電子情報学中間報告資料 2024 年 5 月 10 日

Leveraging eBPF to Uncover the Characteristics of Shadow Attack

落合研究室 電子情報学専攻 修士課程 2 年 48-236427 手塚 尚哉

Abstract

In the complex arena of cybersecurity, evasive malware, such as shadow attacks that obscure malicious activities through multiple processes, challenge conventional detection methods. This paper introduces an innovative detection and analysis methodology using Extended Berkeley Packet Filter (eBPF) technology to counteract these threats. eBPF facilitates real-time, in-depth monitoring of system operations, enabling the identification of sophisticated malware behaviors.

We leverage eBPF to trace process interactions and system calls, identifying malicious patterns indicative of shadow attacks. This approach distinguishes between legitimate and malicious activities by analyzing process executions and network communications. Despite challenges like data volume and behavior differentiation, we apply smart filtering and machine learning to enhance detection accuracy.

Our research showcases eBPF's potential in detecting complex malware through case studies, emphasizing the need for advanced analysis techniques in cybersecurity. This work contributes significantly to understanding and mitigating advanced malware threats, proving eBPF as a vital tool in modern cybersecurity defenses.

1 Introduction

In the contemporary digital landscape, the sophistication of malware, particularly in its ability to evade detection and analysis, poses a significant challenge to cybersecurity efforts. Among these evasive tactics, shadow attacks [1], which cleverly distribute malicious activities across multiple processes, stand out as particularly insidious. These attacks exploit the inherent complexity of operating systems, mimicking benign multi-process behavior to obfuscate their malicious intent. Traditional detection mechanisms, reliant on static and dynamic analysis techniques, often fall short in identifying these distributed threats, necessitating the exploration of more advanced methodologies.

This paper introduces an innovative approach to tackling the challenge posed by shadow attacks through the use of Extended Berkeley Packet Filter (eBPF). eBPF, a technology that allows for the safe execution of custom code within the Linux kernel without changing kernel source code or loading kernel modules, offers a powerful mechanism for monitoring and tracing system-level operations. Our research leverages eBPF to analyze the interconnections between function calls, thereby revealing the execution patterns of processes involved in shadow attacks. By mapping these patterns, we aim to uncover the stealthy operations of evasive malware, providing insights that could lead to more effective detection and mitigation strategies.

We focus on the potential of eBPF to provide granular visibility into the behavior of systems at runtime, enabling the identification of the complex orchestration of processes characteristic of shadow attacks. Through the detailed analysis of function call chains, we can trace the flow of execution within malicious processes, identifying their strategies and mechanisms. This approach not only enhances our understanding of how such attacks are constructed and executed but also opens new avenues for developing countermeasures that can detect and neutralize these threats more efficiently.

By employing eBPF to dissect the intricacies of process execution and interaction in the context of shadow attacks, our research contributes a novel perspective to the field of cybersecurity. We demonstrate how eBPF's capabilities can be harnessed to advance our understanding of malicious process execution, offering a promising methodology for combatting evasive malware. Through this work, we

aim to bolster the cybersecurity community's arsenal against the ever-evolving landscape of malware threats, ensuring a more secure digital environment for all users.

Our exploration into the use of eBPF against shadow attacks not only highlights the adaptability and complexity of modern malware but also underscores the necessity for innovative detection and analysis techniques. As we delve into the capabilities and applications of eBPF, we pave the way for future research and development in the domain of cybersecurity, seeking to establish more sophisticated defenses against the cunning and elusive nature of malware attacks.

2 eBPF

2.1 Berkeley Packet Filter

Steven と Van [2] は 1993 年に，Unix 系の OS 上でパケットキャプチャを効率的に行うためのアーキテクチャである BSD Packet Filter を提案した．以下，Berkeley Packet Filter を「BPF」と表記する．
論文発表当時のパケットキャプチャでは，カーネル空間で取得したパケットをすべてユーザー空間にコピーしてからフィルタリングしていて，これが無駄なオーバーヘッドの原因となっていた．[2] は特殊な 32 ビット命令で記述されたプログラムを解釈してフィルタリングを行う疑似マシン (BPF pseudo-machine) を考案し，それをカーネル空間で動作させることでこの問題に対処した．既存のシステムとの比較では，BPF は最大で 20 倍程度高速に動作した．BPF のアーキテクチャの概要を Fig. 1 に示す．

BPF は Linux カーネルの v2.1.75 にて Linux Socket Filter という名前で導入され，`tcpdump` コマンドの高速化のために使われた．

