

2025 年度 情報システム実験 B

バイナリプログラムの解析

eBPF を用いた Android 向けデバッガ「eDBG」の
x86_64 Linux への移植

提出日：2026 年 2 月 6 日

目次

1	はじめに.....	4
2	背景.....	4
2.1	ptrace を用いた従来のデバッガ.....	4
2.2	eBPF を用いたデバッガ「eDBG」.....	4
3	目標.....	5
4	技術的背景.....	5
4.1	GDB Remote Protocol	5
4.2	eBPF	5
4.2.1	フックポイント.....	5
4.2.2	helper 関数.....	5
4.2.3	eBPF Map によるデータ共有.....	6
5	実装.....	6
5.1	構成.....	6
5.2	使用ライブラリ.....	6
5.3	実装した機能の概要.....	6
5.4	プロセスの実行制御.....	7
5.4.1	エントリポイントでのブレーク.....	7
5.4.2	ソフトウェアブレークポイント.....	7
5.4.3	ハードウェアブレークポイント.....	8
5.4.4	システムコールキャッチ.....	8
5.4.5	シングルステップ実行.....	8
5.4.6	継続実行.....	8
5.5	プロセスの状態取得.....	9
5.5.1	レジスタの値の読み込み.....	9
5.5.2	レジスタの値の書き込み.....	9
5.5.3	メモリの読み書き.....	9
5.6	GDB Remote Protocol のサポート.....	9
5.7	その他に実装した機能.....	9
6	評価.....	9
6.1	評価環境.....	9
6.2	anti-debug 回避.....	10
6.2.1	評価方法.....	10
6.2.2	評価結果.....	11
6.2.3	考察.....	11
6.3	性能比較.....	12
6.3.1	評価方法.....	12
6.3.2	評価結果.....	13
7	考察.....	13
8	課題.....	14
8.1	システムコール直前でのブレーク.....	14

8.2 レジスタ書き換え.....	14
8.3 マルチスレッド対応.....	14
8.4 一般ユーザー権限での実行.....	14
9 感想.....	14
10 おわりに.....	15
参考文献.....	15

1 はじめに

ソフトウェアの動的解析において、デバッガは最も基本的なツールの一つである。デバッガを用いることで、プログラムの実行を特定の箇所で停止させ、レジスタやメモリの状態を観察したり、1 命令ずつ実行を進めて挙動を確認したりなどが可能になる。この技術は、一般的なプログラムの開発ではもちろんのこと、マルウェア解析やリバースエンジニアリングなどの分野においても重要となっている。

しかし、マルウェアや商用ソフトウェアなどでは、デバッガによる解析を妨害するための anti-debug 技術が使われていることがある。デバッガによる解析を検知して自身の動作を変更する仕組みがその一例であり、これは従来のデバッガが依存している ptrace システムコールに関連する挙動を監視することによって実現できる。したがって、ptrace に依存しないデバッガが実装できれば、こうした anti-debug 技術を回避し、より効率的な動的解析を実現できると考えられる。

本レポートでは、大課題として行った ptrace システムコールに依存しないデバッガ「eDBG」[1]の x86_64 Linux への移植、および改善について報告する。eDBG は eBPF を用いてデバッガのコア機能を実装することで、ptrace ベースの anti-debug 技術を回避する。オリジナルの実装では ARM64 Android 向けに開発されていたが、これを x86_64 Linux で動作するよう移植した。また、GDB Remote Protocol への対応を追加したほか、ブレークポイントのパフォーマンス改善を行った。

なお、以降ではオリジナルの実装を eDBG、移植後の実装を eDBG-x64 として区別する。eDBG-x64 のソースコードは <https://github.com/arata-nvm/eDBG-x64> にて公開している。

2 背景

2.1 ptrace を用いた従来のデバッガ

Linux 環境において、GDB をはじめとする従来のデバッガはその機能の実現に ptrace システムコールを使用している。ptrace の主要な機能としては、例えば以下が挙げられる。

