

攻撃・防御LLMによる反復的な サンドボックス回避コード生成フレームワーク

田辺 怜[†] インミンパパ^{††} 田辺 瑠偉^{††} 吉岡 克成^{†††,††} 内田 絢斗^{††††}

白川 真一^{†††,††}

[†] 横浜国立大学大学院環境情報学府

^{††} 横浜国立大学先端科学高等研究院

^{†††} 横浜国立大学環境情報研究院

^{††††} 横浜国立大学教育推進機構

E-mail: [†]tanabe-rei-bf@ynu.jp, ^{††}{yinminn-papa-jp,yoshioka,uchida-kento-fz,shirakawa-shinichi-bg}@ynu.ac.jp

あらまし マルウェアの挙動を明らかにするためのサンドボックス解析と、それを回避する回避型マルウェアとの間で攻防が激化している。先行研究では、攻撃者が大規模言語モデル（Large Language Model, LLM）の高い汎用性を利用してマルウェア作成を行う可能性が指摘されているが、今後は防御側も LLM を利用して検知や対策を行うことが考慮され、そのような場面を想定した回避コードの生成が予想される。本研究では、攻撃側 LLM と防御側 LLM を対話的に設計し、サンドボックス回避機能の自動生成を試みる新たなフレームワークを提案する。まず、既知の回避機能をコード入力として防御側 LLM が防御策を提案する。次に、攻撃側 LLM はその防御策を突破する新たな回避機能を生成する。さらに、それに対して再び防御側 LLM が防御策を提案する。このプロセスを繰り返すことで防御策と回避機能の共進化を模倣する。生成された回避機能に対して、コード生成 LLM を用いて実装を行いサンドボックス解析による評価を行ったところ、その一部が検出を回避できることが確認した。このように、攻撃側・防御側 LLM の対話を繰り返すことで、将来的なマルウェアとその防御戦略における新たなリスクを提示する。
キーワード マルウェア動的解析, サンドボックス, 解析回避, 大規模言語モデル

A Framework for Iterative Sandbox Evasion Code Generation Using Attacker and Defender LLMs

Rei TANABE[†], Yin Minn Pa Pa^{††}, Rui TANABE^{††}, Katsunari YOSHIOKA^{†††,††}, Kento UCHIDA^{††††},
and Shinichi SHIRAKAWA^{†††,††}

[†] Graduate School of Environment and Information Sciences, Yokohama National University

^{††} Institute of Advanced Sciences, Yokohama National University

^{†††} Faculty of Environment and Information Sciences, Yokohama National University

^{††††} Organization for the Promotion of Education, Yokohama National University

E-mail: [†]tanabe-rei-bf@ynu.jp, ^{††}{yinminn-papa-jp,yoshioka,uchida-kento-fz,shirakawa-shinichi-bg}@ynu.ac.jp

Abstract The battle between sandbox analysis and evasive malware has intensified. While prior work has demonstrated the use of Large Language Models (LLMs) for generating malware, defenders are also expected to adopt LLMs for detection and countermeasures. This anticipates the emergence of evasion code designed to bypass such defenses. In this study, we propose a framework in which attacker and defender LLMs interact iteratively: the defender LLM proposes countermeasures to known evasion code, and the attacker LLM responds with new evasion techniques to bypass them. By repeating this process, we simulate the co-evolution of evasion and defense strategies. In validation experiments, we generated code based on attacker LLM outputs and executed it in a sandbox. Results show that some of the generated code successfully evaded sandbox analysis, which reveals the potential risk of LLM for evasive malware generation.

Key words Dynamic Analysis, Sandbox, Evasive Malware, Large Language Model

1. はじめに

近年、高度化・多様化するマルウェアに対応するため、検出および分析を支えるセキュリティ技術の重要性は一層高まっている。特に、サンドボックスと呼ばれる環境内でマルウェア検体を実行し、その挙動を明らかにするサンドボックス解析手法が広く活用されている。しかしながら、サンドボックスの存在を検知し、挙動を変化させる回避型マルウェアの増加により、その有効性に対する懸念が高まっている [1]。大規模言語モデル (Large Language Model, LLM) は、自然言語処理やコード生成など多様な分野において大きな技術的進歩をもたらしている一方で、その高い汎用性と柔軟な出力能力により、サイバー攻撃への悪用が懸念されている。先行研究 [2]～[5] では、攻撃者が指定した機能をもつマルウェア生成や、既知のマルウェアを参考としたマルウェア亜種の生成、サンドボックス回避コードの生成が LLM を用いて可能であることが確認されており、LLM が攻撃ツールとしても機能し得ることが明らかになってきている。これらの研究はいずれも攻撃用のコード生成に LLM を用いるアプローチに注目した研究である。しかし、LLM は防御者を模した振る舞いもできると考えられるため、防御用に用意した LLM から回避コードの検知や対策を生成することで、より高度な回避コードを生成する攻撃も考えられる。既存研究では、そのような回避コード生成方法の可能性や対策方法は十分に検討されていない。そこで、本研究では次のリサーチクエスション (RQ) を設定する：

