

IoT デバイスのファジングのための LLM ベース生成型ミューテーション手法

中西 響[†] 長谷川健人^{††} 披田野清良^{††} 福島 和英^{††} 橋本 和夫[†]
戸川 望[†]

[†] 早稲田大学基幹理工学研究科 〒169-8555 東京都新宿区大久保 3-4-1

^{††} 株式会社 KDDI 総合研究所, 埼玉県ふじみ野市大原 2-1-15

E-mail: [†]{hibiki.nakanishi,kazuo.hashimoto}@togawa.cs.waseda.ac.jp,

^{††}{kt-hasegawa,se-hidano,ka-fukushima}@kddi.com, ^{†††}ntogawa@waseda.jp

あらまし 近年, IoT デバイスの普及に伴い様々なサービスが展開されている一方で, IoT デバイスの脆弱性を突いた悪意のある攻撃も増加している. ファジングは, IoT デバイスをブラックボックスに検査可能であり, 有効な脆弱性検査手法の一つである. ミューテーションベースのファジングは, 検査の初期値 (初期シード) にビットを反転させたり空にしたりする操作であるミューテーションオペレータを適用して変異させることで, 新たな検査値を生成し, 検査を実行する. しかし, 多くのミューテーションベースのファジングツール (ファザー) は, ミューテーションオペレータの選択を最適化することはあっても, ミューテーションオペレータそのものはファザーに事前定義されており, ファジングの実行状況に応じた追加や変更はされない. 本稿では, ファジングの実行状況に応じた動的なミューテーションを実現するため, LLM を用いたミューテーション手法を提案する. 提案手法は, 脆弱性を検出しなかったシードを LLM に与え, それを踏まえたミューテーションを促すことで, 検査対象のデバイスに対する有効な検査値を得る. 評価実験の結果, 既存の IoT ファザーのミューテーションによる検査値では検出できなかった脆弱性の検出に成功した.

キーワード 大規模言語モデル, IoT デバイス, 脆弱性検査, ファジング, ミューテーション

1. はじめに

近年, IoT 技術は急速に発展し, スマートホーム, 産業制御システム, 医療機器など幅広い分野において実用化が進んでいる. IoT 技術の実用化に伴い, 世界中でインターネットに接続される IoT デバイスの数も急増しており, 2024 年時点で約 180 億台に達したその数は 2033 年には約 400 億台に増加すると予測されている [1]. IoT デバイスは, ネットワーク経由で個人情報や機密データを取り扱うケースも多く, セキュリティの確保が極めて重要な課題となっている [2]. しかし, IoT デバイスはハードウェア資源やソフトウェア更新の制約が大きく, 出荷後の脆弱性対応が限定的となる場合が多い. また, 開発工程においてコスト削減や開発期間短縮を目的としてサードパーティコンポーネントが利用されており, 製品内部の構造が不透明となる傾向がある [3]. 上記の背景から, 製品が市場に投入される前に未知の脆弱性を発見する手段として, 内部構造に依存せず検査を実行可能なファジングが有効とされている.

ファジングは, テスト対象のプログラムに対する大量のテストケースを生成し, 送信して動作を監視することで, セキュリティ上の欠陥を発見する手法である. ファジングは, テストケースの生成方法から生成ベースとミューテーションベースに分類

される. 図 1 に一般的なファジングのテストケース生成の流れを示す. 図 1 の破線の矢印は生成ベース, 実線の矢印はミューテーションベースによるテストケースの生成を示す. 生成ベースのファジングでは, 検査の初期値 (初期シード) の文法をもとに, テストケースを直接生成する. 一方, ミューテーションベースのファジングでは, 初期シードを変異させるミューテーションを繰り返すことでテストケースを生成し, ファジングを実行する. 一般的に, IoT デバイスを制御するための API やコマンドは入力フォーマットに対して厳密であり, フォーマットの異なる入力の受信を拒否する. IoT デバイスのファームウェアは入手困難である場合が多いため, 生成ベースのテストケース生成を適用するのは難しく, ミューテーションベースのファジングツール (ファザー) が広く使用されている [4], [5].