2.2 BPF の拡張

BPF は Linux カーネルの v3.18 にて大幅に改修，拡張が行われ，extended BPF すなわち eBPF と呼ばれるようになった [3]．拡張された部分は多岐にわたるが，代表的なものを以下に列挙する [4]．

- BPF 命令セットが 32 ビットから 64 ビットに書き直され，実行効率が向上した．
- eBPF map が導入され，ユーザー空間とカーネル空間の間でデータを共有する手段が追加された．
- eBPF プログラムが安全に実行できることを検証する eBPF verifier が追加された．

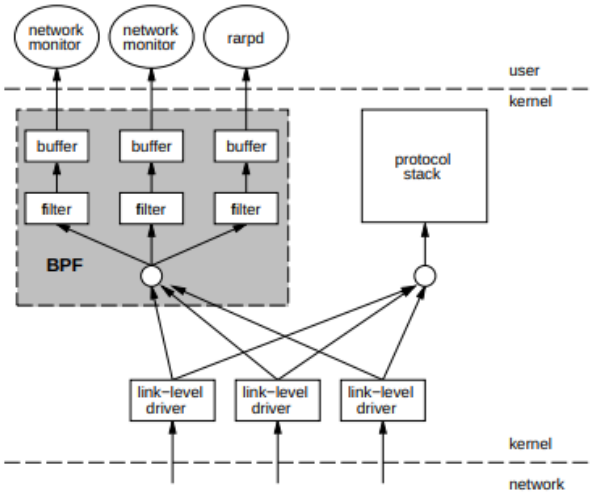

Fig. 1: The overview of BPF architecture. It accelerates packet capture by performing filtering in the kernel space. [2]

また，eBPF がカバーする領域も拡大していった．ネットワークの文脈では，Linux のネットワークスタックの様々なレイヤー，例えば Unix socket やネットワークデバイスなどを扱えるようになった．加えて eBPF プログラムは Linux システムのパフォーマンストレーシングやセキュリティ向上にも利用できるようになり，”BPF”という単語は本来の意味である”Berkeley Packet Filter”を失い独立した単語として使われるようになっていった．

便宜上，v3.18 における拡張以前の BPF を classical BPF，あるいは cBPF と呼ぶことがある．

2.3 eBPF のアーキテクチャ

eBPF のアーキテクチャの概要を Fig. 2 に示す．以下，Fig. 2 を参照しながら重要な処理フローについて説明する．

2.3.1 Event-Driven Architecture

eBPF は event-driven である．カーネル内でのイベントに eBPF プログラムをフックし，イベントが発生したら所定の処理を行う，という仕組みである．eBPF プログラムは動的にロードまたは削除することが可能である．

eBPF がフックできるイベントはカーネルのソースコード [*1] において Program Type として定義されている．Program Type をイベントの種別ごとに分類す

[*1] `include/uapi/linux/bpf.h`

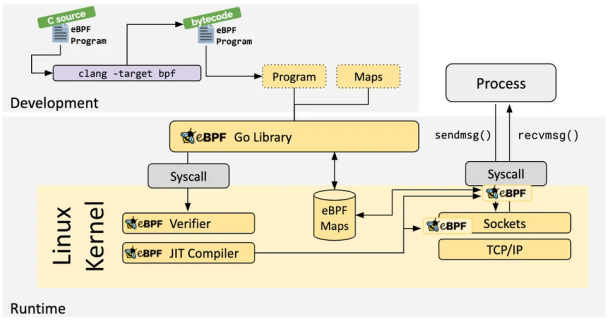

Fig. 2: The overview of eBPF architecture. This figure shows how eBPF programs are compiled, verified, and executed. [5]

ると，以下のようなものが挙げられる．

- XDP: ネットワークデバイスにパケットが到着したとき，カーネル空間にデータがコピーされる前にパケットを操作するためのイベント．
- tracing: カーネル関数の呼び出しやトレースポイントの通過を検知するためのイベント．
- LSM: Linux Security Module を利用してセキュリティポリシーを適用するためのイベント．

このようなイベントが発生したとき，eBPF プログラムは Program Type に応じた処理を実行する．例えば XDP のイベントにフックされているプログラムがパケットを accept するか drop するかを判断する，といった処理が可能である．

2.3.2 eBPF Verifier

eBPF verifier はバイトコードに変換された eBPF プログラムを入力とし，そのプログラムがカーネル上で安全に実行できることを検証するプログラムである．バイトコードは `bpf()` システムコールでカーネル上にロードされる (Fig. 2 の”Syscall”) が，verifier による検証を通過しない限りプログラムは実行されない．具体的には，メモリアクセス違反を起こさないことやプログラムが正常に終了すること，不必要な privilege がプログラムに与えられていないことなどが確認されている．