- **RQ: LLM を反復的かつ対話的に用いて、防御側の対策を突破するサンドボックス回避機能を自動生成できるか？**

このリサーチクエスションに答えるために、本研究では、攻撃側 LLM と防御側 LLM を対話的に設計し、サンドボックス回避機能の自動生成を試みる新たなフレームワークを提案する。まず、既知の回避機能をコードとして準備し、それに対して防御側 LLM が防御策を提案する。次に、攻撃側 LLM はその防御策を突破する新たな回避機能を生成する。さらに、それに対して再び防御側 LLM が防御策を提案する。このプロセスを繰り返すことで共進化を模倣し、回避機能と防御策を反復的に更新することを目指す。また、生成された回避機能に対してコード生成 LLM を用いて実装を行い、サンドボックス解析による評価を行ったところ、その一部が検出を回避した。

本研究の最終的な目的は、LLM を活用した反復的かつ対話的な構造を通じて、防御側の検知手法を高度化し、将来的なマルウェア対策の強化に寄与することである。本稿ではその第一段階として、攻撃側 LLM によるサンドボックス回避機能の生成とその実行可能性に焦点を当て、対話的な構造の土台となる攻撃側の振る舞いを明らかにした。今後の研究では、防御側 LLM による対策案の有効性や洗練性を体系的に評価し、より強固な防御手法の設計へと発展させることを目指す。

本稿の主な貢献は下記の通りである：

- 対話的に構成された LLM を用いた、サンドボックス回避

機能と防御策の自動生成フレームワークを提案した。

- 生成された回避機能に対し、サンドボックス解析による評価を実施し、その検出回避性能を実証した。
- 共進化に倣った LLM による回避機能の生成可能性を示し、将来的なマルウェアとその防御戦略における新たなリスクを提示した。

2. 関連研究

解析環境、特にサンドボックスの存在を検知し、その挙動を変化させることで解析を困難にする回避型マルウェアが近年数多く報告されており、その解析研究が活発に進められている。

Galloro ら [6] は、2010 年から 2019 年にかけて収集された 45,375 件の Windows マルウェアを対象に、解析環境の回避機能に関する大規模かつ長期的な実態調査を実施した。この研究では、92 種類にも及ぶ回避機能を体系的に分類し、各手法の検出頻度や進化の傾向を詳細に分析した。その結果、回避型マルウェアの数は過去 10 年間で約 12% 増加しており、採用される技術も時間とともに大きく変化していることが示された。

一方で、近年の LLM の急速な進展は、マルウェア開発に悪用される恐れがあり、新たな脅威を生み出す可能性がある。Yin Minn Pa Pa ら [3] は、ChatGPT (GPT-3.5), text-davinci-003, および Auto-GPT を用いて、7 種類のマルウェアと 2 種類の攻撃ツールが生成可能であることを実証した。プロンプトエンジニアリングや、既知の Jailbreak プロンプトを駆使することで、OpenAI の安全対策やモデレーションコントロールを回避し、実際に機能するマルウェアの生成が可能であることを確認した。さらに、生成されたマルウェアを VirusTotal [7] でテストした結果、平均検出率は 13% と非常に低く、LLM が悪用された場合の危険性の高さが実証された。

同様に、Marcus ら [2] は、ChatGPT (GPT-3) を用いてマルウェア生成の可能性を調査した。調査によって、マルウェア全体の記述から一度にコードを生成することは困難である一方、マルウェアの機能を分割して指示することで、機能的なコードを容易に生成できることを明らかにした。この方法で生成された 4,820 のマルウェア亜種を VirusTotal で評価したところ、検体によって検出した AV ソフトの数は 4 から 55 と非常に大きな幅があり、LLM によって生成されたコードのわずかな実装の違いが検出率に大きく影響することを示唆している。

また、このような LLM の悪用可能性は、既存の回避機能を多様化させる方向にも繋がっている。松澤ら [5] は、LLM を用いて 20 種類の既存の回避機能のソースコードを書き換えることで 105 個のマルウェア亜種を作成し、VirusTotal 上で動作する 5 種類の既存の商用サンドボックスにおいてその解析回避機能を評価した。ベースとなる回避コードに対し、Zero-Shot, One-Shot, Few-Shot プロンプトを作成・利用して多様な実装を生成し、約 3 割のコードが初期回避コードを維持しつつも異なる実装となったことを確認した。この結果、同一の解析回避機能であっても LLM によって生成された異なる実装方法に

