MDPI

Article

Model Checking-Based Malware Detection

Carlo Colizzi

[1] Department of Computer Science, University of Salerno, 84084 Fisciano, Italy; c.colizzi@studenti.unisa.it
* carlo.colizzi@gmail.com

1. Introduction

In recent years, the increasing spread of malware has made it necessary to develop advanced detection techniques. Among these, Model Checking-Based Malware Detection stands out for its formal and rigorous approach to analyzing program behavior. This technique relies on the automated verification of specific properties within a software model, enabling the identification of malicious behaviors with a high degree of precision.

This report will analyze the fundamental principles of Model Checking applied to malware detection, its main methodologies, and the advantages over traditional detection techniques based on signatures or heuristics. Finally, some case studies and the potential evolutions of this technology in countering cyber threats will be discussed.

2. Fundamentals of Model Checking

Model Checking is an automated verification technique used to analyze the behavior of a system and ensure that it satisfies formally specified properties. This approach is widely applied in the verification of software and hardware systems to guarantee their correctness against defined requirements.

2.1. Formalization of Model Checking

Model Checking is based on representing the system as a *state transition model*, typically a *Finite State System* (FSS) or a *stochastic process*. The properties to be verified are expressed using temporal logics such as:

- **LTL (Linear Temporal Logic)**: Describes properties along single execution paths of the system, following a sequential approach.
- **CTL (Computation Tree Logic)**: Allows the expression of properties over computation trees, considering multiple possible system evolutions.

Verification is performed by an algorithm that explores the system's state space and checks whether the specified properties hold. If a violation is found, the Model Checker provides a *counterexample*, a sequence of states that demonstrates the failure to meet the specification.

2.2. Applications and Limitations

Model Checking is applied in various domains, including:

- **Verification of communication protocols** to ensure security and reliability.
- **Analysis of critical software** to detect bugs and vulnerabilities.
- **Malware detection**, where it is used to identify malicious behaviors by analyzing the program's interactions with the operating system.

Despite its advantages, Model Checking has some limitations, such as:

- **State explosion problem**, which makes it difficult to analyze complex systems.

Received:
Accepted:
Published:

Citation: . Model Checking-Based Malware Detection. *Journal Not Specified* , *1*, 0.

- **High computational cost**, due to the necessity of exploring all possible system executions.

In recent years, several optimization techniques, such as *symbolic verification* and *abstraction refinement*, have been developed to mitigate these challenges and extend the applicability of Model Checking to large-scale systems.

3. Fundamentals of Malware Detection

Malware detection is the process of identifying and analyzing malicious software to prevent it from compromising the security of computer systems. Modern malware employs sophisticated evasion techniques, making its detection a challenging task. Various detection methods have been developed over the years, ranging from traditional signature-based approaches to more advanced behavior-based and AI-driven techniques.

3.1. Malware Detection Approaches

Malware detection methods can be broadly categorized into the following approaches:

- **Signature-Based Detection**: Compares files against a database of known malware signatures. It is fast and effective against known threats but fails to detect new or obfuscated malware.
- **Heuristic-Based Detection**: Uses rule-based techniques and machine learning algorithms to identify potentially malicious behaviors. This approach can detect unknown malware but may produce false positives.
- **Behavior-Based Detection**: Analyzes the execution behavior of programs to identify malicious activities. This method is effective against polymorphic and metamorphic malware but can be computationally expensive.
- **Model Checking-Based Detection**: Applies formal verification techniques to analyze the system's state transitions and detect deviations from expected behaviors. While precise, this approach faces scalability issues.
- **Cloud-Based Detection**: Offloads malware analysis to cloud-based services, providing large-scale detection capabilities with minimal resource consumption on client devices.
- **Deep Learning-Based Detection**: Utilizes neural networks and AI models to classify malware based on extracted features. This method improves detection accuracy but is vulnerable to adversarial attacks.

3.2. Challenges in Malware Detection

Despite advancements in malware detection techniques, several challenges remain:

- **Evasion Techniques**: Malware authors use obfuscation, polymorphism, and metamorphism to bypass traditional detection methods.
- **Zero-Day Attacks**: Newly developed malware may not match any existing signature or known behavior pattern, making detection difficult.
- **False Positives and False Negatives**: Balancing detection accuracy while minimizing incorrect classifications remains a significant issue.
- **Performance Overhead**: Some detection methods, especially behavior-based and model-checking techniques, require extensive computational resources.