- `PTRACE_TRACEME`, `PTRACE_ATTACH`: プロセスへのアタッチ
- `PTRACE_PEEKTEXT`, `PTRACE_POKETEXT`: メモリの読み書き
- `PTRACE_GETREGS`, `PTRACE_SETREGS`: レジスタの読み書き
- `PTRACE_SINGLESTEP`: シングルステップ実行
- `PTRACE_CONT`: 継続実行

これらの機能を使用することで、デバッガはデバッグ対象のプロセスを制御することができる。

しかしながら、ptrace システムコールには、自身がデバッグされていることをプロセスが容易に検知できるという欠点がある[2]。例えば、1 つのプロセスは 2 つ以上のプロセスからアタッチできないため、他プロセスからデバッグされていた場合、プロセス自身が `PTRACE_TRACEME` を呼び出すとエラーが返される。また、`/proc/self/status` には `TracerPid` という名前の項目があり、他プロセスからデバッグされていた場合、その PID が記録される。これらの特性は、デバッガの存在を検知するための手がかりとして容易に参照することができる。

2.2 eBPF を用いたデバッガ「eDBG」

eDBG [1]は、ARM64 Android 向けに開発されたデバッガである。eDBG の最大の特徴は、従来のデバッガが ptrace システムコールに依存しているのに対し、eBPF を用いてデバッガのコア機能を実装している点にある。

eDBG の開発動機は、前述した ptrace システムコールの欠点を克服し、ptrace ベースの anti-debug 技術を回避できるデバッガの必要性にある。eBPF はカーネル内でコードを実行する機能であり、ptrace は異なるメカニズムでプ

プロセスの監視・制御を行う。そのため、ptraceがデバッガで利用されることを前提とした anti-debug 技術は eDBG を検知できないことが期待できる。

3 目標

大課題のゴールとして、以下の 2 点を設定した。

1. eDBG を x86_64 Linux 向けに移植する
2. GDB Remote Protocol に対応する

4 技術的背景

4.1 GDB Remote Protocol

GDB Remote Protocol[3]は、GDB がリモートのターゲットをデバッグするために使用するプロトコルである。このプロトコルはテキストベースの表現を用いており、シリアル通信や TCP/IP を介してリモートと通信を行う。QEMU、BitVisor をはじめとした仮想化ソフトウェアでも広く採用されている。

基本的に、デバッガとリモート間の通信は `$<packet-data>#<checksum>` のような形式のパケットで行われる。より具体的な例としては、例えば `m <addr>,<length>$<checksum>` は指定したアドレスから指定した長さの領域のデータをメモリから読み取るためのパケットである。この例の他にも、デバッガの機能を実現するための多数のパケットが定義されている。

eDBG-x64 が GDB Remote Protocol に対応する利点は大きい。GDB はシンボル情報の解析、複雑なブレークポイント条件の設定、スクリプトによる自動化といった非常に豊富な機能を備えている。GDB Remote Protocol に対応することで、これらの GDB の機能をそのまま活用することができ、デバッガの実現のために必要な実装量を大幅に減らすことができる。

4.2 eBPF

eBPF は Linux カーネル内で安全にユーザープログラムを実行するための機能である。BPF(Berkeley Packet Filter)という名前を冠しているが、近年ではネットワーク以外にもトレーシングやパフォーマンス監視等で重要な技術になっている。

4.2.1 フックポイント

eBPF プログラムは「フックポイント」と呼ばれる地点にアタッチすることができ、Linux カーネル内で特定のイベントが発生した際に実行される。主要なフックポイントの例としては以下が挙げられる[4]。

- `raw_syscalls:sys_enter`, `raw_syscalls:sys_exit`: システムコールの呼び出し前、呼び出し後
- `kprobe`: カーネル空間の特定アドレスのコード実行時
- `uprobe`: ユーザー空間の特定アドレスのコード実行時

これら以外にも、目的や用途の異なる多数のフックポイントが Linux カーネル内に定義されている。eDBG-x64 では、これらのフックポイントを活用してプロセスをどこで停止させるかを制御している。