このように，eBPF verifier は eBPF プログラムに制約を課すことで安全性を高めている．verifier が eBPF において重要な意味を持つことから，verifier のロジックを数学的に検証する研究 [6] も行われている．

2.3.3 JIT コンパイル

verifier を通過した eBPF バイトコードは JIT コンパイラによってターゲットの CPU 上で直接動作する機械語に変換される．これにより実行速度が最適化され，ソースコードから直接コンパイルされたカーネルおよびカーネルモジュールと同程度効率的に動作する．

3 関連分野

3.1 カーネルモジュール

カーネルモジュールは Linux カーネル実行時にオブジェクトファイルをロードすることで，カーネルのソースコードを変更せずにカーネルの機能を拡張する仕組みの 1 つである．カーネルモジュールには eBPF verifier のような制約がないため，プログラムの自由度は高い．しかしカーネルモジュールはカーネルと同じ privilege で実行されるので，脆弱性を埋め込まないように注意して開発を行う必要がある [7].

Mayer ら [8] が指摘するように，カーネルモジュールの作成を避けることができるのは eBPF の長所であると言える．

3.2 Linux Security Modules

LSM (Linux Security Modules) は Linux カーネルのセキュリティポリシーを実装するためのフレームワークである．LSM はアクセス制御に関するカーネル内の処理にフックポイントを提供し，そのポイントにセキュリティポリシーの具体的な実装であるセキュリティモジュールを組み込むことができる[*2]．カーネルがフックポイントを通過すると，セキュリティモジュールへのコールバックが発行されて強制的にアクセス制御が行われる．LSM の処理フローの概要を Fig. 3 に示す．LSM は Linux の capability や SELinux で利用されている [10].

カーネル v5.17 から LSM BPF という機能が追加され，eBPF プログラムを LSM が提供する API にフックすることができるようになった．

3.3 Seccomp

Seccomp は Linux で採用されているセキュリティシステムの 1 つで，あるプロセスが発行することができるシステムコールを制限する．v6.6 時点で Seccomp の実装は cBPF (eBPF ではない) に依存しており，cBPF を用いてシステムコールのフィルターをプロセスごとに変更することができる．この機能を seccomp-bpf と呼ぶ．

Seccomp は Docker [11] や Kubernetes [12] などの仮想化技術で広く用いられている．

[*2] セキュリティモジュールはカーネルモジュールのように動的にロードまたはアンロードすることができず，カーネルのビルド時に静的にリンクする必要がある．

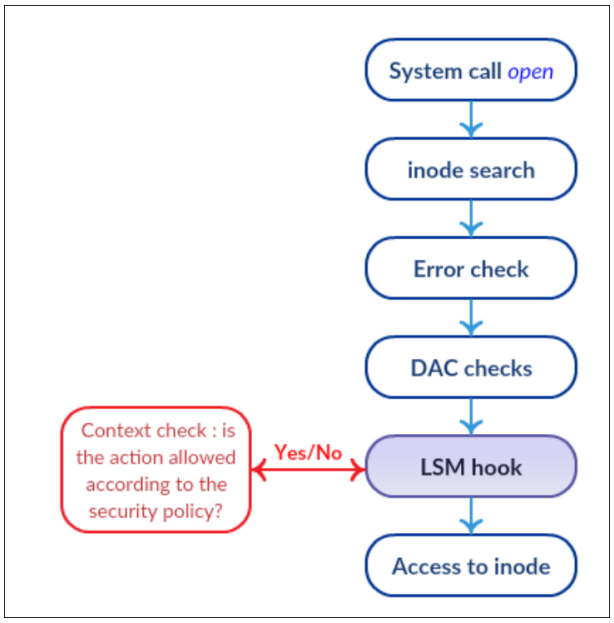

Fig. 3: The process flow of LSM when `open()` system call is issued. [9]

3.4 クラウドネイティブ

クラウドネイティブとは，クラウド基盤を前提としてアプリケーションやシステムを構築および管理するアプローチのことである．物理インフラの調達とメンテナンスのコストが不要になるほか，アプリケーションの開発やデプロイが効率的になる点，高い可用性が確保できる点がメリットとして挙げられる．

eBPF はクラウドネイティブ環境で幅広く使われている技術で，代表的なプロダクトとして Cilium [13] がある．Cilium によってコンテナ間の L3 通信を制御したり，セキュリティ機能や可観測性をコンテナ間パケットに付与したりすることができる．

eBPF は以下のような理由でクラウドネイティブ技術とよく適合する．

- ネットワークやシステムコール，アプリケーションのイベントをリアルタイムで監視することができる．
- カーネル空間で実行されるためオーバーヘッドが小さく，スケーラビリティを損ねない．
- 動的なロードまたはアンロードが可能で，システムの継続的インテグレーション/デリバリーを容易にする．