3.3. Future Directions

Future research in malware detection aims to improve accuracy and efficiency while reducing resource consumption. Key areas of development include:

- **Hybrid Detection Methods**: Combining multiple approaches, such as heuristic and behavior-based detection, to enhance accuracy.

- **AI and Machine Learning Enhancements**: Improving deep learning models to reduce susceptibility to adversarial attacks.
- **Cloud-Integrated Security Solutions**: Expanding cloud-based malware analysis to provide real-time detection with minimal impact on local systems.
- **Formal Methods for Security Verification**: Extending model checking techniques to handle complex and large-scale systems effectively.

Malware detection remains an evolving field, requiring continuous advancements to stay ahead of emerging threats.

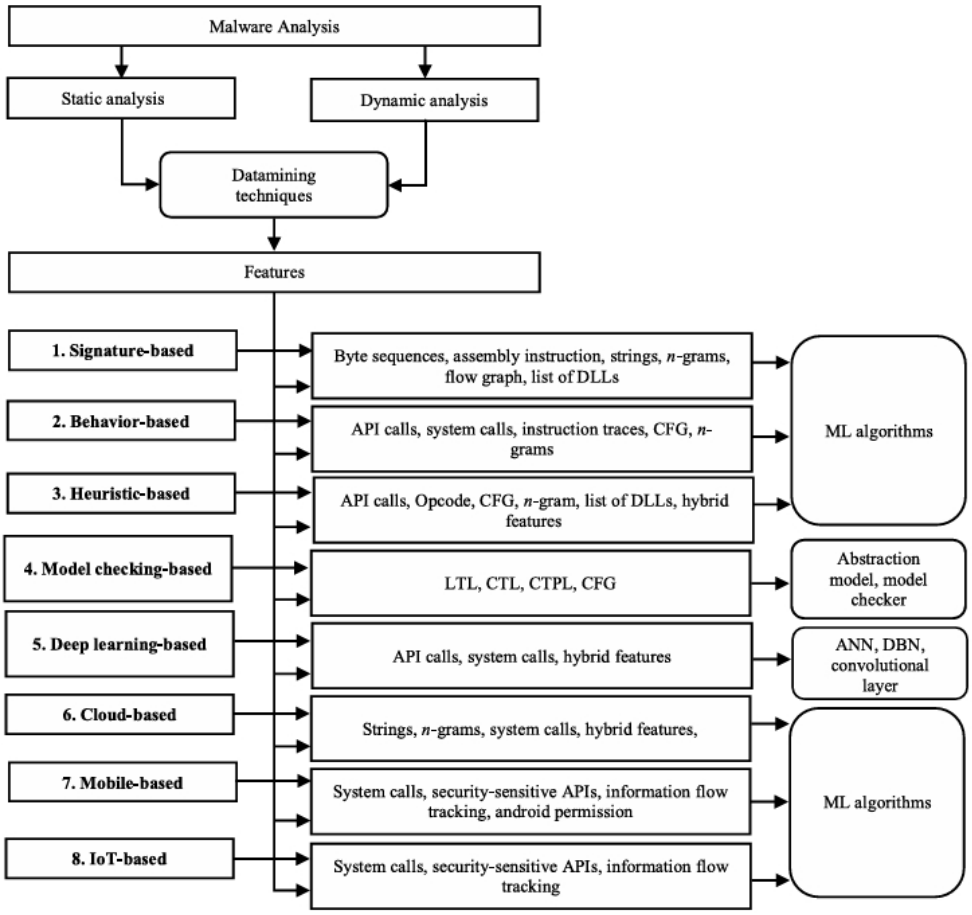

Figure 1. Malware Detection Approaches

4. Model Checking-Based Malware Detection intro

Although model checking is originally developed to verify the correctness of system against specifications, it has been used to detect malware as well. In this detection approach, malware behaviors are manually extracted and behavior groups are coded using temporal logic to display a specific feature. Program behaviors are created by looking at the flow relationship of one or more system calls and define behaviors by using properties such as hiding, spreading, and injecting. By comparing these behaviors, it is determined whether the program is malware or benign. Model checking-based detection can detect some new malware to a certain degree, but cannot detect all new generation of malware.