4.2.2 helper 関数

eBPF プログラムは、あらかじめ定義された「helper 関数」を呼び出すことで OS の情報を取得、あるいは制御することができる。主要な helper 関数の例としては以下が挙げられる[5]。

- `bpf_probe_read_user()`, `bpf_probe_write_user()`: ユーザー空間のメモリの読み書き
- `bpf_probe_read_kernel()`: カーネル空間のメモリの読み込み

- `bpf_send_signal()`: 特定プロセスへのシグナルの送信
- `bpf_get_current_task()`, `bpf_task_pt_regs()`: 特定プロセスの `task_struct`, `pt_regs` 構造体へのアクセス

eDBG-x64 では、これらの helper 関数を活用してプロセス停止時にプロセスに関する情報を収集している。

4.2.3 eBPF Map によるデータ共有

eBPF Map は、eBPF プログラムとユーザー空間のプログラムの間でデータを共有するための仕組みである。eBPF プログラムはカーネル空間で実行されるため、通常の方法ではユーザー空間のプログラムと通信することが難しい。そこで eBPF Map を使用することによって、eBPF プログラムが取得した情報をユーザー空間のプログラムに伝達したり、ユーザー空間のプログラムからの指示を eBPF プログラムに伝えたりといったことが可能になる。

eBPF Map にはアクセス方法やデータの格納方法が異なるいくつかの種類があり、例えば以下などが定義されている[6]。

- `BPF_MAP_TYPE_HASH`: キーとバリューのペアを格納するハッシュマップ
- `BPF_MAP_TYPE_ARRAY`: 同じ型の値を複数格納する配列形式のマップ

eDBG-x64 では、これらの eBPF Map を活用してプロセスに関する情報をデバッガプロセスに共有している。

5 実装

5.1 構成

eDBG-x64 は Rust で実装されており、以下の 3 つのクレートから構成される。最初期は `ebpfdbg` という名前で開発しており、その名残でクレート名が `edbg` ではなく `ebpfdbg` から始まっている。

- `ebpfdbg`: メインとなるアプリケーションで、eBPF プログラムの制御、GDB Remote Protocol の実装などを含む
- `ebpfdbg-ebpf`: カーネル空間で実行される eBPF プログラムで、各フックポイントで実行されるプログラムを実装
- `ebpfdbg-common`: `ebpfdbg` と `ebpfdbg-ebpf` で共有されるデータ構造を定義

5.2 使用ライブラリ

eDBG-x64 は以下の外部クレートを利用している。

- `aya`: Rust で eBPF に関連する操作を行うためのライブラリ
- `gdbstub`: GDB Remote Protocol に対応したアプリケーションを実装するためのライブラリ
- `capstone`: ディスアセンブルを行うためのライブラリ
- `nix`: *nix 環境向けにシステムコールのラッパーを提供するライブラリ

5.3 実装した機能の概要

表 1 に、GDB と eDBG-x64 のデバッガの機能の実装方法の比較を示す。

表 1: GDB と eDBG-x64 におけるデバッガの機能の実装方法の比較

機能	GDB での実装方法	eDBG-x64 での実装方法
エントリポイントでのブレーク	ptrace(PTRACE_TRACEME)	eBPF + tracepoint
ソフトウェアブレークポイント	ptrace(PTRACE_POKETEXT)	eBPF + uprobe
ハードウェアブレークポイント	ptrace(PTRACE_POKEUSER)	eBPF + perf_event
システムコールキャッチ	ptrace(PTRACE_SYSCALL)	eBPF + tracepoint
シングルステップ実行	ptrace(PTRACE_SINGLESTEP)	eBPF + uprobe
継続実行	ptrace(PTRACE_CONT)	kill(SIGCONT)
レジスタ読み込み	ptrace(PTRACE_GETREGSET)	eBPF
レジスタ書き込み	ptrace(PTRACE_SETREGSET)	未実装
メモリ読み込み	process_vm_readv()	process_vm_readv()
メモリ書き込み	process_vm_writev()	process_vm_writev()