ミューテーションにおいて, 初期シードや既にミューテーションされたシードをどう変異させるかを決定するものをミューテーションオペレータと呼ぶ. ミューテーションオペレータには, ビット反転やバイト挿入, バイト削除などがある. ファジング実行中にどのミューテーションオペレータを選択するかを決定するため, ミューテーションスケジュールという仕組みがあり, マルコフ連鎖モンテカルロ (Markov Chain Monte Carlo: MCMC) や粒子群最適化 (Particle Swarm Optimization: PSO) を用いて実

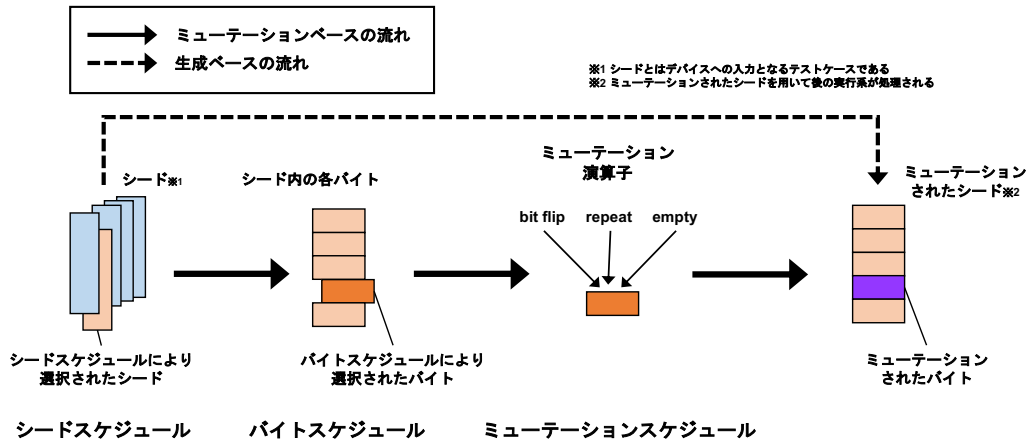


図 1: ファジングのテストケース生成の流れ。

行状況に応じ最適化される [6]。しかし、多くの場合、ミューテーションオペレータの種類はファザーごとに事前定義されており、ファジングの実行状況に基づいて追加されたり変更されたりしない [7],[8]。したがって、脆弱性検出の効率は初期シードの質やファザーに設定されたミューテーションオペレータの種類に大きく左右される。

ここで大規模言語モデル (LLM) に注目する [9]。LLM はさまざまな自然言語処理タスクを実行する言語モデルである。LLM は生成タスクを実行可能であり [10]、これを利用することで、デバイスごとの仕様の差異やファジングの実行状況に柔軟に対応可能なミューテーションの実現が期待される。

本稿では、デバイス仕様や検査の実行状況を考慮した動的なミューテーションを行うため、LLM ベース生成型ミューテーション手法を提案する。評価実験の結果、LLM ベース生成型ミューテーション手法による生成値は、IoT デバイスの既存ファザーである Snipuzz [8] のミューテーションよりも、特定の脆弱性検出において有効であることが確認された。

本稿の貢献を以下に示す。

- (1) LLM や Chain-of-Thought (CoT) を利用し、検査対象デバイスの仕様やファジングの実行状況を考慮した柔軟なミューテーションを行う手法を提案する。提案手法では、脆弱性を検出しなかったシードを LLM に与えてデバイス内に実装されている動作仕様や攻撃回避処理を推定する。得られた推定結果に基づき、それらを回避するよう設計された新たなシードを生成することで、より効果的な脆弱性の検出を目指す。
- (2) 評価実験の結果、提案手法は既存ファザーのミューテーションよりも脆弱性検出において有効なシードを生成することを確認した。

2. 既存手法

2.1 Snipuzz

Feng らは、IoT デバイスからのレスポンスメッセージを利用して、メッセージスニペットに基づいたミューテーションを行う Snipuzz を提案している [8]。Snipuzz では、レスポンスメッセージを編集距離、階層的クラスタリングに基づきスニペッ

トで分類する。スニペットは、特定のコード部分を実行するためのメッセージ中の連続したバイトであり、スニペットを推論することでメッセージを各バイトの役割に基づき分類する。Snipuzz は、メッセージスニペットに基づいたファジングを行うことで、入力の文法仕様に違反せず、ファームウェアのテストを行う。しかし、前述のように、Snipuzz を含む多くの既存手法はミューテーションオペレータを事前定義しており、ファジングの実行状況に基づいて追加されたり変更されたりしない。例えば、Snipuzz では特定のスニペットを繰り返したり、“on” や “null” など操作上問題が発生しそうな単語に置換したりするミューテーションオペレータが定義されているが、繰り返す回数や置換する単語は事前定義されている (表 4 を参照)。したがって、脆弱性検出の効率は初期シードの質やファザーに設定されたミューテーションオペレータの種類に大きく左右される。