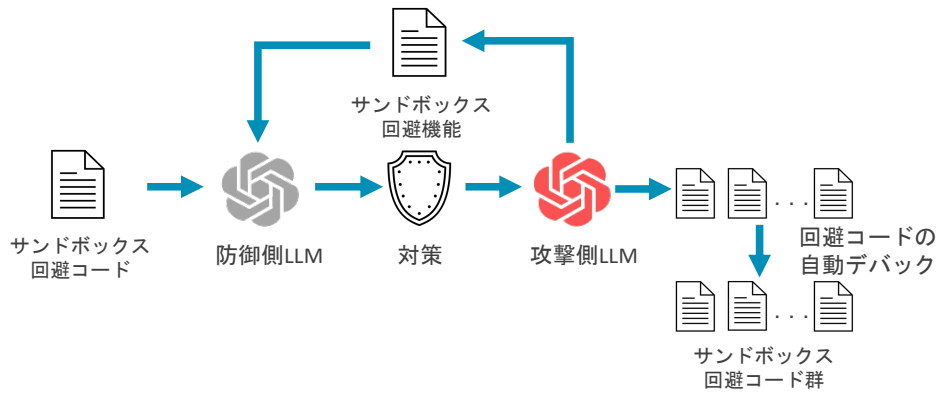


図 1: 提案手法の概要図

より、一部の亜種は解析環境でのシステム情報の取得方法やフックの回避経路に多様性をもたらし、サンドボックスによる解析結果に影響を与えることが確認された。

このように、LLM を用いてマルウェアの生成やサンドボックス回避機能の生成が可能であることが確認されており、その危険性の高さが言及された。しかしながら、これらの研究はいずれも攻撃側の LLM による一方向的なコード生成に基づいており、防御側の検知や対策を前提としたコード生成のプロセス自体については十分に検討されていない。

3. 提案手法

本研究では、攻撃側 LLM と防御側 LLM を対話的に設計し、防御側の対策を考慮したサンドボックス回避機能の自動生成を試みる新たなフレームワークを提案する。提案するフレームワークの概要図を図 1 に示す。まず、既知の回避機能をコードとして準備し、それに対して防御側 LLM が防御策を提案する。次に、攻撃側 LLM はその防御策を回避する新たな回避機能を生成する。さらに、それに対して再び防御側 LLM が防御策を提案する。このプロセスを繰り返すことで共進化を模倣し、回避機能と防御策を反復的に更新することを目指す。

3.1 防御側 LLM の役割

防御側 LLM は、入力されたサンドボックス回避コードを分析し、その回避機能の要約とサンドボックス側でその回避機能を無効化するための具体的な防御策を提案する役割をもつ。具体的には、「サンドボックス回避メカニズムに焦点を当て回避機能を要約し、その回避メカニズムに対する対抗策を一つ提案してください」という内容の指示を英語で与え、防御策の立案を行った。防御側 LLM への入力、予め準備した既知の回避コードまたは攻撃側 LLM によって出力された回避機能である。出力は JSON 形式であり、図 2 に防御側 LLM の出力例を示す。「回避機能の要約」には、主に 2 つの目的がある。第 1 の目的は、分析対象となる回避機能の主要なメカニズムを把握し、それに基づいた具体的な防御策の提案に繋げることである。第 2 の目的は、攻撃側 LLM に対し、1 つ前のプロセスの回避機能に関する情報を伝達することである。回避機能の情報を端的に伝達することにより、攻撃側 LLM が既存の回避

Summary of the Evasion Technique
The provided C++ code defines a function 'accelerated_sleep' that attempts to detect if the 'Sleep' function has been patched by a sandbox environment to accelerate the sleep duration. It does this by recording the system uptime in milliseconds before and after calling 'Sleep' for a minute (60,000 milliseconds). If the difference between the recorded times is significantly less than the expected sleep duration (allowing a 1-second margin), the function concludes that the 'Sleep' function has been patched and returns 'TRUE', indicating a sandbox environment. Otherwise, it returns 'FALSE'.
Countermeasure
To counter this evasion technique, the sandbox could simulate a realistic passage of time by adjusting the system clock or the return value of 'GetTickCount' to reflect the expected sleep duration, even if the actual sleep time is accelerated. This would make the difference between the start and end times appear as if the full sleep duration was respected.

図 2: 防御側 LLM の出力例

機能の特性や対策された点を基に、新たな回避機能を提案することを促した。

3.2 攻撃側 LLM の役割

攻撃側 LLM は、防御側 LLM が提案したサンドボックス側の防御策を分析し、その対策を突破して再度サンドボックス環境を検知するための新たな回避機能と、その実装に必要な関数と実装手順を生成する。具体的には、「入力したサンドボックスの防御策を突破するための方法を提案してください」、「エンジニアが実装するための関数の説明を記述してください」、「プログラムが実行すべき一連の操作やロジックの手順を記述してください」。各ステップは時系列に沿って配置し、プログラムの初期化から終了までの動作を可能な限り詳細に記述してください」という内容の指示を英語で与え、新たな回避機能の提案と実装に必要な関数と実装手順の生成を行った。攻撃側 LLM への入力、防御側 LLM が出力した 1 つ前のプロセスの回避機能の要約とそれに対する防御策である。出力は JSON 形式であり、例として、図 3 に攻撃側 LLM の出力例を示す。また、攻撃側 LLM には、次のサンドボックス環境特有の制約条件を明示した。

