Systemtap - instrumentacja działającego Linuksa

Kamil Nowosad

Systemy Rozproszone 2010/2011

4 listopada 2010

- Co chcemy osiągnąć
- Kprobes - niskopoziomowa baza
- SystemTap
  - co się na niego składa
  - kilka praktycznych zastosowań
  - rozwój i rozszerzenia

Problemy, o których tu mówimy

- Naszym celem jest badanie (z możliwością drobnych modyfikacji) interakcji programu ze światem (poprzez OS):
  - otwierane pliki (ile?, jakie?, jak otwierane?)
  - połączenia sieciowe
  - zużycie zasobów
  - częstość wybudzania (context switche)
  - czas trwania takich operacji
- Program niekoniecznie musi być nasz (niekoniecznie mamy źródła)

Problemy, o których tu mówimy 2

- Być może chcemy też badać grupę programów
  - jak często komunikują się po pipe'ie
  - kto ile czeka na komunikaty
- Albo nawet cały system
  - jakie wykonujemy połączenia w świat
  - kto najintensywniej korzysta z dysku
  - ile czasu procesy czekają na semaforach na dostęp do interfejsów sieciowych
- może też chcemy wprowadzić drobne modyfikacje - np zasymulować jakieś błędy systemowe...

Narzędzia znane i od lat stosowane

- top
- strace
- vmstat
- iotop
- ...
- i inne programy oparte głównie na procu i syscallach

- generalnie wyspecjalizowane narzędzia, z których każde pokrywa pewną część pożądanych funkcji

Wady tych narzędzi

- mogą mieć duży narzut (wołają dużo syscalli, czytają i przetwarzają dużo danych z proca)
- nie zawsze wystarczają
  - ich output trzeba obrabiać w dodatkowych skryptach
  - lub w ogóle nie da sie wyciągnąć tego, co się chce
- ogólnie czasem robią za dużo, czasem za mało i są mało dostosowywalne (customizable) i wcale nie są rozszerzalne
- jest to wiele niezależnych narzędzi, w związku z czym trudno integrować ich dane wyjściowe (trzeba dużo grepować i sedować)

Cechy idealnego rozwiązania

- łatwo dostępne - najlepiej zintegrowane z systemem operacyjnym i dostępne na żądanie
  - przydatne np przy poszukiwaniach: "dlaczego akurat na tym systemie mój program działa tak wolno"
- nie mające narzutów wtedy, kiedy jest wyłączone
  - czyli odpada np nawrzucanie do jądra kodu, który zrzuca nam gdzieś przeróżne statystyki
  - albo kompilowanie aplikacji tak, żeby przy każdym wywołaniu read'a badała czas jego trwania

Cechy idealnego rozwiązania 2

- bezpieczne w użyciu w środowisku produkcyjnym (znowu problemy z wdrożeniem, często u klienta)
- wszechstronność - chcemy badać równie dobrze aplikacje użytkownika i sterowniki
- łatwość użycia
- szybkość użycia - najlepiej bez przkompilowywania całego projektu / jądra
- możliwość tworzenia reużywalnych rozwiązań

Taki właśnie jest SystemTap !