2.2 LLMUZZ

Gao らは、IoT デバイスのための LLM ベースのファジング手法である、LLMUZZ を提案している [11]。LLMUZZ は、大きく分けて、LLM ベースの初期シード生成、静的解析による隠しキーワード分析、LLM ベースのミューテーションの 3 要素で構成される。初期シード生成では IoT デバイスのファームウェアを利用して、LLM がフロントエンドファイルを読み取ることで HTTP リクエストを生成する。隠しキーワード分析は、セキュリティの観点で危険性のあるパラメータの特定を目的としている。フロントエンドファイル、バックエンドバイナリファイルを解析してユーザからは見えない隠しキーワードを特定し、Taint 分析により伝播する関数に基づいたタグ付けを行う。例えば、*system* 関数に伝播する場合、キーワードは “Command Injection” のタグが付与され、*sprintf* 関数に伝播する場合は “Buffer Overflow” のタグが付与される。ミューテーションでは、データ構造の保持を目的とし、ミューテーション可能なフィールドを LLM が識別する。プロンプトでミューテーション対象の HTTP リクエストと Taint 解析で得たキーワードおよび付与されたタグ情報を与え、変異すべきフィールドをプレースホルダで置換するよう指示する。得られた出力中のプレースホルダは、外部エンジンにより実際のミューテーション値が挿入される。以下にプロンプトと出力の例を示す。なお、[11] では Prompt: LLMUZZ mutation に示す command infection 用のプ

ロンプトテンプレートのほかに、buffer overflow 用、関連パラメータ予測用のテンプレートが存在する。

Prompt: LLMUZZ mutation

Task: Use the taint information to mutate the following HTTP request for fuzzing of IoT devices. Use <payload_command> in place of the command to be injected.

Taint information:

```
{“Command Injection”: start_time};  
{“Command Injection”: end_time_default}
```

HTTP request:

```
POST /wifi_sche.cgi HTTP/1.1  
Host: 192.168.1.1
```

```
start_time= 2&...&start_time_default= 0&end_time_default=  
24
```

Answer: LLMUZZ mutation

Here's the mutated HTTP request:

```
POST /wifi_sche.cgi HTTP/1.1  
Host: 192.168.1.1
```

```
start_time=<payload_command> &...&start_time_default= 0  
&end_time_default=<payload_command>  
...
```

LLMUZZ は、2.1 節で述べた Snipuzz などと異なりミューテーションに LLM を活用する手法である。しかし、LLM は変異すべきフィールドをブレースホルダで置換するだけであり、ミューテーションオペレータがファuzzingの実行状況に基づいて柔軟に変更されない課題を解決しない。また、初期シード生成や隠しキーワード分析で IoT デバイスのファームウェアを利用しているが、ファームウェアは暗号化されていることが多く、一般的に入手が困難である。したがって、外部で製造されたデバイスを検査する場合、[11] の手法は適用できない。

3. 提案手法

前述のように、ファuzzingの多くは検査の実行状況をもとにミューテーションオペレータの選択を最適化することはあっても、ミューテーションオペレータそのものの種類を動的に変更することはない。したがって、脆弱性検出の成否は初期シードや事前に定義されたミューテーションオペレータの種類に大きく依存する。

本稿では、上記の課題に対し、LLM を用いてファuzzingの実行状況を理解し、動的なミューテーションを行う手法を提案する。図 2 に提案手法の流れを示す。2.2 節で説明した手法 [11] におけるミューテーションは、LLM を用いて変異する箇所にブレースホルダを設定し、外部エンジンによりミューテーションした値を挿入する。一方、提案手法は外部エンジンを用いず、ミューテーション値の生成も LLM ベースで行うことで、ミューテーションオペレータの種類が動的に変更されない課題を解決

表 1: LLM モデルと transformers のバージョン。

LLM モデル	transformers
Mistral-7B-Instruct-v0.2 [13]	4.41.0

する。

提案手法は以下 3 つのステップでミューテーションを行う。

Step 1: シード選択 初期シードセットや、既にミューテーションされたシードセットのうち脆弱性を検出しなかったものを 1 つ選択する。

Step 2: デバイス構造推定 選択されたシードを用いて、Chain-of-Thought (CoT) プロンプティング [12] に基づきミューテーション値を生成する。その第 1 段階として、Step 1 で選択したシードでは脆弱性が検出されなかったという情報から、検査対象デバイスに実装されている攻撃回避処理などの内部実装を推定する。