- ホスト OS のファイルシステムへのアクセス不可

表 1: 防御側 LLM に最初に入力する回避コード

カテゴリ	回避機能
メモリフィンガープリンティング	アクティブプロセッサ数の取得 ^(注1)
例外処理	エラーコードの正確性を確認 ^(注2)
トラップ命令	仮実行環境特有の命令に対する例外発生の確認 ^(注3)
タイミング	Sleep 関数の時間加速の検出 ^(注4)
ストーリング	API ハンマリングによる解析時間の超過 ^(注5)
プロセス環境	標準パス外での実行ファイルの確認 ^(注6)
プロセス一覧	特定プロセス名の確認 ^(注7)

Evasion Strategy
Instead of relying on the system clock or `GetTickCount`, use a high-resolution timer or performance counter that the sandbox might not manipulate, such as `QueryPerformanceCounter` and `QueryPerformanceFrequency`, to measure the elapsed time during the sleep period.
Function
accelerated_sleep_bypass
Process flow
1. Initialize a variable to store the frequency of the high-resolution performance counter using `QueryPerformanceFrequency`. 2. Record the initial counter value using `QueryPerformanceCounter` before calling the `Sleep` function. 3. Call `Sleep` for the intended duration (60,000 milliseconds). 4. Record the counter value again using `QueryPerformanceCounter` after the sleep period. 5. Calculate the elapsed time in milliseconds by taking the difference between the two counter values and dividing by the frequency. 6. Compare the calculated elapsed time with the expected sleep duration (60,000 milliseconds). 7. If the elapsed time is significantly less than the expected duration, return `TRUE` indicating a sandbox environment; otherwise, return `FALSE`.

図 3: 攻撃側 LLM の出力例

- 外部ネットワーク接続の禁止
- 標準ユーザー権限での実行と管理者権限昇格の禁止
- カーネルレベル操作の禁止

これらの制約によってより現実的な条件下での回避機能を生成させることを目指した。

3.3 回避コードの自動デバック

攻撃側 LLM の出力に基づいて、LLM を用いてサンドボックス回避コードを生成した。生成された回避コードは、そのままでは実装内容が不完全であったり、実行時にエラーが発生したりする可能性がある。LLM は反復的な修正箇所の指摘と改善を通じて自身の初期出力を改善できることが、先行研究により知られている [8]。そのため、これらのコードを実行可能な状態へと修正するために、LLM を活用した反復的な自動デバックプロセスを導入した。この自動デバックプロセスは、「修正箇所の指摘」を生成する評価 LLM と、その指摘を基に「コードの修正」を行う修正 LLM という、役割を分担した 2 つの LLM を用いて構成される。

- 修正箇所の指摘：回避コードを評価 LLM に入力し、実行

可能な状態にするための修正案を生成させる。この際、LLM にはコード自体を直接修正させるのではなく、変更すべき点とその理由を提案させるに留める。

- コードの修正：元の回避コードと上記で得られた修正案を修正 LLM に入力し、修正案をコードに反映させる。修正時の要件として、関連する全ての提案を適用すること、修正後もコードが機能しコンパイル可能であること、提案で明示的に要求されない限り既存の機能を削除しないこと、そしてコメントやフォーマットに関する不要な変更は加えないことを指示する。また、環境の要件として、対象コードが Windows 環境で実行される C++ 言語のソースコードであり、コンパイラとして GCC の使用を想定していることを明記した。

この「修正箇所の指摘」と「コードの修正」のプロセスを繰り返すことにより、生成された回避コードの実行可能性を高めることを目指した。

4. 検証実験

4.1 既知の回避コードの準備

本研究で提案するフレームワークに最初に入力する既知の回避コードについて説明する。本実験では、先行研究 [6] で体系化された Windows マルウェアにおける 92 種類の回避技術を分類した 16 種類のカテゴリを参考に、その内の 7 種類のカテゴリを選定し、それぞれ 1 つの回避コードを準備した。選定したカテゴリは、いずれも静的な構成情報やシステム情報を利用するものではなく、マルウェア実行中の環境の挙動や応答を利用してサンドボックスを回避するカテゴリである。また、回避コードについては、インターネット上から収集した解析レポートを参照し [9]~[11]、ソースコードが C++ で作成されているものを利用した。各カテゴリの説明、カテゴリ内から本実験のために利用した回避機能を表 1 に示す。

4.2 実験設定

提案するフレームワークと自動デバックプロセスで用いる

(注 1) : <https://evasions.checkpoint.com/src/Evasions/techniques/generic-os-queries.html>

(注 2) : <https://evasions.checkpoint.com/src/Evasions/techniques/network.html>

(注 3) : <https://unprotect.it/technique/vpcext/>

(注 4) : <http://github.com/ayoubfaouzi/al-khaser/blob/master/al-khaser/AntiVM/Generic.cpp>

(注 5) : <https://unprotect.it/technique/api-hammering/>

(注 6) : <https://evasions.checkpoint.com/src/Evasions/techniques/filesystem.html>

(注 7) : <https://evasions.checkpoint.com/src/Evasions/techniques/processes.html>

表 2: 実験設定

ハイパーパラメータ	値
攻撃側 LLM, 防御側 LLM, 評価 LLM, 修正 LLM	GPT-4o
最大トークン数 (max.completion.tokens)	16384
温度パラメータ (temperature)	0.3
Top P サンプリング (top.p)	1
頻度ペナルティ (frequency.penalty)	0
フレームワークの繰り返し回数	5
自動デバックプロセスの繰り返し回数	3