Idea

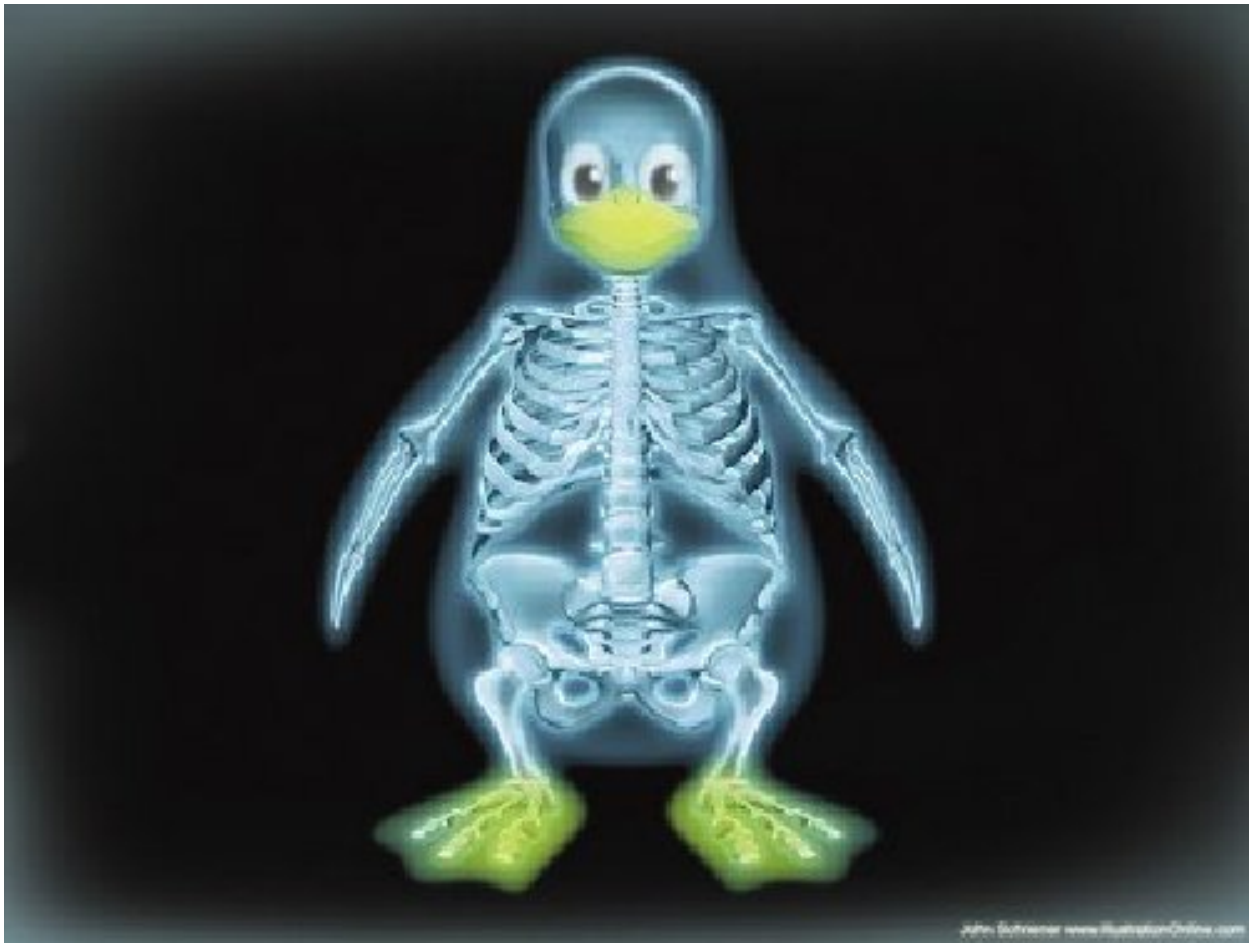
- Wszystkie potrzebne informacje można wyciągnąć z jądra:
  - siedzą w strukturach danych
  - są do wyciągnięcia po prześledzeniu przepływu sterowania w jądrze
  - i sprofilowaniu funkcji jądra
- Uniezależniamy się więc od kodu aplikacji

Idea - przykład

Powiedzmy, że chcemy przechwytywać jakiś syscall, np. read:

- można go przechwycić na poziomie uruchamianej aplikacji
  - opakowując wołania reada
  - używając `LD_PRELOAD` ze swoim opakowanym readem
- można użyć strace (czyli ptrace)
  - to już jest niezłe, ale musimy podać konkretny pid
- a można też wpiąć się funkcję `sys_read` w jądrze i przechwycić wszystko

Najpierw o tym, na czym się opieramy



Kprobes

Co to jest Kprobes

Kprobes to API umożliwiające pisanie modułów, które umieszczają sondy (ang. probes) w kodzie jądra.
Sonda

- jest wywoływana w momencie, kiedy wywoływanie kodu dojdzie do ustalonego miejsca
- otrzymuje jako argument kontekst wykonania (struct pt_regs)
- o ile kontekst nie zostanie zmodyfikowany, sondowanie nie zmienia wykonania kodu

Cechy kprobes

- wymagają skompilowania i załadowania modułu - nie wymagają modyfikacji w samym jądrze
- przed umieszczeniem i po usunięciu sond nie ma żadnego narzutu
- taka sonda może być umieszczona pod praktycznie dowolnym adresem w jądrze (wyjątki później)

Rodzaje sond

Omówię dwa spośród kilku typów sond, które są w Kprobes zaimplementowane:

- Kprobe
  - umożliwia zarejestrowanie pary funkcji w dowolnym miejscu kodu (na dowolnej instrukcji), które zostaną wywołane za każdym razem bezpośrednio przed wykonaniem tej instrukcji i po
  - do funkcji będzie przekazany uchwyt sondy (zawierający m.in. adres sondowanej instrukcji)
  - i struktura pt_regs zawierająca zawartość rejestrów w chwili sprzed wykonania instrukcji
- Return probe (kretprobe)
  - przypisana do funkcji
  - potrafi dostać się do wartości zwracanej z funkcji

Jak działa sonda typu Kprobe

- rejestrując sondę podajemy:
  - adres operacji
  - funkcję do wywołania przed operacją (pre_handler)
  - funkcję do wywołania po operacji (post_handler)
- na początek sondowanej instrukcji: int 3 (ma ona jeden bajt)
- kiedy dojdzie do naszego int 3, wołany jest breakpoint handler