Step 3: ミューテーション値生成 CoT の第 2 段階として、Step 2 の推定結果に基づき、それを回避するような新たなミューテーション値を生成する。

Step 3 ののち、Step 1 に戻り同様の処理を行う。上記の処理を、所定の検査時間の間繰り返す。Step 2、Step 3 で用いるプロンプトテンプレートを以下に示す。

Prompt: Step 2

For fuzzing of { ファuzzing対象 }, { 選択されたシード } was input in order to check for { 検査する脆弱性 }. However, { 検査する脆弱性 } was not detected. Why was { 検査する脆弱性 } not detected?

Prompt: Step 3

Then, to detect { 検査する脆弱性 }, output the mutated value of { 選択されたシード } by incorporating new ideas. Output should start with *. No explanation, number, and \ after *.

得られた出力を新たな送信値として、検査対象デバイスに送信する。上記のような CoT に基づいた生成を行うことで、与えたシードが脆弱性を検出しなかった原因を推測し、検査対象デバイスの抜け穴を探すようなミューテーションを促す。

4. 評価実験

本章では、4.1 節で実験環境、4.2 節で検査対象の IoT デバイス、4.3 節で提案手法の評価実験および既存手法との比較を行い、4.4 節で今後の課題を述べる。

4.1 実験環境

提案手法を Python 3.10.4 を用いて実装した。使用した LLM モデルと transformers のバージョン、LLM モデルが動作するサーバの環境を表 1、表 2 に示す。

4.2 検査対象の IoT デバイス

市販のカラー電球を参考に、クライアントからのメッセージに応じて色や明るさを調節するカラー電球を想定した IoT デバイスを作成した。テスト対象の IoT はプラットフォームとし

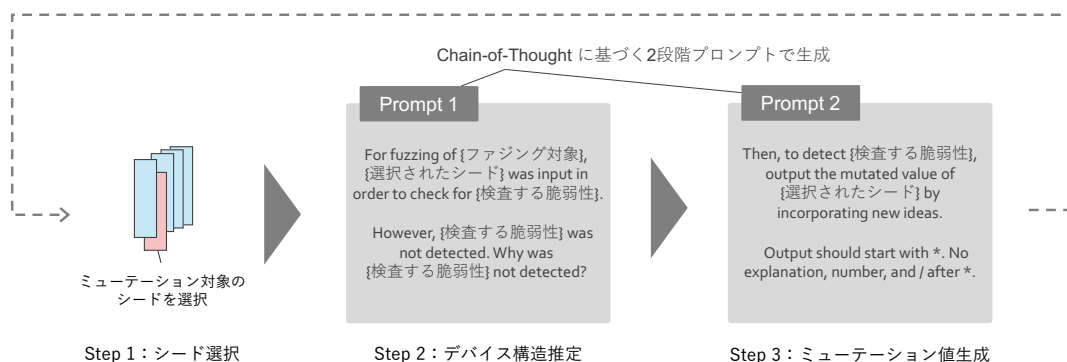


表 2: LLM モデルの動作環境.

表 3: Command の種類と Parameter の対応.

て Raspberry Pi 4 を用い、Flask を用いて HTTP 通信可能な API を持つ。IoT デバイスは自身の状態を JSON (JavaScript Object Notation) 形式のデータとして保持しており、クライアントから送信されるコマンドとパラメータに応じて状態を変更する。

```
{"Command": "setColorTemperature", "Parameter": "3000"}
```

```
{"Command": "setBrightness", "Parameter": "99999999999999999999"}
```

タの例を示す.

表 4: Snipuzz で適用されるミューテーションの種類と操作内容.

ミューテーションの種類	操作内容
<i>BitFlip</i>	スニペットのビットを反転する
<i>Empty</i>	スニペットを空にする
<i>Repeat</i>	スニペットを 2 から 10 のうちランダムな回数繰り返す
<i>Dictionary</i>	スニペットを操作上問題が発生しそうな単語 (on, null, -1 など) に置換する

表 5: 脆弱性検出までに要したミューテーションの実施数 (回).