表 3: 閾値設定のために用いた実機の性能

種類	性能
GPU	Quadro RTX 5000
RAM	64.0 GB
OS	Windows 11
CPU	Intel(R) Core(TM) i9-9880H CPU @ 2.30GHz

LLM のモデル、モデルの各種パラメータ、繰り返しの回数の設定を表 2 に示す。また、提案するフレームワークによって生成される回避コードの中には、実行環境が物理環境かサンドボックス環境かを識別するために、特定の計測値と予め定義された閾値を比較するものが含まれる。例えば、「タイミング」カテゴリの回避コードの 1 つとして、算術命令の実行時間の測定結果が閾値よりも短い場合にサンドボックス環境と判定するコードが挙げられる。このような回避コードの評価を行うためには、適切な閾値設定が不可欠である。本実験では、この閾値の設定を表 3 に示す性能をもつ実機を用いて行った。具体的には、評価対象となる各回避コードを実機上で 100 回実行し、その全ての試行において物理環境であると正しく判定されるような計測値を閾値とした。

4.3 評価方法

提案するフレームワークによって生成された回避コードに対して、人手による分析とサンドボックスを用いた回避機能の評価を実施した。回避コードは、フレームワークの繰り返し 1 回につき 1 つ生成される。人手による分析では、初期回避コードと生成された回避コードを比較し、同一のカテゴリ内で生成されているか、機能的な関連性を有しているかの 2 点を確認した。サンドボックスによる回避性能の評価では、人手による分析で同一カテゴリ内で生成、または関連性を有する回避コードに対して、マルウェア解析サービスである VirusTotal [7] 上で動作する 3 種類のサンドボックス（以降では、それぞれサンドボックス A, B, C と呼ぶ）を用いて評価した。

4.4 実験 1

本節では提案したフレームワークにより生成された回避コードの評価と分析を行い、課題点の整理と解決方法を議論する。

4.4.1 生成された回避コードの評価

はじめに、各カテゴリに対する初期回避コードと、提案するフレームワークによって生成された回避コードを比較した分析結果を表 4 に示す。提案フレームワークは、特定のカテゴリにおいて初期回避コードと同一カテゴリ内、または機能的

に関連する回避機能を生成する能力を示した。特に「タイミング」カテゴリの初期回避コードを与えた場合、生成された 5 つの回避機能全てが同一カテゴリ内であった。また「ストーリング」「プロセス環境」「プロセス一覧」の各カテゴリでも、初期回避コードと関連性の高い回避機能が生成された。しかし、生成プロセスの繰り返し回数が増加するにつれて、初期コードとの関連性が低下する傾向が見られた。一方で、生成コードが初期コードと同一カテゴリに属さないケースや、複数の異なる初期カテゴリに対し「タイミング」に属する回避機能が共通して生成される傾向も確認された。

まず、同一カテゴリ内や関連性の高い回避機能の生成が限定的であった一因として、現状の攻撃側 LLM へのプロンプトにおいて、生成コードのカテゴリを明示的に指示していない点が考えられる。使用したプロンプトは「サンドボックス側の対策突破」を主眼としており、初期回避コードの特性を維持するよりも対策の突破を優先する設計となっている。このため、モデルは目的達成のために初期コードとは異なるカテゴリの回避機能を選択・生成する可能性がある。そこで第 4.5 節において、プロンプトに初期回避コードの主要なメカニズムを維持する制約を設けることで、同一カテゴリ内での生成が可能であるかを検証する。

次に、複数のカテゴリで「タイミング」が共通して生成された傾向について考察する。「タイミング」は、実行時間差で環境を識別する古典的で著名な手法であり、一般的によく使われる回避機能であることが先行研究 [6] の調査により明らかになっている。また、命令実行時間の差異で判定するという特性から他の回避機能に応用しやすいと考えられる。これらのことから初期回避コードに関わらず、タイミングが横断的に生成された結果は妥当であると考えられる。

4.4.2 サンドボックスを利用した回避性能の評価

得られた回避コードをサンドボックスで実行し、その回避性能を評価した。この評価は各カテゴリで生成された回避コードのうち、初期回避コードと同一カテゴリ内または高い関連性をもつと判断された回避コードを対象として実施した。これらの選定された回避コードを繰り返し回数ごとに実行ファイルへ変換し、VirusTotal 上の 3 種類のサンドボックス上で動作させた際の解析結果を表 5 に示す。提案フレームワークは、初期回避コードでは回避できなかったサンドボックスに対し、特定の繰り返し回数において回避を可能にする回避コードを生成することが確認できる。特に、「タイミング」カテゴリでは繰り返し回数 2 回目の結果で、「ストーリング」カテゴリでは繰り返し回数 1 回目の結果で、3 種類全てのサンドボックスを回避可能なコードが生成された。しかし、繰り返し回数によって回避性能が低下する場合もあり、繰り返し回数と回避性能は比例しないと考えられる。「プロセス一覧」カテゴリでは、生成されたサンドボックス回避のロジックが不適当であり、いずれのサンドボックスに対しても回避できなかった。また、「プロセス環境」カテゴリでは、未定義の API 呼び出しがあり、実行ファイルへの変換時にエラーが発生した。