5.4 プロセスの実行制御

プロセスの停止・再開の制御は基本的に SIGSTOP/SIGCONT シグナルを用いて行う。停止させたいタイミングに応じてフックポイントを選択し、そこに eBPF プログラムをアタッチすることでブレークポイントを実現している。eBPF プログラムで行う処理はブレークポイントの種類に依らずほぼ同じであり、概ね以下のような処理を行っている。

1. プロセスの現在の状態を記録し、eBPF Map に格納する
2. プロセスに SIGSTOP を送信して停止させる

ユーザー空間のデバッグプロセスは waitpid システムコールでデバッグ対象のプロセスが停止するまでブロックされている。eBPF プログラムが SIGSTOP を送信したタイミングで再開され、eBPF Map でプロセスの現在の状態を取得する。

5.4.1 エントリポイントでのブレーク

デバッグ対象のプログラムを起動し、エントリポイントで停止させるための機能である。sys_exit_execve フックポイントを使用し、execve システムコールの実行直後に eBPF プログラムを実行することで実現している。

より詳細な処理の流れは次の通りである。まずデバッガが fork システムコールで子プロセスを生成し、子プロセスは kill(SIGSTOP) で自身を停止させる。次に親プロセスが sys_exit_execve フックポイントに eBPF プログラムをアタッチし、kill(SIGCONT) で子プロセスを再開させる。子プロセスが execve システムコールでデバッグ対象のプログラムをロードすると、その直後に eBPF プログラムが実行されて bpf_send_signal(SIGSTOP) でプロセスが停止する。

5.4.2 ソフトウェアブレークポイント

ソフトウェアブレークポイントは、プロセスを特定のアドレスで停止させる機能である。eDBG-x64 では uprobe フックポイントを使用し、ユーザー空間のプログラムの特定のアドレスにフックを設定することで実現している。

内部的には GDB と同様に uprobe もブレークポイント命令を使用してブレークポイントを実装しているが[7]、ブレーク時にまずカーネル内でハンドルされる点が異なる。

なお、eBPF プログラムのアタッチ・デタッチは時間のかかる、重い操作である。一方 GDB はプログラムの実行中のみブレークポイントを設定し、それ以外の状況ではブレークポイントを削除する。これはブレークポイントが高頻度に追加・削除されるということであり、GDB からの指示に従って毎回アタッチ・デタッチを行うと非常にパフォーマンスが悪くなってしまう問題がある。そのため、eDBG-x64 では一度アタッチを行った後は原則デタッチを行わず、

eBPF Map を通じてブレークポイントの有効・無効フラグを切り替える構造とした。eBPF プログラムは各フックポイントで実行された際にこのフラグを確認し、無効であれば何もせずに処理を終了するようになっている。

5.4.3 ハードウェアブレークポイント

ハードウェアブレークポイントは、CPU のデバッグレジスタを使用してブレークポイントを実現する機能である。ソフトウェアブレークポイントとは異なり命令の書き換えを行わないため、自己書き換えを行うプログラムなどにも使用できる。また、特定のメモリアドレスへのアクセスの監視にも使用できる。eDBG-x64 では perf_event フックポイントを使用してハードウェアブレークポイントを実現している。特定のアドレスが実行された場合、もしくは特定のメモリアドレスにアクセスがあった場合に eBPF プログラムが実行される。

5.4.4 システムコールキャッチ

システムコールキャッチは、特定のシステムコールが実行された際に停止する機能である。eDBG-x64 では sys_enter および sys_exit フックポイントを使用してシステムコールキャッチを実現している。

特定のシステムコール番号でのみブレークさせるため、ブレークしたいシステムコール番号を eBPF Map で共有し、それに含まれない場合は eBPF プログラムは何もせずに処理を終了するようになっている。

現状システムコール実行前にブレークポイントを設定することはできない。これは sys_enter フックポイントで bpf_send_signal(SIGSTOP) を実行したとしても、システムコール実行後にシグナルの送信が行われる仕様によるものである。したがって、システムコール実行後になってからプロセスが停止し、sys_enter, sys_exit の 2 つの時点のプロセスの状態が観察できるようになる。ただし、GDB Remote Protocol にこのような 2 つの時点のシステムコールの情報を取得するプロトコルは存在しないため、sys_enter の情報を無視して sys_exit の情報のみを返すよう実装している。