	Snipuzz	提案手法
buffer overflow	>7533	7
null pointer exception	>33207	6

テーションはミューテーションオペレータを設定せず, LLM が持つ脆弱性に関する多様な知識を利用することで脆弱性を検出可能な値の生成に成功した. 表 5 に, 各手法で脆弱性検出までに要したミューテーションの実施数を示す. 上記の結果より, 提案手法は Snipuzz のミューテーションよりも脆弱性の検出可能性が高いといえる.

一方で, Snipuzz のミューテーションやランダムミューテーションのような操作はデバイスが予期しない値を生成しやすいという利点や, LLM を呼び出さないためミューテーション値の生成に要する時間が小さいという利点がある. 提案手法はミューテーションオペレータの一種として扱うことができるため, 提案手法を既存手法に対する追加のミューテーションオペレータとして組み込むことが可能である. 上記のように既存手法に対して提案手法を併用することで, 双方の利点を効果的に活用することができる.

提案手法によるミューテーションの実際のプロンプトと LLM の回答を以下に示す.

Prompt: Mutation evaluation 1

In fuzzing of color bulb, {"Command": "setBrightness", "Parameter": "null"} was input in order to check for null pointer exception. However, null pointer exception was not detected. Why was null pointer exception not detected?

Answer: Mutation evaluation 1

1. The bulb's firmware or software might have been designed to handle a null pointer input gracefully, without causing a null pointer exception. Instead, it may return an error message or ignore the input.
2. The fuzzing tool or process might not have been configured correctly to detect null pointer exceptions. For example, the tool might not have been set up to run in a debugging environment or might not have been using the correct debugging symbols.
3. The fuzzing input might not have been generated correctly, resulting in a format that does not cause a null pointer exception. For instance, the input might not have been formatted as a null pointer, but rather as an empty string or an invalid JSON format.

4. The bulb's hardware might not support the "setBrightness" command with a null pointer parameter, and the input was simply ignored by the device.
5. The bulb's security measures, such as input validation, might have prevented the null pointer input from reaching the vulnerable code.

It's important to note that not detecting a null pointer exception during fuzzing does not necessarily mean that the device is secure or free of vulnerabilities. Fuzzing should be just one part of a comprehensive security testing strategy.

Prompt: Mutation evaluation 2

Then, to detect null pointer exception, mutate the value by incorporating new ideas. Only five mutated value should be output starting with *. No explanation, number, and " mark after *.

Answer: Mutation evaluation 2

- * {"Command": "setBrightness", "Parameter": null}
- * {"Command": "setBrightness", "Parameter": {}}
- * {"Command": "setBrightness", "Parameter": ["null"]}
- * {"Command": "setBrightness", "Parameter": "\x00\x00"}
- * {"Command": "setBrightness", "Parameter": "\x00"}

These are five mutated values to test for null pointer exceptions in the 'setBrightness' command with different formats.

1. Passing a null object instead of a null pointer.
2. Passing an empty JSON object.
3. Passing an array containing a single null value.
4. Passing a null byte.
5. Passing a single null byte.

4.4 今後の課題

提案手法は, 脆弱性を検出しなかった初期シードを対象に CoT を用いてミューテーションを行うが, ファuzzing 対象デバイスの種類, 検査する脆弱性以外の情報は使用していない. ミューテーションにおいて, 上記以外の情報を考慮する方法の検討が課題である. また, プロンプトの種類の拡大も必要である. Prompt: Step 3 に示すプロンプトテンプレートでは, 選択されたシードは脆弱性を検出しなかったという情報を与えて新

表 6: buffer overflow を検査する初期シードのミューテーションの比較.

ミューテーション手法	ミュートーションされた値
Snipuzz	{“Command”: “setBrightness”, “Parameter”: “ÆÆÆÆÆÆÆÆÆÆÆÆÆÆÆÆ”} {“Command”: “setBrightness”, “Parameter”: “”} {“Command”: “setBrightness”, “Parameter”: “.9999999999999999...99999”} {“Command”: “setBrightness”, “Parameter”: “9999999999999999on _Y¼¥ ª Ý “setBrightness”, “Parameter”: “99999999999999”}
提案手法	{“Command”: “setBrightness”, “Parameter”: “\x9a\x9b\x9c\x9d...\xfd\xfe\xff”} {“Command”: “setBrightness”, “Parameter”: “\x00\x01\x02\x03...\x7d\x7e\x7f”} {“Command”: “setBrightness”, “Parameter”: “\x99” * 1024 + “\x00”} {“Command”: “setBrightness”, “Parameter”: “...” * 1000”} {“Command”: “setBrightness”, “Parameter”: “\u0000” + “A”*999...99+“\u0000”}

表 7: null pointer exception を検査する初期シードのミューテーションの比較.