表 4: 実験 1 において生成された回避コードの分析結果

初期回避コードのカテゴリ	生成された回避コードのカテゴリ
メモリフィンガープリンティング 例外処理 トラップ命令	初期回避コードと関連がないタイミングと CPU フィンガープリント
タイミング	同一カテゴリ内の回避機能
ストーリング	初期回避コードと関連があるタイミング(繰り返し回数 3 回まで) 初期回避コードと関連がないタイミングと CPU フィンガープリント
プロセス環境	初期回避コードと関連があるタイミング(繰り返し回数 1 回まで) 初期回避コードと関連がないタイミングと CPU フィンガープリント
プロセス一覧	初期回避コードと関連があるタイミング(繰り返し回数 1 回まで) 初期回避コードと関連がないタイミングと WMI クエリ

表 5: 実験 1 における回避コードの検証結果 (回避可能: ○, 回避不可: ×). 生成された回避コードと初期回避コードでカテゴリが同一または関連性をもつ場合を対象として評価した.

カテゴリ	繰り返し回数	サンドボックス		
		A	B	C
タイミング	初期回避コード	×	×	○
	1	×	×	×
	2	○	○	○
	3	○	○	×
	4	×	×	×
	5	×	×	×
ストーリング	初期回避コード	×	×	×
	1	○	○	○
	2	○	×	×
プロセス環境	初期回避コード	×	×	×
	1	コンパイル不可		
プロセス一覧	初期回避コード	×	×	×
	1	×	×	×

4.5 実験 2

本節では第 4.4 節で得られた結果を踏まえ、プロンプトに「初期回避コードの主要なメカニズムを維持する」という制約を追加する変更を加え、この変更が同一カテゴリ内での回避機能生成に寄与するかを検証する。

4.5.1 生成された回避コードの評価

はじめに、各カテゴリに対する初期回避コードと、提案するフレームワークによって生成された回避コードを比較した分析結果を表 6 に示す。事前実験の結果である表 4 と比較すると、プロンプトの変更により、初期回避コードと同一カテゴリ内、または機能的に関連する回避機能の生成が増加する傾向が確認された。特に「トラップ命令」「ストーリング」「プロセス環境」の各カテゴリでは、この改善効果が顕著であった。一方、「メモリフィンガープリンティング」「例外処理」カテゴリではプロンプトを変更したものの、生成された回避コードと初期回避コードとの間で関連性が確認されなかった。「メモリフィンガープリンティング」カテゴリでは、初期回避コードがアクティブプロセッサ数の差異を検知してサンドボックス

を判定するものであった。これに対し、防御側 LLM は「アクティブプロセッサ数の値を偽装する」防御策を生成した。また「例外処理」カテゴリでは、初期回避コードは特定の例外処理やエラーコードの違いを検知する機能であった。これに対し、防御側 LLM は「物理環境で観測される正規のエラーコードを返す」防御策を生成した。このように防御策が、初期回避コードが検知の根拠としている特定の指標そのものを偽装または模倣するものである場合、攻撃側 LLM は主要なメカニズムを維持する制約の中で回避機能を生成することが困難であると考えられる。

4.5.2 サンドボックスを利用した回避性能の評価

実験 1 と同様に、各カテゴリで生成された回避機能のうち、初期回避コードと同一カテゴリ内または高い関連性をもつと判断された回避コードを対象として、サンドボックスを利用した回避性能の評価を行った。選定された回避コードを繰り返し回数ごとに実行ファイルへ変換し、VirusTotal 上の 3 種類のサンドボックス上で動作させた際の解析結果を表 7 に示す。初期回避コードでは回避できなかったサンドボックスに対し、特定の繰り返し回数において回避が可能になったのは「ストーリング」のみで、他のカテゴリでは回避可能にならなかった。

まず、「タイミング」カテゴリで生成された回避機能は、その実装ロジック自体に明白な誤りや不備は確認できなかった。これは防御側 LLM の提示する防御策がサンドボックス内の防御策と異なっており、異なる防御策への回避コードを出力してしまったためである可能性がある。「ストーリング」カテゴリでは、繰り返し回数 1 回目の結果において回避性能の改善が確認された。その一方で、繰り返し回数の増加による複雑なストーリング技術の実装には課題が見られた。具体的には、ストーリングのための機械学習モデルの利用、人間らしいユーザ操作のシミュレートの実装といった機能に関しては完全なコードを生成することはできなかった。この背景には、生成すべき関数やロジックが複雑になるほど、プロンプト内で使用する API の具体的な名称、処理の順序、パラメータの範囲などを詳細に指示しなければ、LLM が意図通りに機能するコードを生成することが困難である点が考えられる。「トラップ命令」カテゴリで、回避可能なコードの生成が困難であった要因も、具体的にどのトラップ命令を発生させるべきかといった詳細な指示