4 既存研究の取り組み

これまでに述べたように，eBPF の応用範囲はパケットキャプチャにとどまらずネットワークスタック全般やシステムの監視にまで広がっている．本章では，eBPF を活用したセキュリティシステムの中でも，ネットワーク領域の事例とそれ以外の領域，そして両方に跨る事例をそれぞれ紹介する．

4.1 DNS クエリのプライバシーの向上 [14]

この研究では，DNS クエリのプライバシーを毀損する攻撃についてまとめつつ，それらを緩和する方法の実現が既存のシステムでは難しいことを主張している．そして eBPF を利用して攻撃への対策を効率的に実施する手法を提案している．

4.1.1 問題

DoT (DNS over TLS) や DoH (DNS over HTTPS) といった暗号化プロトコルは，DNS クエリを暗号化することでネットワーク上の攻撃者がクエリを盗聴することを防いでいる．しかし DoT や DoH で暗号化されるのはクライアントとキャッシュサーバの間の通信のみであり，キャッシュサーバ上でクエリは復号化される．したがってキャッシュサーバの運営者，典型的には ISP，はユーザとクエリ情報を結びつけることができるため，クエリ情報からユーザのプロファイルを作成する de-anonymization を試みる危険性がある．

RFC 7626 は de-anonymization への対策の 1 つとして，アプリケーションごとにキャッシュサーバを変更することを推奨している．これによりクエリの情報が分散するので，プロファイルの作成が難しくなる．

上記の対策を実施する方法は手動で DNS クライアントを設定する以外に存在せず，時間がかかる作業である上に実行効率も悪いという問題点がある．

4.1.2 提案手法

この研究では，XDP で DNS または DoT のクエリパケットの宛先アドレスを編集することでクエリのリダイレクトを実行している．提案手法による DNS クエリの処理フローを Fig. 4 に示す．XDP にフックされた eBPF プログラムはあるパケットがどのプロセスから送出されるかを認識しており，key: プロセス名，value: 宛先のハッシュマップを参照してクエリのアドレスを書き換える．なお，この研究は DNS クエリを振り分けるキャッシュサーバのリストを指定できる user-friendly なインタフェースを提供している．

DoT では TLS で規定された処理を行う必要があるため，AF Socket という XDP の機能を使う．先述

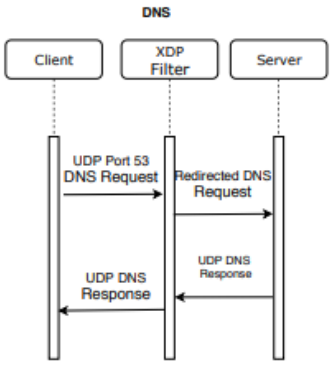

Fig. 4: DNS queries are intercept in XDP and eBPF implementation edits their destination. [14]

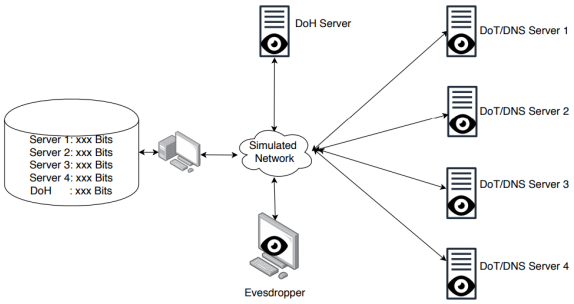

Figure 3: Experimental Network overview

Fig. 5: The overview of the experimental network.