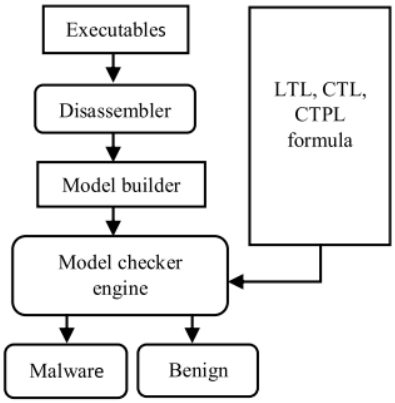

Figure 2. Model checking-based malware detection schema.

5. Analysing Paper: Detecting Malicious Code by Model Checking

This research presents a novel approach for detecting malicious code in CFG using *model checking*. The authors introduce the **Computation Tree Predicate Logic (CTPL)**, an extension of CTL, to define malicious code patterns in a flexible and efficient manner. The detection process consists of three main steps:

1. extracting the control flow graph (CFG) from the binary,
2. specifying malicious behavior using CTPL formulas,
3. verifying the presence of malicious patterns using a model checking algorithm.

5.1. Control Flow Graph Extraction

A potentially infected executable is disassembled and transformed into a *Kripke structure*, a finite-state representation where:

- Each node corresponds to an assembler instruction.
- Edges represent possible execution paths.
- Labels are assigned to nodes, capturing instruction semantics (e.g., register assignments, memory access).

Conditional branches create multiple execution paths, while procedure calls can be inlined or handled separately.

5.2. Computation Tree Predicate Logic (CTPL)

CTPL extends CTL by introducing *predicate logic* to handle variable bindings dynamically. The key features include:

- **Quantified predicates:** allowing expressions such as $\exists r\, \mathrm{EF}(\mathrm{mov}(r, 937) \wedge \mathrm{AF}(\mathrm{push}(r)))$, meaning that some register r first receives value 937 and is later pushed onto the stack.
- **State-based modeling:** The program's behavior is expressed in terms of state transitions in a Kripke structure.
- **Flexible malware specification:** Instead of using fixed signatures, CTPL defines patterns based on control flow and data dependencies, making it robust against obfuscation.

5.3. Model Checking Process

Given an executable's Kripke structure and a CTPL specification, the model checker applies a dynamic programming approach to verify whether the specification holds. The algorithm:

1. Traverses the Kripke structure and labels states with atomic propositions from CTPL formulas.
2. Computes variable bindings dynamically, propagating constraints through the execution paths.
3. Determines if a malicious code pattern exists by checking satisfiability of CTPL formulas over the Kripke structure.

5.4. Experimental Results

The prototype was tested on multiple worm families, including *NetSky*, *MyDoom*, and *Klez*. A single CTPL formula successfully detected several variants of each worm, demonstrating:

- **Generality:** One formula matched multiple malware variants.
- **Robustness:** The approach overcame common obfuscation techniques.
- **Efficiency:** Checking an executable with 150 assembler instructions took approximately 2 seconds on an Athlon XP 2600+ with 512MB RAM.

5.5. Conclusion

Model checking with CTPL provides an effective method for detecting malware and its derivatives. Unlike signature-based detection, this technique generalizes across similar threats and remains effective against syntactic variations. Future work includes optimizing the model checking algorithm and integrating abstracted instruction semantics to improve expressiveness and performance.

6. Introduction to PDA

Pushdown automata (PDA) are computational models that extend finite-state automata (DFA and NFA) by incorporating an auxiliary memory structured as a stack. This structure enables them to recognize context-free languages and to model systems with hierarchical or recursive behavior, such as function calls in software programs.

In model checking, pushdown automata are used for the analysis and verification of systems with a potentially infinite state space. A typical use case is the verification of software that employs recursion, dynamic data structures, or communication protocols with unbounded buffers. Unlike finite-state models, PDAs can represent the nesting depth of function calls, making them particularly useful for verifying properties related to the correctness of execution flows in programs.

6.1. Structure and Functioning

A pushdown automaton is formally defined as a tuple:

$$A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F),$$

where:

- Q is a finite set of states;
- Σ is the input alphabet;
- Γ is the stack alphabet (set of symbols that can be written to the stack);
- δ is the transition function:

$$\delta : Q \times \Sigma \times \Gamma \to \mathcal{P}(Q \times \Gamma^*),$$

  which describes how the state and stack evolve based on the input and the top symbol of the stack;
- $q_0 \in Q$ is the initial state;

- $Z_0 \in \Gamma$ is the initial stack symbol;
- $F \subseteq Q$ is the set of final states.

The transitions of the automaton can modify both the state and the stack simultaneously, allowing for push (writing to the stack), pop (removing the top symbol), and stack maintenance operations.