5.4.5 シングルステップ実行

シングルステップ実行は、プログラムを 1 命令ずつ実行する機能である。GDB では ptrace システムコールで CPU のトラップフラグを立てることで実現しているが、eBPF ではそのような操作を行うことができない。したがって eDBG-x64 では、次に実行される命令のアドレスを予測し、そこへ一時的なブレークポイントを設定する形で実現している。

より具体的な手順は以下の通りである。

1. 現在停止しているアドレスの命令をメモリから取得する
2. capstone を利用して命令をディスアセンブルする
3. 命令の種類と現在のレジスタの値から次に実行されるアドレスを予測する
4. 予測したアドレスにブレークポイントを設定し、継続実行する

次に実行されるアドレスは命令の種類によって異なるため、それぞれに合わせた予測ロジックを組む必要がある。通常の命令は現在のアドレスに命令長を足したアドレスが次に実行される。RET 命令の場合は、スタックのトップに置かれているリターンアドレスが次に実行される。CALL 命令の場合は、即値、レジスタ、メモリ等で指定される呼び出し先のアドレスが次に実行される。無条件分岐命令の場合は、ジャンプ先のアドレスが次に実行される。分岐命令の場合は、分岐条件に応じて分岐先もしくは次の命令が実行される。

5.4.6 継続実行

継続実行は、停止しているプロセスの実行を再開させる機能である。eDBG-x64 では、単純に kill(SIGCONT) を実行することで実現している。

その後、waitpid システムコールを実行してデバッグ対象のプロセスが再度停止するまで待機する。

5.5 プロセスの状態取得

5.5.1 レジスタの値の読み込み

各 eBPF プログラムで記録されたレジスタの値を eBPF Map から取得し、その値を参照する。eBPF プログラムでは `pt_regs` 構造体から汎用レジスタの値を記録しているほか、`task_struct` 構造体から `thread_struct` 構造体を参照し、`fsbase` や `gsbase` などセグメントレジスタの値も記録している。

5.5.2 レジスタの値の書き込み

レジスタの値の書き込み機能は現在の eDBG-x64 では実装されていない。これは eBPF の根本的な制限に由来するもので、基本的には観測のための技術であるのでデバッガのような用途での使用が想定されておらず、実装が難しい。

5.5.3 メモリの読み書き

メモリの書き込み機能は GDB と同様に `process_vm_readv`, `process_vm_writev` システムコールを呼び出すことで実現している。

5.6 GDB Remote Protocol のサポート

GDB Remote Protocol のサポートのために、eDBG-x64 では `gdbstub` クレートを使用している。`gdbstub` クレートでは GDB との通信部分が実装されており、受信したパケットに応じて特定の関数が呼び出されるので、それらの関数の実体を定義すると GDB Remote Protocol のサポートを実装できる。そのほか、各アーキテクチャのレジスタに GDB 内で `id` が割り振られており、その対応関係を定義してくれているので、レジスタ周りの処理の実装が楽に行える。

ただし、`gdbstub` クレートは GDB との通信を 1 文字ずつ行うよう実装されていた。この実装がパフォーマンス上の大きなボトルネックとなっていたため、Buffered IO を実装した。その結果、後述する `loop-syscall` ベンチマークにおいて実行時間が 49.8 秒から 8.76 秒へと約 5 倍の高速化を達成した。

5.7 その他に実装した機能

以上に述べた基本的なデバッガの機能に加えて、GDB との連携で重要な以下の機能を実装した。

- 補助ベクタの取得: `/proc/{pid}/auxv` の内容を返す。
- ホストでのファイル操作: `open`, `readlink`, `pread`, `clone` などの操作を実行する。GDB がターゲットの `/proc/{pid}/cwd` やライブラリファイルを取得する際に使われる。
- デバッグ対象のプログラムのファイルパス取得
- デバッグ対象のプロセスの PID 取得

