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 Related Work

2.1 Malware Analysis

Malware analysis is a critical aspect of cybersecurity, enabling the identification and mitigation of malicious software threats. Analysis techniques can be roughly categorized into two classes: static and dynamic.

2.1.1 Static Analysis

Static analysis involves examining the structure and content of a suspected malicious file without executing it. Static analysis are often conducted as a preliminary step to grasp the general characteristics of the malware before more sophisticated analysis techniques are applied [2].

Signature matching is a common technique that compares the "signature", characteristics of a file against a database of malware such as VirusTotal [3]. The characteristics include file size, hash values, and byte sequences. Although this method has been a staple in malware detection, it is limited by its inability to detect only known malware.

Disassemble and decompile is another static analysis technique that involves converting the binary code of a malware into a human-readable format. From the decompiled code, analysts might be able to identify the malware's functionality and behavior as well as finding interesting strings like URLs, IP addresses, and encryption keys.

2.2 Dynamic Analysis [4]

Dynamic analysis entails examining the behavior of malware by executing it in a controlled environment. This approach enables analysts to witness the interactions between the malware and the system, providing insights into its true intentions and capabilities.

Function call analysis is a method that focuses on tracking functions issued by the malware and the parameters passed to them. One way to archive this is by code injection, in which analyzing code is hooked into a specific function call and various information is collected and notified when the function is called. Carsten et al. [5] created an automated malware analysis system that injects DLLs within CWSandbox, letting analysts monitor system calls.

Data flow tracking is another approach that tracks the flow of data through the malware, and data tainting is an established technique in this area. It involves marking (or "tainting") specific data components and then monitoring how this tainted data propagates through the system. Data tainting could be utilized in static analysis, but due to some evasion strategies like encryption and obfuscation it has endured challenges in practice [6]. SELECTIVE-TAINT [7] was invented to address performance overhead issue by employing static binary rewriting to selectively instrument only instructions related to taint analysis.

2.3 Shadow Attack [1]

Shadow Attack is one of the techniques used by malware writers to evade behavior-based detection systems, orchestrating multiple processes to stealthily carry out malicious activities.

Such detectors typically rely on comparing system call graph within a process under scrutiny with predefined malware specifications established on specific sequences or graphs of system calls [8]. For example, the malware specification of download-and-execute is expressed as follows [1]:

$$\texttt{recv} \wedge \texttt{open} \rightarrow \texttt{write} \rightarrow \texttt{exec} \tag{1}$$

, where $s_1 \wedge s_2$ denotes both of two system calls s_1, s_2 are executed and $s_1 \rightarrow s_2$ denotes s_1 is followed by s_2.

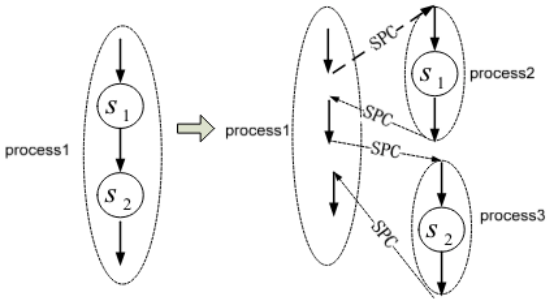

Fig. 1: Illustration of Shadow Attacks [1]

So more specifically, the goal of Shadow Attack is to bypass dynamic malware detection based on the analysis of system call graphs by exporting any critical system call included in malware specifications to other collaborating processes, which is called shadow processes. We call the Communication shadow processes do with each other as Shadow Process Communication (SPC). The concept of Shadow Attack is illustrated in Fig. 1.

Shadow attacks can be categorized into in-host, remote-network-coordinated, and hybrid. In in-host shadow attacks, all shadow processes are executed on the same host and SPCs are conducted through unix domain socket or stream pipe, while remote-network-coordinated shadow attacks involve multiple remote hosts that are connected through network sockets. Prototype implementation is made by the authors only for in-host shadow attacks.

One countermeasure is extracting correlation between processes and reconstructing the original system call graph, but the authors conducted an evaluation experiment following this approach, whose result suggested that solution would encounter challenges of high overhead.

