Instrumentierung des Kernels
Seminar Leistungsanalyse unter Linux

Hajo Möller

9. April 2012
WS 2011/12

Inhaltsverzeichnis

1 Abstract

Mit der *KProbes-API* werden wir drei mächtige Werkzeuge kennen lernen, die es uns erlauben, den laufenden Kernel dynamisch umzuprogrammieren; kprobe, jprobe sowie kretprobe.
Diese *KProbes* werden zu Modulen kompiliert, welche sich zur Instrumentierung in den Kernel laden lassen.

- *kprobe*: wird auf genau eine Instruktion gesetzt; bietet Handler, in denen vor und nach Ausführung der Instruktion lesend auf Stack und Register zugegriffen werden kann.
- *jprobe*: wird auf den Eintrittspunkt einer Funktion gesetzt; an die Funktion übergebene Parameter sind lesbar.
- *kretprobe*: wird auf die Rücksprungadresse einer Funktion gesetzt; Register und Stack, also auch Rückgabewert der Funktion, sind dort sichtbar.

Im zweiten Abschnitt wird auf einen konkreten Anwendungsfall von *kprobes* eingegangen, das Instrumentierungswerkzeug *Systemtap* wird vorgestellt. Dazu wird an kurzen Beispielen die Benutzung der Skriptsprache und des zugehörigen Interpreters dargestellt.

2 Einführung

2.1 Grundlagen

Das Wort *Linux* bezeichnet in erster Linie einen Betriebssystemkern oder *Kernel*. Umgangssprachlich wird unter *Linux* allerdings ein ganzes, auf diesem Kernel basierendes System verstanden. Im Folgenden wird die Bezeichnung *Kernel* synonym zu *Linux*, also dem Betriebssystemkern, gebraucht.

Was ist der Kernel? Der Kernel bezeichnet den Betriebssystemkern, also die Grundkomponente eines Betriebssystems. Er ist verantwortlich für Hardwareabstraktion, die Speicher- und Prozessverwaltung, das Gerätemanagement sowie die Verwaltung von Eingabe/Ausgabe. Linux ist ein monolithischer Kernel, das heißt dass sämtliche Komponenten des Kernels in der höchstprivilegierten Sicherheitsstufe der CPU, im Ring 0 beziehungsweise *Kernel Mode*, laufen. Benutzeranwendungen laufen im Ring 3 oder *User Mode* und sind unprivilegiert; sie kommunizieren über Systemaufrufe, genannt *Syscalls*, mit dem Kernel.
Linux ist ein modularer Kernel, es können also nicht für den Systemstart notwendige Kernelkomponenten, *Module*, während des Betriebs hinzugefügt und wieder entfernt werden.
Alle Codebeispiele basieren auf dem aktuell als stabil markierten Kernel, Linux 3.3.1[1].

Was bedeutet Instrumentierung? Unter *Instrumentierung* versteht man das optionale Hinzufügen von Debugging[2]- und Tracing[3]-Anweisungen zu dem Quelltext eines Softwareprodukts. Durch diese Anwendungen wird der interne Programmstatus extern sichtbar, es lassen sich also unter anderem Aussagen über die Performanz des instrumentierten Produkts treffen. Hier bezeichnet Instrumentierung außer dem Einfügen von Tracinganweisungen auch die Auswertung der entsprechenden Ausgaben.

2.2 Instrumentierung im Userspace

Der *Userspace* kennzeichnet den Speicherbereich, in dem sich Benutzeranwendungen befinden. Sie sind unprivilegiert und können üblicherweise vom ausführenden Benutzer selbst modifiziert werden. Instrumentierung im Userspace dient der Beantwortung von Fragen wie:

- „Warum (und wo) hängt mein Programm?“
- „Welche Syscalls benutzt das Programm, und wofür?“
- „Wo befinden sich Bottlenecks[4], die ich noch beheben kann?“

Einfache Instrumentierung im Userspace ist durch systematisches Hinzufügen von Ausgabeanweisungen wie `printf()` möglich, bei komplexeren Problemen

[1] http://www.kernel.org/pub/linux/kernel/v3.0/linux-3.3.1.tar.bz2
[2] Debugging: Das Suchen und Beheben von Fehlern in Soft- und Hardware
[3] Tracing: Die Analyse eines Programms durch Ausgabe von Informationen zu Funktionsaufrufen
[4] Bottleneck: Programmteil, der die Gesamtleistung des Produkts unnötig stark verschlechtert

ist man mit dedizierter Debugging- oder Tracingsoftware wie *gdb* oder *valgrind* deutlich zuverlässiger und schneller am Ziel.

2.3 Instrumentierung im Kernel

Inklusive Makefiles, Kommentare etc. besteht der aktuelle Kernel (Version 3.3.1) aus 15 166 960 Zeilen Quelltext[5], wodurch „strategisches“ Platzieren von `printk()`[6] sehr unübersichtlich und aufwendig bis unmöglich ist.
Selbst bei kurzen, übersichtlichen Abschnitten des Kernelcodes sollte nicht mit von Hand eingebauten `printk()` gearbeitet werden, da gegebenenfalls der komplette Kernel neu kompiliert und anschließend das System neu gestartet werden müsste, was zu einem massiven[7] Zeitaufwand führt.
Stattdessen sollte Kernelinstrumentierung durch bereits in den Kernel eingebaute Grundlagen oder durch in den laufenden Kernel zu ladende Module dynamisch erfolgen.
Diese Ausarbeitung behandelt ausschließlich die dynamische Instrumentierung des Kernels durch *Kernel Probes* und bietet eine grundlegende Einführung in *Systemtap*. Auf die bereits im Kernel-Quelltext vorhandenen *Tracepoints* und andere Hilfsmittel wird nicht weiter eingegangen.