Jak działa sonda typu Kprobe 2

- w breakpoint handlerze przeszukiwane są nasze sondy i wybierana jest ta odpowiadająca naszej instrukcji
- i wołany jest pre_handler (z wyłączonymi przerwaniami !)
- po nim w trybie single-step jest kopia (koniecznie!) sondowanej instrukcji
- i wołany jest post_handler+
- przywracany jest kontekst wykonania

Jak działa sonda kretprobe

- rejestrując sondę podajemy:
  - adres sondowanej funkcji
  - funkcję do wywołania przed samym powrotem z sondowanej funkcji
- na wejściu sondowanej funkcji ustawiany jest Kprobe, który podmienia adres powrotu z funkcji na adres naszej funkcji sondującej (zapamiętując gdzieś właściwy adres powrotu)
- w momencie returna z funkcji, robiony jest skok w nasz kod
- tam jest wołany nasz handler, a na końcu jest wykonywany skok w zapamiętany adres powrotu

Możliwości i ograniczenia

- jak juz zostało powiedziane, sondować można praktycznie dowolny kawałek kodu
  - z procedurami obsługi przerwań włącznie!
- jest kilka drobnych wyjątków...
  - funkcje samego Kprobes - deadlocki, zapętlenia itd.
  - funkcje, które mogą się wyinline'ować (a kompilujemy zazwyczaj z -O2!)
  - zapętlenia (np. używanie printk sondując printk) - pomijanie drugiego sondowania
- poza tym sondy są uruchamiane z:
  - wyłączonym wywłaszczaniem (preemption)
  - (w zależności od architektury, ew. kontekstu) z zamaskowanymi przerwaniami
- w związku z tym, nie można w sondach wywoływać operacji oddających procesor (np. semaforowych)

SystemTap

Idea SystemTap

- instrumentacja z wykorzystaniem tylko Kprobes nie jest zbyt wygodna
  - trzeba napisać moduł do jądra
  - skompilować
  - załadować
- fajne, ale trochę zbyt niskopoziomowe jak na wyciąganie prostych w zasadzie informacji..
- stwórzmy język skryptowy i zautomatyzujmy cały ten proces!
- i stwórzmy zestaw gotowych narzędzi do wyciągania informacji z jądra dla tego języka
- i narzędzia do wizualizacji pobranych danych

Przykład

Poniższy kawałek kodu wystarczy, żeby wypisać otwierane pliki z wydzieleniem tworzonych:

```
1 : function is_open_creating:long (flag:long){
2 :  CREAT_FLAG = 4 // 0x4 = 00000100b
3 :  if (flag & CREAT_FLAG){
4 : return 1
5 : }
6 : return 0
7 : }
8 :
9 : probe kernel.function("sys_open"){
10:  creating = is_open_creating($mode)
11:  if (creating)
12:     printf("Creating file %s\n", user_string($filename))
13:  else
14:     printf("Opening file %s\n", user_string($filename))
15: }
```

Przykład

```
1 : function is_open_creating:long (flag:long){
...
9 : probe kernel.function("sys_open"){
10:  creating = is_open_creating($mode)
11:  if (creating)
12:     printf("Creating file %s\n", user_string($filename))
13:  else
14:     printf("Opening file %s\n", user_string($filename))
15: }
```

- 9: definiujemy sondę na wejściu do funkcji jądra `sys_open()`
- 10: tworzymy zmienną creating i przypisujemy jej wartość naszej funkcji `is_open_creating` dla argumentu `mode`
  - jest jeden z argumentów funkcji open
  - ponieważ jesteśmy na wejściu funkcji, mam dostępne wszystkie jej argumenty w niezmodyfikowanej formie
- 11-14: standardowy if i standardowy printf

Przykład

```
1 : function is_open_creating:long (flag:long){
2 :  CREAT_FLAG = 4 // 0x4 = 00000100b
3 :  if (flag & CREAT_FLAG){
4 : return 1
5 : }
6 : return 0
7 : }
8 :
```