2.4 eBPF

2.4.1 Berkeley Packet Filter

Steven and Van [9] proposed the BSD Packet Filter architecture in 1993 for efficient packet capture on Unix-based operating systems. In the following, we refer to Berkeley Packet Filter as "BPF." At the time of paper publication, packet capture involved copying all packets acquired in the kernel space to the user space before filtering. This process resulted in unnecessary overhead. [9] devised a

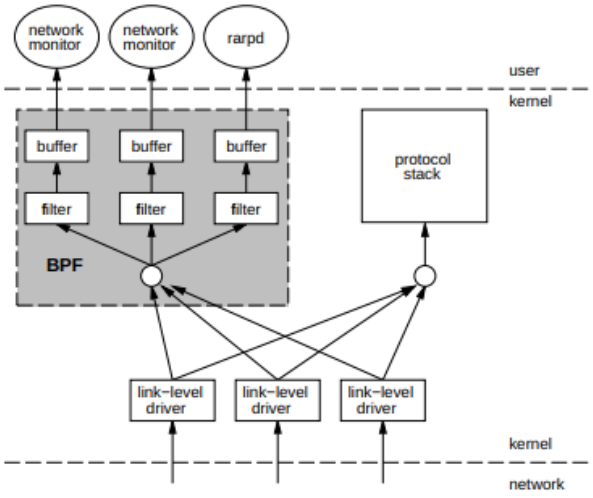

Fig. 2: The overview of BPF architecture. It accelerates packet capture by performing filtering in the kernel space. [9]

pseudo-machine (BPF pseudo-machine) that interprets programs written in special 32-bit instructions to perform filtering. By running this pseudo-machine in the kernel space, they addressed the issue. Compared to existing systems, BPF operated up to 20 times faster. The overview of BPF architecture is shown in Figure 2.

BPF was introduced in the Linux kernel as "Linux Socket Filter" in version 2.1.75 and was used to accelerate the `tcpdump` command.

2.4.2 Extended Berkeley Packet Filter

BPF underwent significant improvements and extensions in the Linux kernel version 3.18, leading to the emergence of extended BPF, commonly referred to as eBPF [10]. The enhancements cover various aspects, with notable additions summarized below [11]:

- 64-Bit BPF Instruction Set: The BPF instruction set was reworked from 32-bit to 64-bit, resulting in improved execution efficiency.
- eBPF Maps: The introduction of eBPF maps allowed data sharing between user space and kernel space. These maps serve as a mechanism for efficient communication.
- eBPF Verifier: To ensure safe execution of eBPF programs, an eBPF verifier was added. It validates the correctness and security of eBPF code.

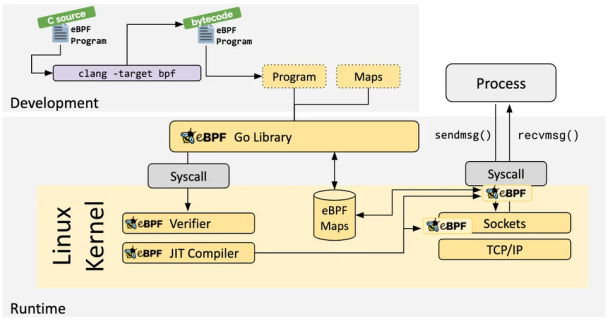

Fig. 3: The overview of eBPF architecture. This figure shows how eBPF programs are compiled, verified, and executed. [12]

The area covered by eBPF has also expanded. In the context of networking, it has become able to handle various layers of the Linux network stack, such as unix domain sockets and network devices. Additionally, eBPF programs can now be used for performance tracing and enhancing the security of Linux systems, leading to the term "BPF" losing its original meaning of "Berkeley Packet Filter" and being used as an independent term.

For convenience, BPF before the extension in v3.18 is sometimes referred to as classical BPF or cBPF.

2.5 Overview of eBPF Architecture

An overview of the eBPF architecture is shown in Fig. 3. Hereafter, we describe the important processing flows with reference to Fig. 3.

