *Proceedings of the Data Compression Conference (DCC)*, pages 383–392. IEEE, 2015.
47. Martin D Muggli, Simon J Puglisi, and Christina Boucher. Efficient indexed alignment of contigs to optical maps. In *Proceedings of the 14th International Workshop Algorithms in Bioinformatics (WABI)*, pages 68–81. Springer, 2014.
48. Martin D Muggli, Simon J Puglisi, and Christina Boucher. Kohdista: an efficient method to index and query possible Rmap alignments. *Algorithms for Molecular Biology*, 14:1–13, 2019.
49. Massimiliano Rossi, Marco Oliva, Paola Bonizzoni, Ben Langmead, Travis Gagie, and Christina Boucher. Finding maximal exact matches using the r-index. *Journal of Computational Biology*, 29(2):188–194, 2022.
50. Heng Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv preprint arXiv:1303.3997*, 2013.
51. Changliu Liu, Tomer Arnon, Christopher Lazarus, Christopher Strong, Clark Barrett, Mykel J Kochenderfer, et al. Algorithms for verifying deep neural networks. *Foundations and Trends® in Optimization*, 4(3-4):244–404, 2021.

Figure Legends

Figure 1. Overview of model checking.
Alt Text: A block diagram showing bioinformatics software feeding into model and property blocks, which are inputs to a model checking block. A counterexample output feeds back into the software.

Figure 2. Overview of theorem proving.
Alt Text: A block diagram where bioinformatics software produces theorems and an abstract model, which go into a theorem proving block. Unproven obligations are fed back to the software.

Figure 3. State diagram of two example temporal operators in LTL.
Alt Text: Two linear state diagrams with nodes labeled 'p' and 'q'. (a) shows states satisfying eventuality operator Fp. (b) shows a sequence satisfying until operator p U q.

Figure 4. State diagram of two example temporal operators in CTL.
Alt Text: Two branching state trees. (a) shows the CTL operator AFp with paths eventually leading to a state satisfying p. (b) shows EGp with all paths remaining in states satisfying p.

Figure 5. Overview of the proposed formal verification framework that consists of model checking and theorem proving.
Alt Text: A complex architecture diagram showing static analysis feeding into a central bioinformatics software block, with connections to design decomposition, model checking, theorem proving, and various behavior categories like insertion, deletion, and initialization.

Figure 6. An example code block in BWA.
Alt Text: A syntax-highlighted C function snippet from BWA showing pointer manipulation and conditionals for sequence allocation using malloc.