- No i możemy deklarować różne przydatne funkcje

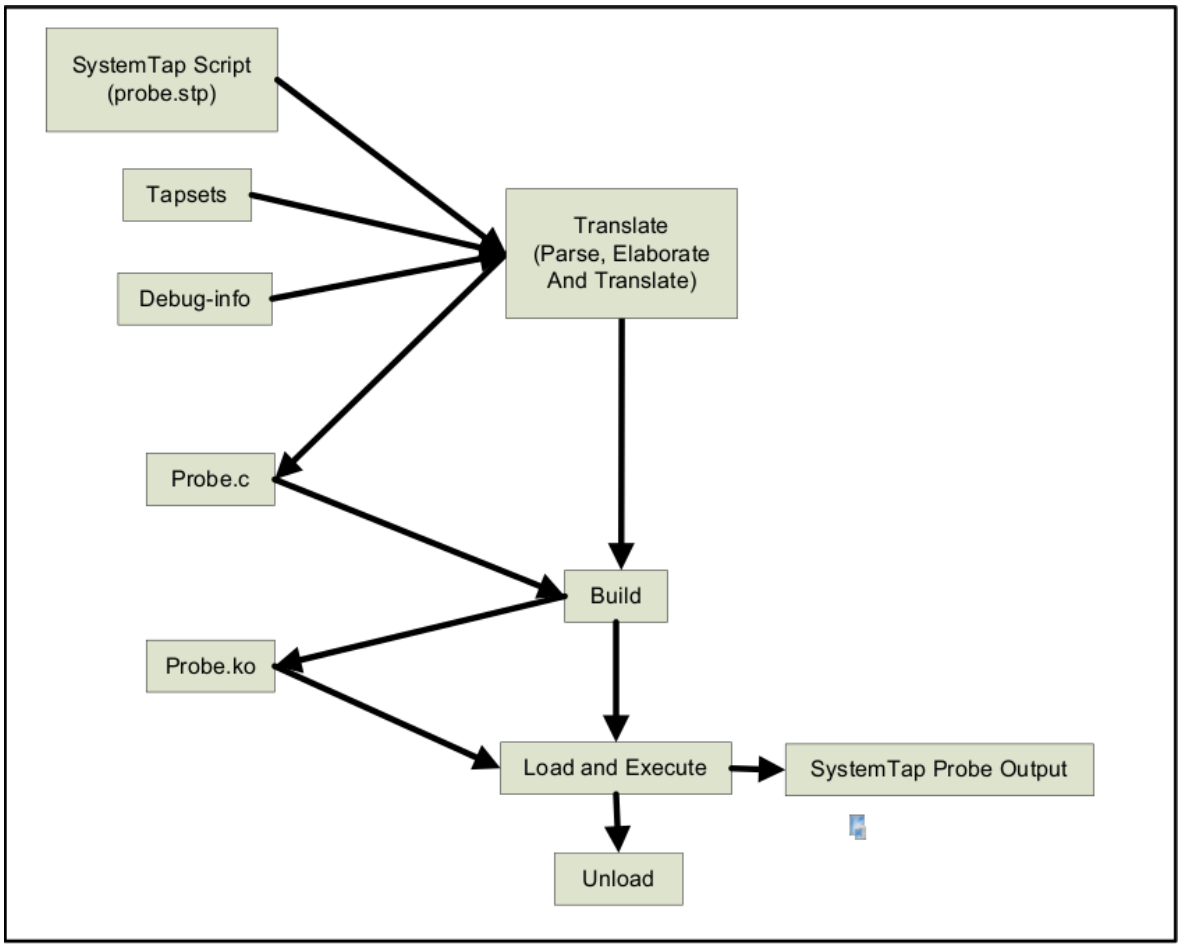
Co dalej z tym robimy...

- zapisujemy nasz pierwszy program do `open.stp`
- i uruchamiamy...

  ```
  $ stap open.stp
  ```

- zbieramy sobie output...
- jak mamy dość - ctrl+c
- ... i tyle :)

Co się dzieje z naszym skryptem...



Co się dzieje... - konwersja do modułu

- rozwiązywane są symbole i referencje
  - korzystając z debug info jądra podpinamy odpowiednie adresy funkcji i zmiennych
- zmienne używane przez wiele sond są zabezpieczane lockami
- są wstawiane różne kontrole bezpieczeństwa (np. NULLe)
- jest budowany moduł do jądra (w większości przypadków) używający Kprobes

Co się dzieje... - uruchomienie

- moduł jest ładowany do jądra
- w sekcji init modułu są ustawiane sondy
- moduł działa dopóki działa skrypt uruchamiający go
  - wyjście z modułu jest przekazywane przez mechanizm zwany relayfs
- wtedy odrejestrowywane są Kprobes
- i moduł jest wyładowywany

Deklarowanie sond

```
probe kernel.function(...)
probe kernel.function(*@net/socket.c")
    // różne wildcardy
probe kernel.trace("tracepoint")
    // tracepointy w kernelach (>= 2.6.30)
probe syscall.system_call
    // np syscall.open ; syscall.*
probe XXX.return
    // np. syscall.open.return
probe module("module").function("function")
    // np. module("ext3").function("*")
probe begin
probe end
probe timer
    // timer.s(3)  ; timer.jiffies(500) (różne jednostki)
```

Użyteczne funkcje

Przykłady funkcji do użycia w sondach

- exit() - Exit the running script.
- probefunc() - Return the name of the function being probed.
- execname() - Return the name of the current running process.
- pid() - Return the process id of the current process.
- uid() - Return the user ID of the current process.
- cpu() - Return the CPU number the current process is executing on.
- returnstr() - Get the return value of probed function.