2.5.1 Event-Driven Architecture

eBPF has an event-driven architecture. The mechanism involves hooking an eBPF program to an event in the kernel, and then performing the specified processing when the event occurs. eBPF programs can be dynamically loaded or removed.

The events that eBPF can hook into are defined as Program Types in the kernel's source code [*1]. A few examples of Program Types are as follows:

- XDP: An event for manipulating packets before data is copied to the kernel space when a packet arrives at a network device.
- Tracing: An event for detecting kernel function calls and passing tracepoints.
- LSM: An event for applying security policies using the Linux Security Module.

When such events occur, the eBPF program executes the processing corresponding to the Program Type. For example, a program hooked to an XDP event can decide whether to accept or drop a packet.

2.5.2 eBPF Verifier

The eBPF verifier is a program that takes eBPF programs converted to bytecode as input and verifies that they can be executed safely on the kernel. The bytecode is loaded into the kernel via the `bpf()` system call (shown as "Syscall" in Fig. 3), but the program will not run unless it passes verification by the verifier. Specifically, it checks for things like avoiding memory access violations, ensuring the program exits normally, and that the program is not granted unnecessary privileges.

In this way, the eBPF verifier enhances security by imposing restrictions on eBPF programs. Since the verifier plays a crucial role in eBPF, research has been conducted to mathematically verify the logic of the verifier [13].

2.5.3 JIT Compilation

The eBPF bytecode that passes the verifier is converted by a JIT compiler into machine code that directly runs on the target CPU. This optimizes the execution speed, allowing it to operate as efficiently as the kernel and kernel modules directly compiled from source code [12].

2.5.4 Kernel Modules

Kernel modules are a mechanism that allows for the extension of kernel functions without modifying the kernel's source code by loading object files during the execution of the Linux kernel. Kernel modules are not subject to constraints like the eBPF verifier, thus offering a high degree of program freedom. However, since kernel modules are executed with the same privileges as the kernel, it is necessary to develop carefully to avoid embedding vulnerabilities [14].

As Mayer et al. [15] point out, avoiding the creation of kernel modules can be considered an advantage of eBPF.

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

3　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．

3.1　DNS クエリのプライバシーの向上 [16]

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．

3.1.1　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．

3.1.2　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

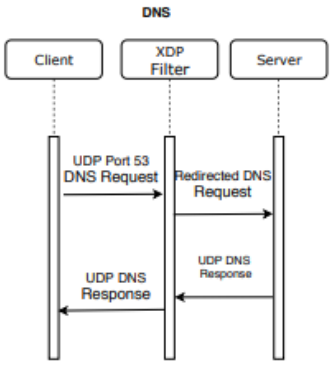

Fig. 4: DNS queries are intercept in XDP and eBPF implementation edits their destination. [16]

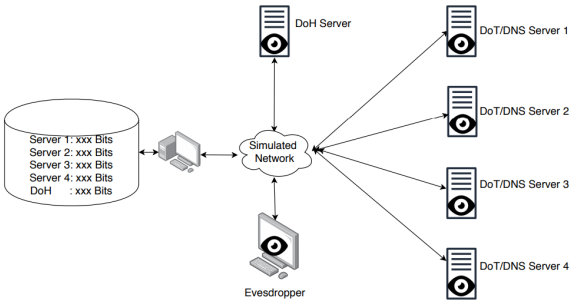

Figure 3: Experimental Network overview

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．

3.1.3　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．

3.2 Programmability の高いシステムコールフィルタ [17]

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．

3.2.1 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 [18] 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．

3.2.2 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．

3.2.3 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

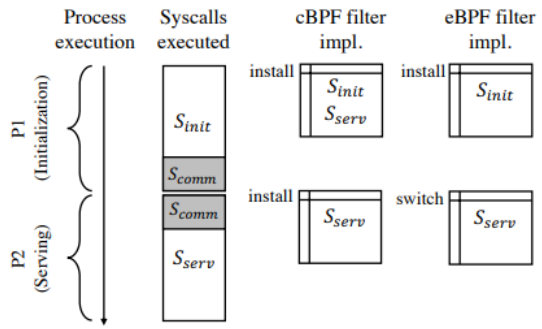

Fig. 6: The concept of precise temporal specialization. It should be noted that the seccomp-cBPF filter in the initialization phase allows S_{serv}.