6.2. Applications in Model Checking

One of the most relevant applications of PDAs in model checking is the verification of recursive systems. For example, the representation of a program with function calls can be modeled as a pushdown automaton, where:

- The states of the automaton represent code instructions;
- The input corresponds to the sequence of program execution;
- The stack stores active function calls, simulating the behavior of an activation stack (call stack).

This representation allows for verifying properties such as the reachability of critical states (e.g., segmentation faults due to incorrect balancing of calls and returns).

A concrete case is the verification of programming languages with recursion, such as C or Java, where the depth of calls is not a priori limited. Techniques based on pushdown automata are used to verify whether a given property, specified in temporal logics such as Linear Temporal Logic (LTL) or Computation Tree Logic (CTL), is satisfied in all possible execution paths.

7. Analysing Paper: Pushdown Model Checking for Malware Detection

A significant contribution to malware detection through model checking was introduced by Fu Song and Tayssir Touili, who proposed a novel approach based on *Pushdown Systems* (PDS) to model program behavior, with particular attention to stack-based operations, which are frequently exploited by malware obfuscation techniques.

7.1. Key Methodology

The core idea of the approach is to model the execution of a binary program using a *Pushdown System*, a computational model capable of handling recursive procedures and unbounded call stacks, thus accurately capturing the behavior of the program's control flow, including calls and returns.

The detection process consists of the following main steps:

1. **Program Modeling:** The binary executable is disassembled and represented as a set of Control Flow Graphs (CFGs), one for each procedure. The nodes represent program locations, and edges are labeled with assembly instructions (e.g., `mov eax, 0`). These CFGs are translated into a Pushdown System (PDS), where:
   - PDS control states represent program locations.
   - The PDS stack tracks the call stack, capturing return addresses and potentially pushed values.
   - Transitions mimic the behavior of instructions such as `push`, `pop`, `call`, and `ret`.
2. **Behavior Specification with SCTPL:** To specify malicious behavior, the authors introduce *Stack Computation Tree Predicate Logic* (SCTPL), an extension of *Computation Tree Logic* (CTL). SCTPL enhances CTL with:
   - *Quantified variables* to refer to registers or values across states.
   - *Predicates over the stack*, allowing the expression of constraints on the stack content using regular expressions.

This makes it possible to define sophisticated malicious patterns, including those involving obfuscated sequences of `push` and `pop` instructions designed to hide system calls.

3. **Model Checking Process:** Malware detection is reduced to verifying whether the PDS representing the program satisfies an SCTPL formula that characterizes malicious behavior. This model-checking problem is further reduced to an *emptiness check* on a *Symbolic Alternating Büchi Pushdown System* (SABPDS). This symbolic representation allows for efficient handling of quantified variables and regular expressions over stack contents.

7.2. Example: Obfuscated API Call Detection

One concrete example presented in the paper concerns the detection of obfuscated calls to critical API functions, such as `GetModuleHandleA`, often used by malware to locate and manipulate system components.

A typical malicious behavior is to push the parameter 0 onto the stack, call `GetModuleHandleA`, and later retrieve the module handle. Malware may obfuscate this behavior by adding arbitrary `push/pop` sequences between the parameter push and the call itself. This class of behaviors can be captured using the following SCTPL formula:

SCTPL Formula for Obfuscated API Call Detection

$$\Psi = \exists r_1 \ \mathsf{EF}\left(\mathsf{mov}(r_1, 0) \wedge \mathsf{EX}\ \mathsf{E}\begin{bmatrix}\neg\exists r_2\ \mathsf{mov}(r_1, r_2)\ \mathsf{U}\ \big(\mathsf{push}(r_1)\wedge \\ \mathsf{EX}\ \mathsf{E}\left[\neg\big(\mathsf{push}(r_1) \vee (\exists r_3(\mathsf{pop}(r_3) \wedge r_1\Gamma^*))\big)\ \mathsf{U}\right. \\ \qquad\qquad \left.(\mathsf{call}(\mathsf{GetModuleHandleA}) \wedge r_1\Gamma^*)\right)\end{bmatrix}\right)$$

This formula expresses that:

- There exists a register $r1$ assigned the value 0.
- $r1$ is pushed onto the stack.
- $r1$ is not overwritten until the function `GetModuleHandleA` is called.
- At the time of the call, $r1$ (value 0) must be on top of the stack.