6 評価

eDBG-x64 の有効性を以下の 2 点から評価した。

- `ptrace` ベースの anti-debug 回避
- GDB との性能比較

6.1 評価環境

以下の環境で評価を行った。

- OS: Ubuntu 24.04.3 LTS (x86_64)

- CPU: AMD Ryzen 5 5600H
- Memory: 12GB
- GDB: 15.0.50.20240403-0ubuntu1

6.2 anti-debug 回避

6.2.1 評価方法

ptrace ベースの anti-debug 技術を実装したプログラム anti-pttrace を作成し、GDB および eDBG-x64 でデバッグした際に検知されるかを確認した。作成したプログラムをリスト 1 に示す。

```
#define _GNU_SOURCE
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

static int check_tracerpid(void) {
    printf("[check_tracerpid] /proc/self/status ... ");

    FILE *f = fopen("/proc/self/status", "r");
    if (!f) {
        perror("fopen");
        exit(1);
    }

    char line[512];
    int tracerpid = -1;
    while (fgets(line, sizeof(line), f)) {
        if (strncmp(line, "TracerPid:", 10) == 0) {
            tracerpid = atoi(line + 10);
            break;
        }
    }

    fclose(f);

    if (tracerpid == -1) {
        printf("TracerPid not found\n");
        exit(1);
    } else if (tracerpid != 0) {
        printf("TRACED (TracerPid: %d)\n", tracerpid);
        return 1;
    } else {
        printf("OK\n");
        return 0;
    }
}
```

```

}

static int check_ptrace_traceme(void) {
    printf("[check_ptrace_traceme] ptrace(PTRACE_TRACEME) ... ");

    errno = 0;
    long ret = ptrace(PTRACE_TRACEME, 0, NULL, NULL);
    if (ret == -1) {
        if (errno != EPERM) {
            perror("ptrace");
            exit(1);
        }

        printf("TRACED\n");
        return 1;
    }

    printf("OK\n");
    return 0;
}

int main(void) {
    printf("pid = %d\n", getpid());

    check_tracerpid();
    check_ptrace_traceme();

    return 0;
}

```

リスト 1: ptrace ベースの anti-debug 技術を実装したプログラム anti-ptrace

6.2.2 評価結果

評価結果を表 2 に示す。

表 2: anti-ptrace の実行結果

	通常実行	GDB	eDBG-x64
check_tracerpid	検知されず	検知	検知されず
check_ptrace_traceme	検知されず	検知	検知されず

6.2.3 考察

eDBG-x64 が ptrace ベースの anti-debug 技術を回避できることを確認できた。

ただしこれは eDBG-x64 が検知不可能であることを示すものではない。eBPF 向けの手法を用いることで eDBG-x64 によってデバッグされていることを検知することは十分可能であると思われる。例として uprobe フックポイントを用いると uprobe 用のページが作成されるケースがあり、そのようなページの存在によってアタッチされていることを検知できる。一方このような手法は ptrace ベースの anti-debug 技術と比較して一般的ではなく、現時点では回避できる anti-debug 技術が一定数存在するものと思われる。

6.3 性能比較

6.3.1 評価方法

ブレークポイントを設定し、継続実行を繰り返した際の実行時間を GDB(gdbserver)と eDBG-x64 で比較した。比較のために作成したプログラムをリスト 2 およびリスト 3 に示す。loop-func.c は 10,000 回空の関数を呼び出すプログラム、loop-syscall.c は 10,000 回 getpid システムコールを呼び出すプログラムである。

```
#include <stdint.h>
#include <stdio.h>

void do_something() { }

int main(void) {
    uint64_t n = 10000ULL;

    for (uint64_t i = 0; i < n; i++) {
        do_something();
    }

    printf("iterations=%llu\n", (unsigned long long)n);

    return 0;
}
```

リスト 2: ベンチマーク用プログラム loop-func.c

```
#include <stdint.h>
#include <stdio.h>
#include <sys/syscall.h>
#include <unistd.h>

int main(void) {
    uint64_t n = 10000ULL;

    for (uint64_t i = 0; i < n; i++) {
        getpid();
    }

    printf("iterations=%llu\n", (unsigned long long)n);

    return 0;
}
```