Tapsets

- SystemTap oprócz języka udostępnia bibliotekę funkcji i predefiniowanych sond
- upraszczają one życie, jak każdy zbiór funkcji bibliotecznych
- i uniezależniają nas od architektury, wersji jądra itp.
- co pomaga nam w pisaniu skryptów, które są przenośne

Przykład predefiniowanej sondy tapsetowej

```
probe usb.submit_urb = kernel.function("usb_submit_urb")
{
    urb = $urb
    dev = $urb->dev
    flags = $mem_flags
}
```

zastosowanie:

```
probe usb.submit_urb
{
    printf("usb_submit_urb called on device at %x\n", dev)
}
```

Tapsets - idea

- pomysł jest taki, że pojedynczy tapset zazwyczaj powinien dotyczyć jakiegoś obszaru jądra
- i być wiarygodnym, szybkim (napisanym bezpośrednio w C) kodem, napisanym przez ekspertów

War stories

Przykłady użycia...

Wydajność

- przyrost ilości kodu:
  - Kprobes - zerowy
  - statyczne markery - kilkanaście bajtów
- narzut czasowy w stanie nieaktywnym
  - Kprobes - nie ma
  - statyczne markery - pomijalny (`if (unlikely(...))`)
- narzut czasowy w stanie aktywnym (tabelka slajd dalej)
  - Kprobes - duży
    - wywołanie przerwania
    - zidentyfikowanie procedury obsługi przerwania
    - zawołanie jej (wejście do Kprobes)
    - odnalezienie sondy i zawołanie jej
    - single-stepping instrukcji
    - itd.
  - statyczne markery - ok. 50-60 cykli

Wydajność - test

- Benchmark na Pentium 4 HT, 3GHz
- Wartości w nanosekundach.
- Każda sonda identyczna, inkrementująca globalny licznik.

function	marker	kprobe	oba	żaden
getuid	820	2100	2250	620
getgid		2100		620

- pełne uruchomienie sondy ze statycznymi markerami: 200ns
- pełne uruchomienie sondy z Kprobes: 1480ns
- samo zawołanie sondy przez statyczny marker: 20ns (zmierzone wcześniej; 50-60 cykli)
- czas działania sondy: 180ns
- narzut Kprobes: 1300ns

Sondy w przestrzeni użytkownika

Kolejna funkcjonalność

- Wydaje się naturalnym rozwinięciem pomysłu
- Skoro tak łatwo nam sondować kod jądra, jeszcze łatwiej będzie z kodem użytkownika...
- Mielibyśmy wtedy jedno zintegrowane narzędzie do badania całego systemu

Ale są inne narzędzia...

- możnaby dyskutować, że jednak istnieją bardzo dobre narzędzia do debugowania czy profilowania kodu użytkownika
- opierają się one zazwyczaj na mechanizmach opartych na `ptrace`
  - mechanizmy te są jednak o rząd wielkości wolniejsze od rozwiązań opartych na Kprobes
- poza tym, czasem może przydać się współpraca operacji po stronie użytkownika i jądra - w naszym rozwiązaniu mamy ją za darmo !
- no i mamy dostęp do bardzo wielu informacji

Sondowanie w przestrzeni użytkownika

```
function time() { return gettimeofday_us() }
probe process("psql").function("SendQuery").call
{
      entry[tid()]=time()
}
probe process("psql").function("SendQuery").return
{
      tid=tid()
      if (! ([tid] in entry)) next
      query=user_string($query)
      queries[query] <<< time() - entry[tid]
      delete entry[tid]
}
/* and an ‘‘end’’ probe to format report */
```

Ładne wypisanie zgromadzonych danych...

```
probe end,error,timer.s(5) {
      foreach ([q] in queries limit 1)
        { any = 1 }
      if (any) {
        printf("%2s %6s %-40s\n",
               "#", "uS", "query");
        foreach ([q] in queries- limit 10)
          printf("%2d %6d %-40s\n",
                 @count(queries[q]),
                 @avg(queries[q]), q)
        printf("\n");
        delete queries
      }
}
```