This allows the detection of both straightforward and obfuscated versions of the same malware behavior.

7.3. Example: Detecting Kernel32.dll Base Address Lookup

Another concrete example presented in the work of Song and Touili concerns the detection of a class of malware that locates the base address of the `Kernel32.dll` library in memory. Many Windows-based malware need access to system APIs provided by `Kernel32.dll` to perform their malicious activities, such as file manipulation, process creation, or network communication. To do this, the malware scans process memory to locate the `DOS header` (signature `MZ`) followed by the `PE header` (signature `PE00`).

This sequence of operations can be formally specified using the following SCTPL formula:

SCTPL Formula for Kernel32.dll Base Address Detection

$$\Psi_{wv} = \mathsf{EG}(\mathsf{EF}(\exists r_1\ \mathsf{cmp}(r_1, 5A4D_h) \wedge \mathsf{EF}\ \exists r_2\ \mathsf{cmp}(r_2, 4550_h)))$$

This formula expresses that:

- There exists a recurring (looping) behavior (captured by `EG`) in which the program scans memory (captured by `EF`) for two consecutive markers.
- First, some register $r1$ is compared to the value `5A4Dh` (the `MZ` signature).
- Later, some register $r2$ is compared to the value `4550h` (the `PE` signature).

This behavioral pattern is a clear indicator of malware that dynamically locates `Kernel32.dll` to retrieve the addresses of critical API functions. Importantly, the use of variables $r1$ and $r2$ makes the formula generic across different malware samples, irrespective of the registers they actually use, which significantly enhances detection robustness even in the presence of minor obfuscations.

This example illustrates the power of SCTPL in expressing behavioral patterns that span multiple instructions and depend on ordered memory scans and comparisons, going beyond what traditional static signature matching could achieve.

7.4. Advantages and Comparison with Other Approaches

Compared to traditional signature-based methods, this approach offers several advantages:

- **Robustness to Obfuscation:** SCTPL specifications describe behavioral properties rather than syntactic patterns, making them resilient to common obfuscation techniques.
- **Precision:** By explicitly modeling the stack and tracking register values, the system captures the exact semantics of many attacks.
- **Generality:** A single SCTPL formula can match multiple variants of the same malware family, including those with different instruction sequences but equivalent malicious behavior.

Compared to previous work based on *Computation Tree Predicate Logic* (CTPL), SCTPL offers greater expressive power by incorporating regular expressions over the stack, enabling the specification of behaviors that rely on sequences of pushes and pops.

7.5. Experimental Evaluation

The proposed method was evaluated on well-known malware families such as `MyDoom` and `NetSky`. The results demonstrated:

- **High Detection Rates:** The SCTPL formulas successfully detected both known and obfuscated variants.
- **Obfuscation Resistance:** The method remained effective even when dead code or misleading sequences of `push` and `pop` instructions were inserted.
- **Reasonable Performance:** Analysis of small binaries (150 instructions) was completed within a few seconds, indicating feasibility for targeted analysis.

7.6. Limitations and Future Work

Despite its strengths, the approach also has some limitations:

- **Computational Complexity:** Model checking for pushdown systems, especially with quantified SCTPL formulas, can be computationally expensive, particularly for large binaries.
- **Specification Crafting:** Defining effective SCTPL formulas requires expert knowledge of malware behaviors and potential obfuscations, which may limit the automation potential.
- **Scalability:** Although suitable for detecting embedded malware components or targeted analysis, applying this technique to large-scale, real-time malware scanning remains challenging.

Future work should focus on:

- Developing automated SCTPL formula generation from malware samples.
- Combining SCTPL-based analysis with machine learning techniques to improve robustness and reduce false positives.
- Exploring abstraction techniques to mitigate the computational overhead when analyzing large programs.

Overall, the work of Song and Touili demonstrates that formal methods, particularly stack-aware model checking, can be an effective weapon in the ongoing fight against increasingly sophisticated malware.

8. Practical Implementation of Model Checking for Malware Detection

Model Checking-Based Malware Detection offers a robust approach to identifying malicious software, but its practical implementation requires careful consideration. This section explores how model checking can be integrated into conventional antivirus systems and cloud-based solutions.

8.1. Integration with Conventional Antivirus Systems

Integrating model checking into traditional antivirus software can enhance detection capabilities, especially for new or obfuscated malware. However, it comes with challenges.