したように宛先アドレスを編集したあと，AF Socket で作られた仮想ソケットにパケットを送信して TCP handshake や TLS の鍵交換を行う．

DoH においては，パケットがカーネルのネットワークスタックに到達する時点でクエリ情報は暗号化されているため，平文の DNS や DoT と同様の方法を適用できない．そこでこの研究では，uProbes (user probes) に eBPF プログラムをフックして DoH クエリの作成とレスポンスの復号化を監視した．uProbes ではデータが read-only であるためクエリの編集ができないが，その代わりにクエリの情報があるキャッシュサーバにどの程度知られているかを計算し，閾値を超えたときにユーザに警告する実装を行った．

4.1.3 評価

Rivera らは Fig. 5 に示す実験用ネットワークを構築して提案手法の評価を行った．

パフォーマンスの評価では DNS と DoT における名前解決時間を計測した．その結果 DNS では 0.44%，DoT では 8.15% のオーバーヘッドが観測された．DoT でのオーバーヘッドが大きいのは，AF Socket の処理がボトルネックになっている可能性がある．DoH クエリの監視については CPU サイクル数を計測し，3.13% のオーバーヘッドが見られた．

プライバシーの評価では，Fig. 5 に示すキャッシュサーバ 4 台のそれぞれに何ビットのクエリ情報が流出しているかを計算し，de-anonymization が実行可能であるかどうかを評価した．DoT 単体では ISP による de-anonymization が実行可能であったが，DoT と eBPF を組み合わせた提案手法はクエリを異なるサーバにロードバランスすることで de-anonymization を防ぐことができることが示された．

4.2 Programmability の高いシステムコールフィルタ [15]

本研究は Linux のシステムコールのフィルタリング機構である seccomp の問題点を論じた．そして seccomp-bpf を eBPF に対応させることでフィルタに高い programmability を与え，既存の seccomp-bpf よりも厳密なセキュリティポリシーを適用するためのデザインの提案と実装を行った．

4.2.1 問題

システムコールフィルタとは，システムコールのエントリーポイントにおいてそのシステムコールが発行されることを許可または拒否するメカニズムのことを指す．seccomp-bpf はプロセスごとに異なるフィルタを cBPF で記述して適用することができ，カーネル空間で実行されるためパフォーマンスが優れている．

しかし cBPF がもつ以下の性質により，seccomp-bpf はプロセスごとに静的な allow list を適用することしかできない．

- ステートレスである
- プログラムサイズの上限が小さい (4096 命令)

このため，本来許可するべきではないシステムコールを許可せざるを得なくなってセキュリティが劣化したり，複雑なルールを複数のフィルタのチェインで表現しなければならないことで実行効率が低下したりする．

seccomp-bpf の機能を補うために seccomp notifier [16] が提案された．notifier はシステムコールが発行されるたびに実行コンテキストをユーザー空間からカーネル空間に移し，そのシステムコールを許可または拒否する判断を行う仕組みである．フィルタの programmability は高いものの，コンテキストスイッチに起因するオーバーヘッドが大きい．

4.2.2 デザインと実装

本研究では既存の seccomp-bpf を seccomp-cBPF，提案手法を seccomp-eBPF と呼び区別している．seccomp-cBPF に欠落しているがシステムコールフィルタが有していると望ましい性質として，Jie らは 4 点を挙げていた．

1 点目は statefulness である．stateful なフィルタはシステムコールの発行回数や頻度を記憶してフィルタを運用することができる．また，プログラムの実行フェイズに応じて許可するシステムコールの集合を変化させる precise temporal specialization も可能になる．

2 点目は expressiveness である．無駄のないフィルタを記述することでフィルタの効率が改善する．

3 点目は synchronization である．システムコールの競合状態を利用する脆弱性が報告されており，そのような脆弱性の原因となるシステムコールが並列に実行される場合に同期を取って競合状態を回避する．

4 点目は safe user memory access である．システムコールの引数にポインタ型の変数が渡される場合，そのポインタが指す構造体を検査することはフィルタの意思決定において有用である．例えば `open()` システムコールに渡されるポインタは開こうとしているファイル名の文字列を指している．

Jie らはこれら 4 つの性質を満たしつつ，提案システムへの移行を容易にするためにの seccomp-cBPF が想定する脅威モデルを維持する実装を行った．seccomp-eBPF の実装で特筆すべき点は以下の通り．

- `BPF_PROG_TYPE_SECCOMP` を Program Type に追加した．
- 既存のヘルパー関数を改修，あるいは新規の関数を追加した．
- 上記 2 点を検証するために eBPF verifier の検証ロジックを変更した．

4.2.3 評価

seccomp-eBPF の機能の 1 つである precise temporal specialization の性能評価を行った．実験では Fig. 6 に示すようにサーバアプリケーションの実行フェーズを初期化フェーズと機能フェーズに分け，それぞれのフェーズでのみ実行されるシステムコールの集合 S_{init} と S_{serv} を seccomp-eBPF のフィルタで許可した．これらのフィルタはプログラムの実行フェーズに応じて動的に適用された．NGINX や Memcached といったプログラムを用いて評価したところ，初期化フェーズにおいて 33.6% から 55.4% の attack surface