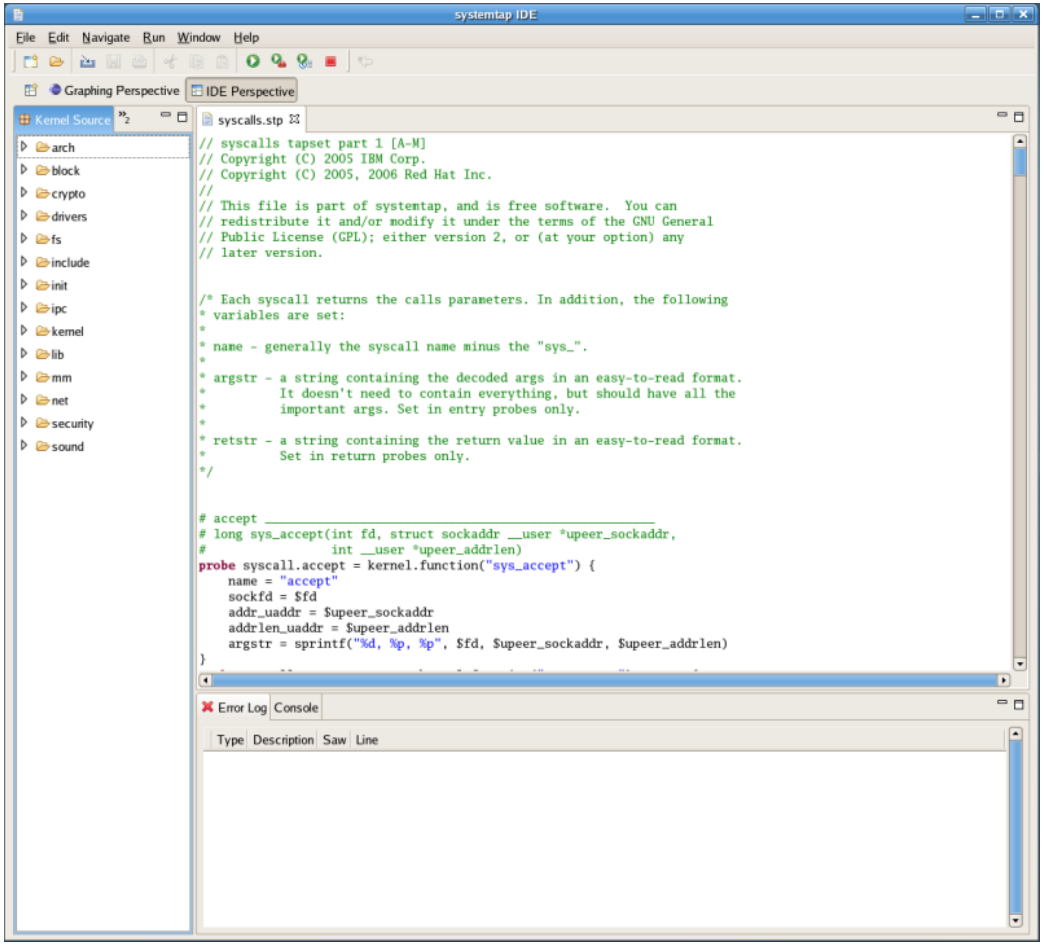
Rezultat

```
 #      uS query
12     990 DELETE FROM num_result;
 6    3909 COMMIT TRANSACTION;
 6     132 BEGIN TRANSACTION;
 6     143 SELECT date '1999-01-08';
 4    3651 insert into toasttest
values(decode(repeat('1234567890',10000),'escape'));
 4    3786 insert into toasttest
values(repeat('1234567890',10000));
 4    1218 SELECT '' AS five, * FROM FLOAT8_TBL;
 3     804 END;
 3     295 BEGIN;
 3    1032 INSERT INTO TIMESTAMPTZ_TBL VALUES ('now');
```