	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ン

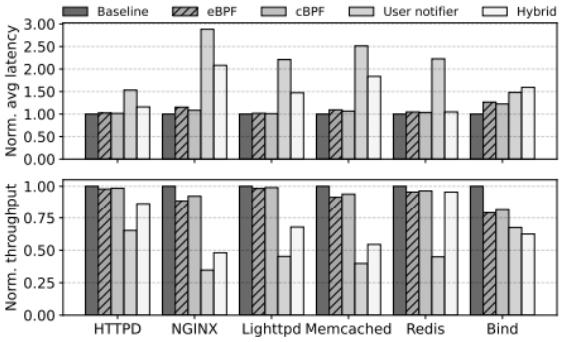

Fig. 8: Execution time of `getpid()` with different filters and filter execution time.

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．

3.3 周辺機器から Linux を保護するフレームワーク [19]

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．

3.3.1 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 [20] 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．

3.3.2 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

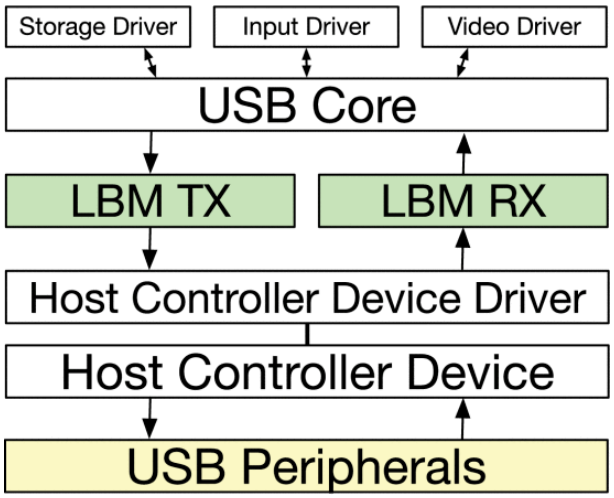

Fig. 9: The overview of LBM in kernel side. [19]

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．

3.3.3 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 [21] 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 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．

	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. [19]

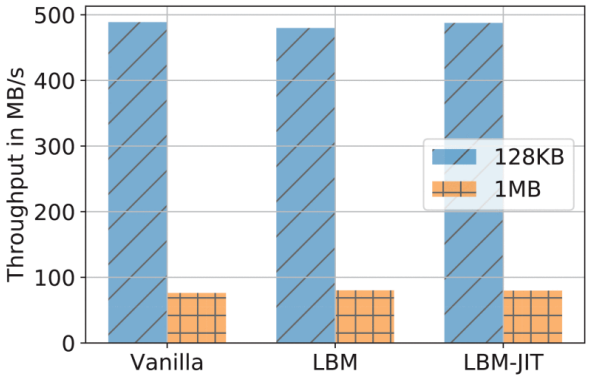

Fig. 11: Throughput of USB 3.0 external storage with filebench. [19]

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 1μs 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．

4 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] S Sibi Chakkaravarthy, D Sangeetha, and V Vaidehi. A survey on malware analysis and mitigation techniques. *Computer Science Review*, 32:1–23, 2019.

[3] Google. Virustotal. `https://www.virustotal.com`.

[4] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. Dynamic malware analysis in the modern era—a state of the art survey. *ACM Comput. Surv.*, 52(5), sep 2019.

[5] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*, 5(2):32–39, 2007.

[6] Abdullah Mujawib Alashjee, Salahaldeen Duraibi, and Jia Song. Dynamic taint analysis tools: A review. *International Journal of Computer Science and Security (IJCSS)*, 13(6):231–244, 2019.

[7] Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang. {SelectiveTaint}: Efficient data flow tracking with static binary rewriting. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1665–1682, 2021.

[8] Clemens Kolbitsch, Paolo Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and Xiaofeng Wang. Effective and efficient malware detection at the end host. pages 351–366, 01 2009.

[9] 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.

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

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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