8.1.1. Architectural Considerations

- **Behavioral Analysis Module:** A dedicated module within the antivirus software that uses model checking to analyze the control flow and state transitions of suspicious programs. This module would extract the program's control flow graph (CFG) and translate it into a state transition model.
- **Temporal Logic Engine:** A component that applies temporal logic formulas (e.g., LTL, CTL, or SCTPL) to the state transition model to verify malicious behavior.
- **Performance Optimization:** Techniques such as symbolic verification and parallel processing can be used to reduce the analysis time. Model checking could be triggered selectively for high-risk programs.

8.1.2. Challenges

- **Performance Overhead:** Model checking is computationally expensive, which could impact real-time scanning performance.
- **Expert Knowledge Requirement:** Crafting effective temporal logic formulas requires deep expertise, limiting the ability to quickly update detection rules.
- **Scalability:** Model checking may not be suitable for large-scale, real-time scanning due to its high resource requirements.

8.2. Cloud-Based Model Checking for Malware Detection

Cloud-based solutions can leverage distributed computing resources to perform intensive model checking tasks, making it feasible to analyze large-scale systems or multiple programs simultaneously.

8.2.1. Architectural Considerations

- **Cloud-Based Analysis Engine:** A cloud service that performs model checking on suspicious files uploaded by client devices. The engine would extract CFGs, translate them into state transition models, and apply temporal logic formulas.

- **Client-Side Lightweight Scanner:** A lightweight scanner on the client side performs initial heuristic and signature-based scans. Suspicious files are uploaded to the cloud for deeper analysis.
- **Distributed Processing:** The cloud service uses distributed computing techniques to parallelize the model checking process, reducing analysis time.
- **Real-Time Feedback:** Analysis results are sent back to the client device for appropriate action.

8.2.2. Advantages

- **Scalability:** Cloud-based solutions can handle large-scale analysis without overburdening client devices.
- **Resource Efficiency:** Offloading computational workload to the cloud maintains high performance on client devices.
- **Rapid Updates:** The cloud-based system can be updated with new temporal logic formulas and detection rules more quickly.

8.2.3. Challenges

- **Latency:** Uploading files to the cloud and waiting for analysis results could introduce latency.
- **Privacy Concerns:** Uploading potentially sensitive files to a cloud service raises privacy and data security concerns.
- **Cost:** Maintaining a cloud-based model checking service could be expensive.

8.3. Hybrid Approach

A hybrid approach combines the strengths of both conventional antivirus systems and cloud-based solutions. Lightweight model checking is performed on the client side for initial detection, while more complex analyses are offloaded to the cloud. This balances performance, scalability, and detection accuracy.

8.4. Summary

Model Checking-Based Malware Detection can be implemented in both conventional antivirus systems and cloud-based architectures, each with its own advantages and challenges. A hybrid approach may provide the most practical solution, combining the speed of client-side scanning with the power of cloud-based analysis. As the field evolves, further research into optimization techniques and automation of temporal logic formula generation will be key to making model checking a viable component of modern malware detection systems.

9. Conclusion

Model Checking-Based Malware Detection represents a significant advancement in the field of cybersecurity, offering a formal and rigorous approach to identifying malicious software. By leveraging the principles of model checking, this technique provides a precise method for analyzing program behavior and detecting deviations that may indicate malicious intent. The use of temporal logics such as LTL, CTL, and their extensions like CTPL and SCTPL allows for the specification of complex behavioral patterns, making it possible to detect malware that employs sophisticated obfuscation techniques.

The case studies and methodologies discussed in this paper highlight the potential of model checking to address some of the most challenging aspects of malware detection, including zero-day attacks and polymorphic malware. The integration of pushdown automata and stack-aware model checking further enhances the capability to analyze

recursive and hierarchical program structures, which are often exploited by advanced malware.

However, the approach is not without its challenges. The computational complexity and scalability issues associated with model checking, particularly for large-scale systems, remain significant hurdles. Additionally, the need for expert knowledge to craft effective temporal logic formulas can limit the automation potential of this technique.

Future research should focus on developing more efficient model checking algorithms, automating the generation of temporal logic specifications, and integrating model checking with other detection methods such as machine learning to create hybrid solutions. These advancements will be crucial in enhancing the robustness, scalability, and practicality of model checking-based malware detection.

In conclusion, while model checking-based malware detection is still evolving, its formal and precise nature makes it a powerful tool in the ongoing battle against increasingly sophisticated cyber threats. As the field continues to advance, it holds the promise of providing more effective and reliable solutions for securing computer systems against malware.