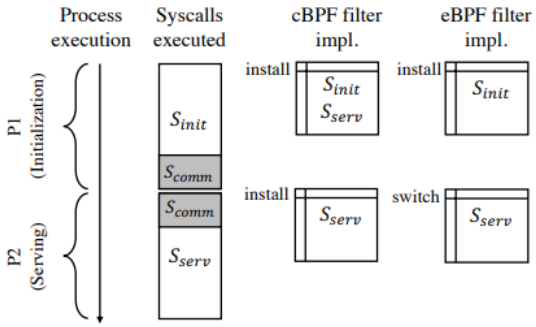

Fig. 6: The concept of precise temporal specialization. It should be noted that the seccomp-cBPF filter in the initialization phase allows S_{serv}.

	getppid (cycles)	filter (cycles)
No filter	244.18	0
cBPF filter (default)	493.06	214.19
cBPF filter (optimized)	329.47	68.68
eBPF filter	331.73	60.18
Constant-action bitmap [31]	297.60	0
Seccomp notifier	15045.05	59.29

Fig. 7: Execution time of `getpid()` with different filters and filter execution time.

の削減が見られた．

さらに，precise temporal specialization 以外に追加されたセキュリティ機能も評価された．システムコールの実行回数を制限する count limiting は CVE-2016-0728 などの脆弱性をブロックし，また serialization の機能は CVE-2018-18281 を防ぐことに成功した．

パフォーマンスの評価では，`getpid()` システムコールを実行するマイクロベンチマークと，サーバアプリケーションのベンチマークによるマクロベンチマークを扱った．

マイクロベンチマークの結果を Fig. 7 に示す．seccomp-eBPF のフィルタは，seccomp-cBPF のフィルタよりもセキュリティが向上しているにも関わらず，コンパイル時最適化を適用した seccomp-cBPF のフィルタと同程度のパフォーマンスを出した．notifier のフィルタはシステムコール自体の実行時間を 45 倍程度大きくし，顕著な性能低下が見られた．

マクロベンチマークの結果を Fig. 8 に示す．図中の”hybrid”は seccomp-cBPF と notifier を組み合わせたフィルタである．同じ機能を実装したフィルタを適用した条件で，Fig. 8 に示されるサーバアプリケーショ

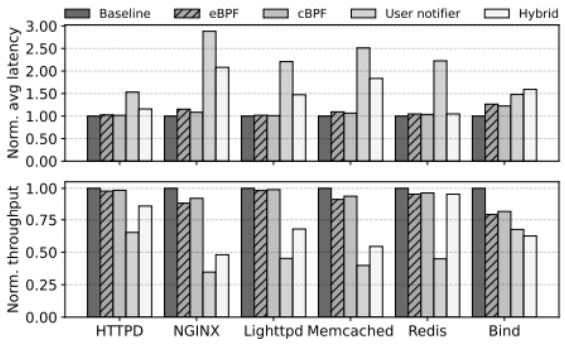

Fig. 8: Execution time of `getpid()` with different filters and filter execution time.

ンを公式のベンチマークを用いてレイテンシとスループットを評価した．アプリケーションごとに差異は見られるが，マイクロベンチマークの結果と同様の傾向が見られた．

4.3 周辺機器から Linux を保護するフレームワーク [17]

本研究は BadUSB などの悪意ある周辺機器を用いた攻撃や，周辺機器のプロトコルスタックのバグを突いた攻撃から Linux を保護するためのフレームワークを提案した．

4.3.1 問題

本研究が対象としている周辺機器は USB，Bluetooth，NFC の 3 種類である．周辺機器は各々のプロトコルに則ったパケットをカーネルとやり取りすることで動作するため，attack surface として悪意あるパケットを送信する方法と，カーネル側のプロトコルスタックの脆弱性を攻撃する方法が考えられる．

しかしながら，USB を対象とした攻撃には系統的な対策が存在する [18] が，Bluetooth や NFC には存在しない．したがって，様々な周辺機器に対応することができる包括的なセキュリティフレームワークが必要である．

4.3.2 デザインと実装

この研究では，様々なプロトコルを扱うことができる拡張性の高いフレームワークとして LBM，Linux (e)BPF Modules を提案した．そして，サポートする周辺機器のすべての入力と出力を検査すること，新しい周辺機器プロトコルのサポートが容易であること，オーバーヘッドの小さいことなどを含む LBM が備えるべき 7 つの性質を整理した．