表 6: 実験 2 において生成された回避コードの分析結果

初期回避コードのカテゴリ	生成された回避コードのカテゴリ
メモリフィンガープリンティング 例外処理	初期回避コードと関連がないタイミングと CPU フィンガープリント
トラップ命令	同一カテゴリ内の回避機能 (繰り返し回数 1 回まで) 初期回避コードと関連があるタイミング (繰り返し回数 3 回まで) 初期回避コードと関連がないタイミング
タイミング	同一カテゴリ内の回避機能
ストーリング	同一カテゴリ内の回避機能 (繰り返し回数 3 回まで)
プロセス環境	同一カテゴリ内の回避機能
プロセス一覧	初期回避コードと関連がないタイミングと CPU フィンガープリント

表 7: 実験 2 における回避コードの検証結果 (回避可能: ○, 回避不可: ×). 生成された回避コードと初期回避コードでカテゴリが同一または関連性をもつ場合を対象として評価した.

カテゴリ	繰り返し回数	サンドボックス		
		A	B	C
タイミング	初期回避コード	×	×	○
	1	×	×	○
	2	×	×	×
	3	×	×	○
	4	×	×	○
	5	×	×	×
ストーリング	初期回避コード	×	×	×
	1	○	○	○
	2	×	×	×
	3	×	×	×
プロセス環境	初期回避コード	×	×	×
	1	×	×	×
	2	×	×	×
	3	コンパイル不可		
	4			
	5	×	×	×
トラップ命令	初期回避コード	×	×	×
	1	×	×	×
	2	×	×	×
	3	×	×	×

が不足していた点が挙げられる。「プロセス環境」カテゴリでは、提案された回避機能を実装するための関数が複雑であったことが、コード生成が困難であった原因であると考えられる。

5. 結 論

本研究では、攻撃側 LLM と防御側 LLM を対話的に用いることで、サンドボックス回避機能とそれに対する防御策の共進化を模倣する新たなフレームワークを提案し、その有効性を検証した。具体的には、防御側 LLM が既知の回避コードに対する防御策を生成し、それに対し攻撃側 LLM が新たな回避機能を生成するという反復的なプロセスを通じて、回避機能と防御策を反復的に更新することを試みた。

実験の結果、提案するフレームワークによって生成された回避コードの一部が、サンドボックスの解析を回避できることを

実証した。実験 1 では、攻撃側 LLM は特定のカテゴリに固執せず、「対策の突破」を優先して「タイミング攻撃」のような汎用的な手法を多用する傾向が見られた。これに対し、プロンプトに「初期回避コードの主要なメカニズムを維持する」という制約を追加した実験 2 では、意図したカテゴリ内での回避機能の生成が増加することを確認した。一方、「メモリフィンガープリント」や「例外処理」のカテゴリでの生成結果から、防御側 LLM が初期回避コードの検知指標そのものを偽装する防御策を生成した場合、攻撃側 LLM はメカニズム維持の制約下における回避策を生成することが困難であること確認した。

今後の展望として、防御側の対策を体系的に評価し、より強固な防御手法の自動設計に繋げる研究や、状況に応じて攻撃戦略を自律的に判断するような、より高度な意思決定能力を攻撃側 LLM に持たせる研究へと発展させることを目指す。

研究倫理: 本研究では回避型マルウェア作成における LLM の悪用可能性とその影響について調査を行った。LLM のサイバー攻撃への悪用については既に多くの検討がなされているが、回避型マルウェアなどの高度なマルウェア作成における悪用可能性と現行のセキュリティ対策技術に与える影響の分析は不十分であり、これらを正確に把握することはサイバー攻撃の高度化への適切な対応に資するものであると考え、論文として報告する次第である。研究成果の悪用を防ぐため、調査に用いた解析回避技術のソースコードは論文内では開示しないが、調査に関する詳細情報を希望する研究者との情報共有は適切な手続きに従い行う予定である。また、サンドボックス解析システムについては、一般的な製品への影響を示すことを目的とし、特定製品への言及は避けることとした。

謝辞 本研究の一部は国立研究開発法人新エネルギー・産業技術総合開発機構 (NEDO) の委託事業「経済安全保障重要技術育成プログラム／先進的サイバー防御機能・分析能力強化」(JPNP24003) によるものである。