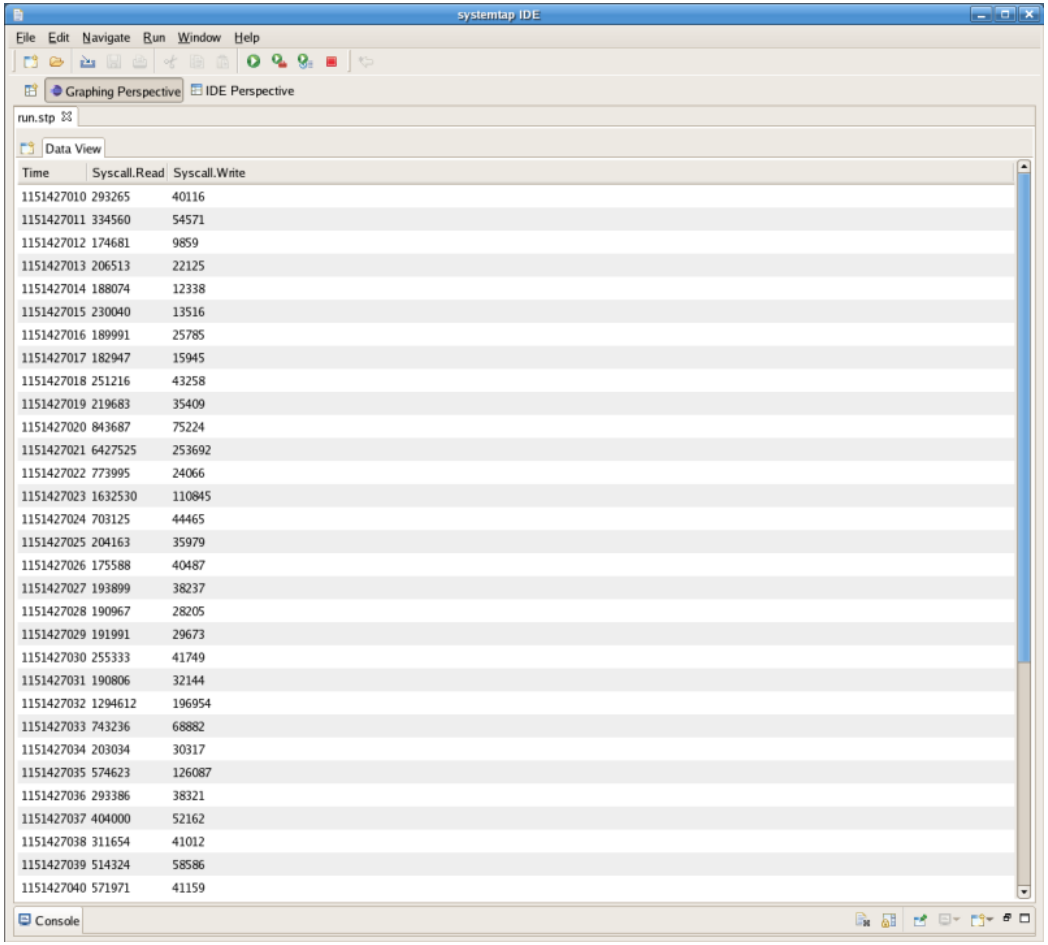
Przyszłość

- sondowanie i backtrace'owanie javy
- tryb użytkownika nieuprzywilejowanego (masochism mode)
  - próba znalezienia backendu dla operacji niekernelowych
- operacje wykonywalne bez znajomości informacji debugujących