Tian らはスタンドアローンなコンポーネントとして LBM を実装した．Fig. 9 にカーネル側のシステム

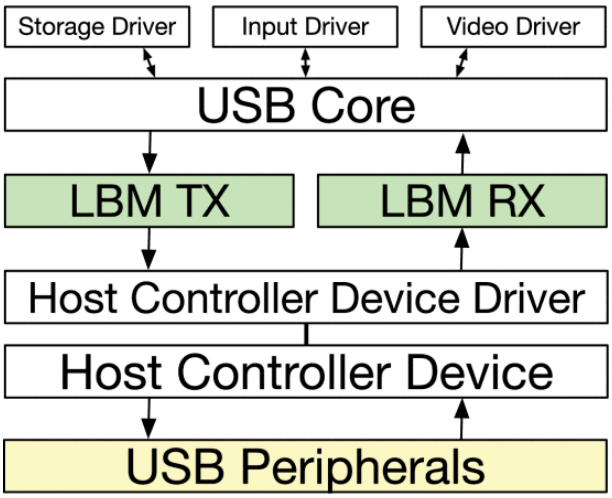

Fig. 9: The overview of LBM in kernel side. [17]

の概要を示す．eBPF でフィルタを適用するにあたって，eBPF プログラムのフックに 2 つのルールを定めている．

1. eBPF プログラムをハードウェアのなるべく近くにフックする (フィルタが回避されることを防ぐため)．
2. 特定のハードウェアの実装に依存しない (拡張性を高めるため)．

このルールを満たすために，LBM TX (周辺機器への入力) と LBM RX (周辺機器からの出力) のフックポイントはプロトコルスタックの下かつ Linux ホストのコントローラーデバイスの上に配置された．

LBM を利用するユーザには LBMTOOL というフロントエンドを提供した．ユーザが PCAP に似たドメイン固有言語でフィルタのルールを記述すると，LBMTOOL はそれをコンパイルして eBPF プログラムを生成し，フックポイントにロードする．

4.3.3 評価

LBM の評価にあたって，本研究では既知の攻撃を実施するケーススタディとベンチマークによるパフォーマンス計測を行った．

まずケーススタディでは，レスポンスパケットの先頭 2 バイトからパケット長をチェックすることでスタックを保護するフィルタを記述することができた．また USB 充電器を装った BadUSB を充電は妨害せずにブロックできること，Bluetooth の脆弱性の一つである BlueBorne [19] を 10 行程度のルールで対策できることを示した．さらに概念実証として LBM による NFC のサポートを行った．その結果カーネルと LBMTOOL

	getppid (cycles)	filter (cycles)
No filter	244.18	0
cBPF filter (default)	493.06	214.19
cBPF filter (optimized)	329.47	68.68
eBPF filter	331.73	60.18
Constant-action bitmap [31]	297.60	0
Seccomp notifier	15045.05	59.29

Fig. 10: LBM overhead in μs measured on 10K packets on the RX path. For each subsystem, the 1st row is for normal LBM and the 2nd row is for LBM-JIT. [17]

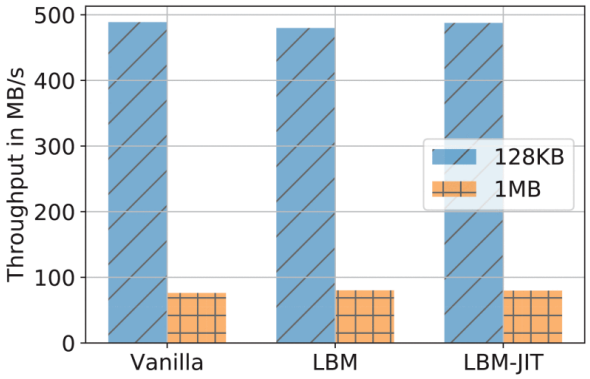

Fig. 11: Throughput of USB 3.0 external storage with filebench. [17]

へのソースコードの変更は合計で 100 行未満であり，拡張性の高さを表す根拠が得られた．

ベンチマークによるパフォーマンス計測では，マイクロベンチマークとマクロベンチマークが使用された．マイクロベンチマークは RX パスで 10,000 のパケットをカウントし，LBM がもたらすオーバーヘッドを計測した．その結果を Fig. 10 に示す．Fig. 10 によると，JIT コンパイル最適化が行われた場合 LBM のオーバーヘッドは，パケットあたり $1\mu s$ 程度であった．