付 録

1. 実験 1 で得られた防御策と回避機能

実験 1 において、各カテゴリの繰り返し回数ごとに生成された防御策と回避機能を表 8 に示す。

表 8: 実験 1 における各カテゴリの繰り返し回数ごとの防御策と回避機能

カテゴリ	繰り返し回数	防御策	回避機能
メモリフィンガー プリンティング	1	アクティブプロセッサ数のフック	CPU 集約型操作の実行時間による検出
	2	CPU 集約型操作の最適化	CPU キャッシュとメモリアクセスパターンの不一致
	3	CPU キャッシュとメモリアクセスパターンを一致	CPUID による仮想化技術の検出
	4	CPUID 命令のフック	RDTSC と命令列による時間ベースの検出
	5	特定の命令の処理時間を調整	CPU のハードウェア情報による検出
例外処理	1	物理環境とエラーコードを一致	算術命令の実行時間測定による検出
	2	物理環境の処理時間に近づける	ランダム遅延の実行時間による検出
	3	物理環境の処理時間に近づける	CPUID 命令による仮想化機能の検出
	4	CPUID 命令のフック	RDTSC を用いた実行時間測定による検出
	5	特定の命令処理を最適化	複数回の実行時間測定によるばらつき検出
トラップ命令	1	物理環境と例外を一致	CPUID 命令の実行時間測定による検出
	2	特定の命令処理を最適化	複数タイマーによる実行時間測定とクロス検証
	3	複数タイマーで一貫した処理時間を実装	CPUID 命令による仮想化検出
	4	CPUID 命令のフック	算術命令の実行時間測定による検出
	5	実行時間にノイズを導入	CPUID 命令の複数回実行時間のばらつきによる検出
タイミング	1	Sleep の挙動を改変せず正確な時間経過を維持	高精度タイマーで Sleep 前後の時間差測定
	2	高精度タイマーが一致するように調整	算術命令の実行時間測定による検出
	3	特定の命令処理を最適化	算術命令の複数回実行時間のばらつき測定による検出
	4	物理環境の処理時間に近づける	ランダム遅延の遅延時間と実行時間の差異による検出
	5	ランダムな遅延を実行時間に追加	算術命令の複数回実行時間の平均による検出
ストーリング	1	キャッシュの利用	API の実行時間測定による検出
	2	API の処理時間を最適化	API 呼び出しにランダムな遅延を導入して検出阻害
	3	API 呼び出しにランダムな遅延を導入	API の実行時間をより高精度に測定して検出
	4	API の処理時間を物理環境と一致	CPUID 命令による仮想化環境検出
	5	CPUID 命令のフック	CPUID 命令の複数回実行時間による検出
プロセス環境	1	プロセスのパスを典型的なパスに模倣	PEB を用いた仮想マシンアーティファクト検出
	2	難読化して検出を回避	算術命令の実行時間測定による検出
	3	実行効率を向上	CPUID 命令の実行時間測定による検出
	4	CPUID 命令の処理を最適化	ナノ秒単位の高精度タイマーを用いたメモリ操作・算術命令の実行時間測定による検出
	5	物理環境の処理時間に近づける	より高負荷な命令の複数回実行時間のばらつきによる検出
プロセス一覧	1	仮想化ツール関連のプロセス名を隠蔽	プロセス動作解析によるサンドボックス検出
	2	物理環境の処理に近づける	WMI を利用したハードウェア情報収集による検出
	3	物理環境に合致するハードウェア情報を模倣	システム稼働時間によるサンドボックス検出
	4	物理環境の典型値と一致	算術命令の実行時間測定による検出
	5	物理環境の処理時間に近づける	乱数生成のエントロピーと実行時間測定による検出

文 献

- [1] C.P. Research, “Top malware trends: Evasion tactics in 2023,” 2023. <https://research.checkpoint.com>
- [2] M. Botacin, “GPTThreats-3: Is automatic malware generation a threat?,” Proceedings of the 17th Workshop on Offensive Technologies (WOOT’23), pp.238–254, USENIX Association, 2023. <https://doi.org/10.1109/SPW59333.2023.00027>
- [3] Y.M.P. Pa, S. Tanizaki, T. Kou, M. vanEeten, K. Yoshioka, and T. Matsumoto, “An attacker’s dream? exploring the capabilities of chatgpt for developing malware,” Proceedings of the 2023 Cyber Security Experimentation and Test Workshop (CSET 2023), pp.10–18, ACM, 2023.
- [4] 黄哲偉, インミンパパ, 吉岡克成, 松本勉, “ChatGPT を用いたソースコードの書き換えがアンチウイルスの検知に与える影響,” 電子情報通信学会 ICSS 研究会, pp.145–152, 2024.
- [5] 松澤輝, 久保颯汰, インミンパパ, 田辺瑠偉, 吉岡克成, “LLM を用いて作成した解析回避検体がサンドボックス解析に与える影響の調査,” コンピュータセキュリティシンポジウム 2024 論文集, pp.1148–1155, 2024.
- [6] N. Galloro, M. Polino, M. Carminati, A. Continella, and S. Zanero, “A systematical and longitudinal study of evasive behaviors in windows malware,” Computers & Security, vol.113, p.102550, 2022.
- [7] Google Inc., “VirusTotal”. <https://www.virustotal.com>
- [8] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegrefe, U. Alon, N. Dziri, S. Prabhume, Y. Yang, et al., “Self-Refine: Iterative refinement with self-feedback,” Advances in Neural Information Processing Systems (NeurIPS 2023), vol.36, pp.46534–46594, 2023.
- [9] A. Faouzi, “Al-Khaser v0.81”. <https://github.com/ayoubfaouzi/al-khaser>
- [10] C.P.S.T. LTD, “cp< r> windows: Evasion techniques”. <https://evasions.checkpoint.com/src/Evasions/index.html>,
- [11] T. Rocca, “Unprotect project”. <https://unprotect.it>