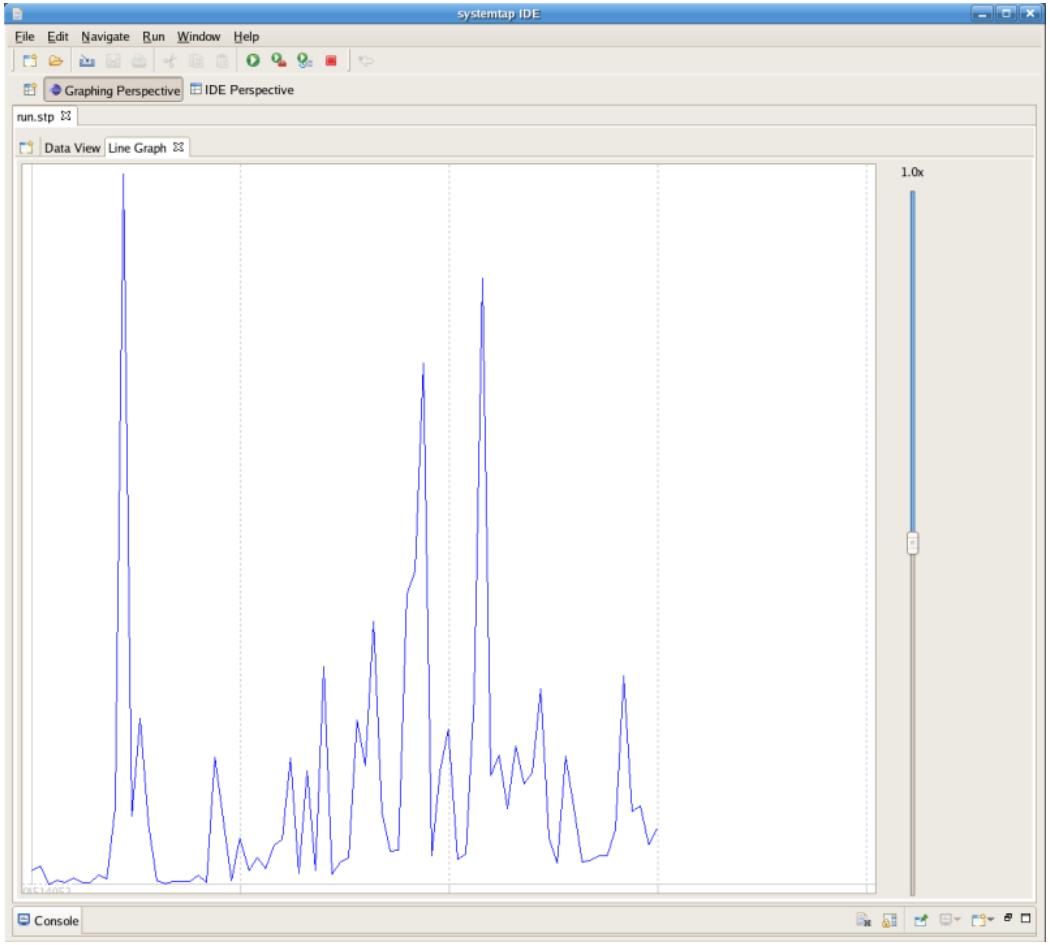
GUI


systemtap IDE
File Edit Navigate Run Window Help
Graphing Perspective
IDE Perspective
Kernel Source
arch
block
crypto
drivers
fs
include
init
ipc
kernel
lib
mm
net
security
sound
syscalls.stp
// syscalls tapset part 1 [A-M]
// Copyright (C) 2005 IBM Corp.
// Copyright (C) 2005, 2006 Red Hat Inc.
//
// This file is part of systemtap, and is free software. You can
// redistribute it and/or modify it under the terms of the GNU General
// Public License (GPL); either version 2, or (at your option) any
// later version.
/* Each syscall returns the calls parameters. In addition, the following
* variables are set:
*
* name - generally the syscall name minus the "sys_".
*
* argstr - a string containing the decoded args in an easy-to-read format.
* It doesn't need to contain everything, but should have all the
* important args. Set in entry probes only.
*
* retstr - a string containing the return value in an easy-to-read format.
* Set in return probes only.
*/
# accept
# long sys_accept(int fd, struct sockaddr __user *upeer_sockaddr,
# int __user *upeer_addrlen)
probe syscall.accept = kernel.function("sys_accept") {
name = "accept"
sockfd = $fd
addr_uaddr = $upeer_sockaddr
addrlen_uaddr = $upeer_addrlen
argstr = sprintf("%d, %p, %p", $fd, $upeer_sockaddr, $upeer_addrlen)
}
Error Log
Console
Type Description Saw Line

GUI


systemtap IDE
File Edit Navigate Run Window Help
Graphing Perspective
IDE Perspective
run.stp
Data View
Time Syscall.Read Syscall.Write
1151427010 293265 40116
1151427011 334560 54571
1151427012 174681 9859
1151427013 206513 22125
1151427014 188074 12338
1151427015 230040 13516
1151427016 189991 25785
1151427017 182947 15945
1151427018 251216 43258
1151427019 219683 35409
1151427020 843687 75224
1151427021 6427525 253692
1151427022 773995 24066
1151427023 1632530 110845
1151427024 703125 44465
1151427025 204163 35979
1151427026 175588 40487
1151427027 193899 38237
1151427028 190967 28205
1151427029 191991 29673
1151427030 255333 41749
1151427031 190806 32144
1151427032 1294612 196954
1151427033 743236 68882
1151427034 203034 30317
1151427035 574623 126087
1151427036 293386 38321
1151427037 404000 52162
1151427038 311654 41012
1151427039 514324 58586
1151427040 571971 41159
Console

GUI


systemtap IDE
File Edit Navigate Run Window Help
Graphing Perspective
IDE Perspective
run.stp
Data View
Line Graph
1.0x
Console

Rozwijane zastosowania

- SCSI fault injection
- boot profiling
- profilowanie NFSa
- Mortadelo - monitorowanie operacji plikowych w systemie
- GUI

Podsumowanie

Zalety SystemTap:

- łatwo dostępny
  - oparty na Kprobes, standardowym składniku jądra
- prosty i bardzo wygodny w użyciu
- dobrze nadaje się do systemów produkcyjnych
- skrypty są czytelne (w odróżnieniu od np. łatek do jądra)
- guru mode i modyfikacje (i już jest prawie jak w DOSie :))

Wady:

- bezpieczeństwo
- koniecznosc posiadania kernel headers i kernel debug info
  - ale można robić cross-kompilację
- kompatybilność z wersjami jądra
  - częściowo rozwiązywana przez tapsety

Bibliografia

- `http://sourceware.org/systemtap`
  dużo papierów, tutoriali, wiki itd.
- `http://lwn.net/Articles/132196/`
  artykuł o Kprobes
- `http://www.mjmwired.net/kernel/Documentation/kprobes.txt`
  dokumentacja Kprobes
- `http://www.redbooks.ibm.com/abstracts/redp4469.html`
  bardzo fajny podręcznik do SystemTap

Koniec

Dziękuję za uwagę !