マクロベンチマークでは，filebench というベンチマークを採用して USB3.0 外部ストレージへのファイル書き込みのスループットを計測した．その結果を Fig. 11 に示す．Vanilla は LBM が組み込まれていない純正の Linux カーネルを意味している．いずれの条件でも同様のスループットが得られており，LBM のオーバーヘッドが小さく抑えられていることを示している．なお，平均ファイルサイズが大きくなるとスループットが極端に落ちる現象はページキャッシュサイズに由来する．

さらに Bluetooth に対しても `l2ping` による RTT の計測が行われた．Fig. 11 の 3 種類の設定で RTT を計測したところいずれも $5ms$ 付近の結果が得られた．LBM による 1 パケットあたりの遅延は $1\mu s$ であったから，RTT と比較すると遅延は無視できる範囲に収まった．

5 おわりに

本稿では，eBPF という技術の発展の歴史，アーキテクチャ，主要なユースケースについて述べた後，eBPF を活用したセキュリティシステムを提案する既存研究の取り組みを紹介した．eBPF が提供するシステムの可観測性，セキュリティ，高パフォーマンスはセキュリティ技術者および研究者にとって強力な武器であり，これまで以上に多岐にわたる問題に対処できるようになったことがわかる．

eBPF は比較的新しい技術であり，既に広く普及していながらも未だ発展途上である．今後もクラウドネイティブ環境を中心に新規のユースケースが提案されることが期待される．

参考文献

[1] Weiqin Ma, Pu Duan, Sanmin Liu, Guofei Gu, and Jyh-Charn Liu. Shadow attacks: automatically evading system-call-behavior based malware detection. *Journal in Computer Virology and Hacking Techniques*, 8:1–13, 2012.

[2] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 46, pages 259–270, 1993.

[3] Linux_3.18 - linux kernel newbies. `https://kernelnewbies.org/Linux_3.18`. (Accessed on 11/05/2023).

[4] Liz Rice. Learning ebpf. O'Reilly, March 2023.

[5] What is ebpf? an introduction and deep dive into the ebpf technology. `https://ebpf.io/what-is-ebpf/`. (Accessed on 11/05/2023).

[6] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Verifying the verifier: ebpf range analysis verification. In *International Conference on Computer Aided Verification*, pages 226–251. Springer, 2023.

[7] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, pages 1–5, 2011.

[8] Andrea Mayer, Pierpaolo Loreti, Lorenzo Bracciale, Paolo Lungaroni, Stefano Salsano, and Clarence Filsfils. Performance monitoring with h^2: Hybrid kernel/ebpf data plane for srv6 based hybrid sdn. *Computer Networks*, 185:107705, 2021.

[9] Linux security modules (part 1). `https://thibaut.sautereau.fr/2017/05/26/linux-security-modules-part-1/`. (Accessed on 11/05/2023).

[10] The Linux Kernel Archive. Linux security module usage — the linux kernel documentation. https://www.kernel.org/doc/html/v4.16/admin-guide/LSM/index.html. (Accessed on 11/05/2023).

[11] Seccomp security profiles for docker — docker docs. https://docs.docker.com/engine/security/seccomp/.

[12] Kubernetes Documentation. Configure a security context for a pod or container — kubernetes. https://kubernetes.io/docs/tasks/configure-pod-container/security-context/, July 2019.

[13] Cilium - cloud native, ebpf-based networking, observability, and security. https://cilium.io/.

[14] Sean Rivera, Vijay K Gurbani, Sofiane Lagraa, Antonio Ken Iannillo, and Radu State. Leveraging ebpf to preserve user privacy for dns, dot, and doh queries. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, pages 1–10, 2020.

[15] Jinghao Jia, YiFei Zhu, Dan Williams, Andrea Arcangeli, Claudio Canella, Hubertus Franke, Tobin Feldman-Fitzthum, Dimitrios Skarlatos, Daniel Gruss, and Tianyin Xu. Programmable system call security with ebpf. *arXiv preprint arXiv:2302.10366*, 2023.

[16] Christian Brauner. The seccomp notifier - new frontiers in unprivileged container development — personal blog of christian brauner. https://brauner.io/2020/07/23/seccomp-notify.html, July 2020.

[17] Dave Jing Tian, Grant Hernandez, Joseph I Choi, Vanessa Frost, Peter C Johnson, and Kevin RB Butler. Lbm: A security framework for peripherals within the linux kernel. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 967–984. IEEE, 2019.

[18] Dave Jing Tian, Nolen Scaife, Adam Bates, Kevin Butler, and Patrick Traynor. Making {USB} great again with {USBFILTER}. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 415–430, 2016.

[19] Blueborne — armis. https://www.armis.com/research/blueborne/.