リスト 3: ベンチマーク用プログラム loop-syscall.c

実行時間の計測にはリスト 4 およびリスト 5 に示す GDB スクリプトを用いた。

```
import time
import gdb

gdb.execute('set debuginfod enabled off')
gdb.execute('target remote :9001')
gdb.execute('b do_something')
```

```

gdb.execute('''commands 1
silent
continue
end
''')

start_time = time.time()
gdb.execute('c')
end_time = time.time()
print(f"Time taken: {end_time - start_time:.3f} seconds")

```

リスト 4: loop-func.c の実行時間計測のためのスクリプト

```

import time
import gdb

gdb.execute('set debuginfod enabled off')
gdb.execute('target remote :9001')
gdb.execute('catch syscall getpid')
gdb.execute('''commands 1
silent
continue
end
''')

start_time = time.time()
gdb.execute('c')
end_time = time.time()
print(f"Time taken: {end_time - start_time:.3f} seconds")

```

リスト 5: loop-syscall.c の実行時間計測のためのスクリプト

それぞれ3回実行時間を計測し、その平均を算出した。

6.3.2 評価結果

評価結果を表3に示す。

表 3: 実行時間（単位：秒）

	ブレークポイントなし	gdbserver	eDBG-x64
loop-func	0.070	12.4	11.0
loop-syscall	0.071	17.3	8.76

結果として、eDBG-x64 は GDB と比較して loop-func で約 1.13 倍、loop-syscall で約 1.97 倍の高速化を達成した。

7 考察

まず、loop-func での明確な高速化要因は特定できていないが、いくつかの要因が考えられる。第一に、GDB と eDBG-x64 はともにブレークポイントの実装にブレークポイント命令を使用している。この点では大きな差異は生じないはずである。第二に、継続実行時のシステムコール発行回数に違いがある可能性がある。GDB の場合、継続実行では PTRACE_POKETEXT で元の命令に戻し、PTRACE_SINGLESTEP で1命令実行し、PTRACE_POKETEXT でブレークポイント命令を書き込み、最後に PTRACE_CONT で再開するという複数のシステムコールが発行される。一方 eDBG-x64 の場合、kill(SIGCONT)のみで済む。このシステムコール発行回数の差が要因の一つとなった可能性がある。第三に、eDBG ではブレークポイントを削除してもブレークポイント命令に

パッチしたままであり、命令キャッシュが効いている可能性がある。GDB では毎回命令を書き直しているため、命令キャッシュの恩恵を受けにくいことが考えられる。

次に、loop-syscall での高速化要因はシステムコールキャッチの実装方式の違いに起因すると考えられる。GDB でシステムコールをキャッチする場合、PTRACE_SYSCALL を使用する。この方式ではシステムコールが発行されるたびにユーザー空間の GDB に処理が移り、ブレークするかを判定したのち ptrace(PTRACE_SYSCALL)で継続実行する。この処理はカーネル空間とユーザー空間でのコンテキストスイッチが伴うため、大きなオーバーヘッドとなる。一方 eDBG では eBPF プログラムがカーネル空間でブレーク対象かの判定を行う。この処理はブレークする場合にのみデバッガプロセスへのコンテキストスイッチが発生するため、オーバーヘッドを減らすことができる。

なお loop-syscall に関連して、プログラムの特性によってはさらに高速化の恩恵を得られる可能性がある。例えばキャッチしたいシステムコールの呼び出し頻度が少なく、他のシステムコールが頻繁に呼び出されるプログラムでは GDB との性能差がより顕著になると予想される。

8 課題

eDBG の移植を通じて、eBPF を利用してデバッガを実装することにはいくつかの根本的な制限があることが明らかになった。eBPF は本来トレース用途の機能であるため、インタラクティブにデバッグを行うための機能は難しいことが多い。

8.1 システムコール直前でのブレーク

sys_enter において bpf_send_signal(SIGSTOP)を呼び出しても、システムコール実行後にプロセスが停止するため、システムコール直前でブレークすることが難しい。