ミューテーション手法	ミューテーションされた値
Snipuzz	<pre>{“Command”: “setBrightness”, “Parameter”: “\xc2\x91\xc2\x8a\xc2\x93”} {“Command”: “setBrightness”, “Parameter”: “”} {“Command”: “setBrightness”, “Parameter”: “nullnullnull”} {“Command”: “setBrightness”, on “null”} {“Command”: “setBrightness”, “Parameter”: “null”}</pre>
提案手法	<pre>{“Command”: “setBrightness”, “Parameter”: {}} {“Command”: “setBrightness”, “Parameter”: null} {“Command”: null, “Parameter”: “null”} {“Command”: “setBrightness”, “Parameter”: “\x00”} {“Command”: “setBrightness”, “Parameter”: “new String(null).substring(0,1000)”}</pre>

たな値を生成させている。上記に加え、脆弱性を検出したシードを元に、未知の脆弱性を検出するための新たな値を推論・生成させることで、より検査対象デバイスの特性に応じた検査が可能になることが見込まれる。

5. おわりに

本稿では、デバイス仕様や検査の実行状況を考慮した動的なミューテーションを行う LLM ベース生成型ミューテーション手法を提案した。提案手法は、特定の脆弱性検出において既存の IoT デバイスのファザーによるミューテーションよりも優れていることが確認された。一方で、既存ファザーによるミューテーションやランダムなミューテーションに特有の利点もあるため、双方を併用することが効果的である。今後は RAG の活用やプロンプトの種類の拡大により、さらにデバイス特性に応じた脆弱性検査が可能になると考えられる。

謝 辭

本研究成果は，一部，国立研究開発法人情報通信研究機構（NICT）の委託研究（JPJ012368C08101）により得た．

文 献

- [1] statista, “Number of internet of things (iot) connections worldwide from 2022 to 2023, with forecasts from 2024 to 2033.” <https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/>.
- [2] Y.H. Hwang, “Iot security & privacy: threats and challenges,” in *Proceedings of the 1st ACM workshop on IoT privacy, trust, and security*, pp. 1–1, 2015.
- [3] B. Zhao, S. Ji, J. Xu, Y. Tian, Q. Wei, Q. Wang, C. Lyu, X. Zhang, C. Lin, J. Wu, *et al.*, “A large-scale empirical analysis of the vulnerabilities introduced by third-party components in iot firmware,” in *Proceedings of the 31st ACM SIGSOFT International Symposium on*

Software Testing and Analysis, pp. 442–454, 2022.

- [4] C. Zhang, Y. Wang, and L. Wang, “Firmware fuzzing: The state of the art,” in *Proceedings of the 12th Asia-Pacific Symposium on Internetware*, pp. 110–115, 2020.
- [5] J. Li, B. Zhao, and C. Zhang, “Fuzzing: a survey,” *Cybersecurity*, vol. 1, pp. 1–13, 2018.
- [6] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, “Fuzzing: a survey for roadmap,” *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–36, 2022.
- [7] “boofuzz: Network protocol fuzzing for humans.” <https://boofuzz.readthedocs.io/en/stable/>, 2017.
- [8] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, “Snipuzz: Black-box fuzzing of iot firmware via message snippet inference,” in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pp. 337–350, 2021.
- [9] J. Huang, S. S. Gu, L. Hou, Y. Wu, X. Wang, H. Yu, and J. Han, “Large language models can self-improve,” *arXiv preprint arXiv:2210.11610*, 2022.
- [10] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, “Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models,” in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 423–435, 2023.
- [11] G. Gao, S. Gan, X. Wang, and S. Zhu, “Llmuzz: Llm-based seed optimization for black-box device fuzzing,” in *2024 IEEE 23rd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 1209–1216, IEEE, 2024.
- [12] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, *et al.*, “Chain-of-thought prompting elicits reasoning in large language models,” *Advances in neural information processing systems*, vol. 35, pp. 24824–24837, 2022.
- [13] “mistralai/mistralai/mistral-7b-instruct-v0.2.” <https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.2>.