8.2 レジスタ書き換え

eBPF プログラムからカーネル空間へのメモリ書き込みは禁止されており、pt_regs 構造体の値を変更できないため、レジスタの値を書き換えることが難しい。

8.3 マルチスレッド対応

eBPF は本質的にイベント駆動の仕組みであり、フックポイントが発火した場合にのみ処理を行うことができる。マルチスレッドプログラムのデバッグでは、あるスレッドでブレークした場合に他のスレッドでも eBPF プログラムを実行してすべてのスレッドの状態を取得できる必要がある。しかし、最初にブレークしたスレッドでプロセスを停止させるため、他のスレッドで発火させられるフックポイントが存在しない。したがって、他のスレッドの状態を取得することが困難であり、それゆえにマルチスレッド対応が難しい。

8.4 一般ユーザー権限での実行

eBPF プログラムをアタッチするためには CAP_BPF や CAP_PERFMON といった権限が必要であり、一般ユーザー権限では eDBG-x64 を実行できない。

9 感想

eDBG の ARM64 Android から x86_64 Linux への移植は想像よりスムーズに行うことができた。これは、eBPF のポータビリティの高さと、GDB のマルチアーキテクチャ対応によるものが大きいと思われる。特別に変更が必要だった点としては、レジスタセットと命令セットの違いへの対応が挙げられる。前者は pt_regs 構造体の構造がアーキテクチャごとに異なるため、その差異を修正する必要があった。後者はシングルスレッドの実行において、次に実行されるアドレスの予測のために命令を解析するため、x86_64 向けに新規にロジックを組む必要があった。

10 おわりに

本レポートでは、eBPF を用いたデバッガ「eDBG」の x86_64 Linux への移植と改善に関して報告した。

eDBG は ptrace を使用せず eBPF でデバッガのコア機能を実装することで、ptrace ベースの anti-debug 技術を回避できることを実証した。この特徴は移植後の eDBG-x64 においても実証された。

eDBG-x64 での新規の貢献として、GDB Remote Protocol に対応することで GDB の豊富な機能と組み合わせて使用できるようにした。また、ブレークポイント及び gdbstub のパフォーマンス改善を行い、最大 2 倍程度の高速化を達成した。特にシステムコールキャッチでの高速化は eBPF がカーネル内で処理を完結できる特性を活かした結果と言える。

一方で eBPF の制限により、レジスタの書き込み、システムコール直前でのブレーク、マルチスレッド対応などの機能は実現が難しい。また、実行のために強い権限が必要である点も制限の一つである。

総じて、eBPF を用いてデバッガを実装するアプローチは、ptrace ベースの anti-debug を回避したい場面、高速なシステムコールのキャッチが必要な場面などで有効であることが示された。ptrace ベースの従来のデバッガを完全に置き換えるものではないが、デバッガを実装するための技術の新たな選択肢として意義があると考えられる。

参考文献

- [1] “ShinoLeah/eDBG: eBPF-based lightweight debugger for Android.” [Online]. Available: <https://github.com/ShinoLeah/eDBG>
- [2] “Programming Linux Anti-Reversing Techniques.” [Online]. Available: <https://www.kneda.net/documentos/Programming%20Linux%20Anti-Reversing%20Techniques.pdf#page=224.85>
- [3] “Remote Protocol (Debugging with GDB).” [Online]. Available: <https://sourceware.org/gdb/current/onlinedocs/gdb.html/Remote-Protocol.html>
- [4] “Program types (Linux) - eBPF Docs.” [Online]. Available: <https://docs.ebpf.io/linux/program-type/>
- [5] “Helper functions - eBPF Docs.” [Online]. Available: <https://docs.ebpf.io/linux/helper-function/>
- [6] “Map types (Linux) - eBPF Docs.” [Online]. Available: <https://docs.ebpf.io/linux/map-type/>
- [7] “4. Working with dynamic user probes | bpftrace.” [Online]. Available: <https://bpftrace.org/hol/user-probes>