2.4 Geschichte

Linux wurde zwar seit 1991 entwickelt, aus verschiedenen Gründen (relative Unbekanntheit, fehlendes Vertrauen von Firmen) spielt dessen Instrumentierung erst seit etwa 1999 eine Rolle. Zu dem Zeitpunkt war Linux endgültig in der Geschäftswelt angekommen: Dell begann, Systeme mit Linux zu verkaufen, IBM und Redhat gingen eine Partnerschaft ein, Redhat ging an die Börse, Microsoft attackierte Linux, ... [1]
So kam es zu folgender Entwicklung:

- Juli 2002, Linux 2.5.26: *Kernel Probes* (dynamische Instrumentierung) werden offiziell in Linux aufgenommen
- Jan. 2008, Linux 2.6.24: *Kernel Marker* (statische Instrumentierung) werden aufgenommen, sollten KProbes ergänzen und teilweise ablösen
- Dez. 2008, Linux 2.6.28: *Tracepoints* (statische Instrumentierung) werden aufgenommen, um Schwächen von *Kernel Markern* zu beheben
- Dez. 2009, Linux 2.6.32: *Kernel Marker* wurden durch *Tracepoints* ersetzt und werden endgültig aus dem Tree entfernt

Dynamische (relativ langsam, dafür universell einsetzbar) und statische (relativ schnell, nur an festen Punkten im Code) Instrumentierung ergänzen sich gut, Toolkits wie *LTTng* verwenden deshalb beide Methoden zur Instrumentierung[4].

[5] `find linux-3.3.1/ -type f -not -regex '\./\.git.*' | xargs cat | wc -l`
[6] `printk()` schreibt, analog zu `printf()`, formatierten Text in den Kernelbuffer
[7] Mit einem i7 640LM mit 8 GB RAM dauert das Kompilieren und Rebooten etwa 40 Minuten

3 Kernel Probes

3.1 Geschichte

2000 begann die IBM mit der Entwicklung eines Toolkits zur Kernelinstrumentierung, Dynamic Probes bzw. *dprobes*[2], das auf dynamischer Instrumentierung des Kernels basiert. *Dprobes* setzt als Anwendung auf die Low-Level-API *Kernel Probes* auf.
Das Dprobes-Toolkit sollte ursprünglich die Instrumentierung des Kernels erleichtern, erwies sich dann aber als zu instabil und komplex als dass man es in den offiziellen Kerneltree mit aufnehmen wollte. Stattdessen wurde 2002[8] die KProbes-API in den Tree aufgenommen, IBM stellte die Arbeit an Dprobes bis 2005 ein und schob das Projekt auf das Open-Source-Verzeichnis sourceforge.net[3] ab.
Die *Kernel Probes*-API, ab hier kurz *KProbes* genannt, befindet sich also seit Version 2.5.26 im Kernel und wird weiterhin genutzt und gepflegt.

3.2 Konzepte

KProbes[6] erlauben dynamische Instrumentierung des Kernels durch die Benutzung von selbstkompilierten Kernelmodulen, mit denen Teile des aktuell laufenden Kernels umgeschrieben werden.
Eine Probe bezeichnet einen Messpunkt, also ein näher zu betrachtendes Ereignis, auf das gewartet wird. Sie wird in den Kernel eingefügt, in dem das die Probe enthaltende Modul geladen wird. Entfernt werden Probes entsprechend durch entfernen ihrer jeweiligen Module.
Der genaue Aufbau von Kernelmodulen soll hier nicht behandelt werden. Wichtig ist nur, dass die (De-)Registrierung der KProbes über die Funktionen
`register_kprobe(struct kprobe *kp)`
beziehungsweise
`unregister_kprobe(struct kprobe *kp)`
erfolgt.

3.3 Kernel-Konfiguration

Damit *KProbes* in den laufenden Kernel integriert werden können, sollten folgende Konfigurationsoptionen vor dem Kompilieren des Kernels in dessen `.config` gesetzt werden:

`CONFIG_KPROBES=Y` – *KProbes* müssen aktiviert sein
`CONFIG_HAVE_KPROBES=Y` – wie `CONFIG_KPROBES`, seit 2.6.25 erforderlich
`CONFIG_MODULES=Y` – der Kernel muss erlauben, Module nachzuladen
`CONFIG_MODULE_UNLOAD=Y` – ferner ist es nützlich, Module auch wieder aus dem Kernel entfernen zu können
`CONFIG_KALLSYMS=Y` – optional, lädt (fast) alle Debuggingsymbole[9] mit in den Kernel
`CONFIG_KALLSYMS_ALL=Y` – optional, lädt alle verfügaren Debuggingsymbole

[8] Linux 2.5.26, Juli 2002
[9] Debuggingsymbole: Informationen, die beim Debuggen benötigt werden. Namen von Variablen und Funktionen, erweiterte Informationen über den Stack usw.

Registrierte KProbes sind in `/sys/kernel/debug/kprobes/list` sichtbar. KProbes lassen sich aktivieren, in dem `/sys/kernel/debug/kprobes/enabled` entsprechend gesetzt wird: `echo 1 > /sys/kernel/debug/kprobes/enabled`. Deaktivieren erfolgt analog: `echo 0 > /sys/kernel/debug/krpobes/enabled`.

3.4 `struct kprobe *kp`

Wie wir in **3.2 Konzepte** gesehen haben, wird bei der Registrierung und Deregistrierung einer KProbe das `struct kprobe *kp` übergeben[10]:

```
1  struct kprobe {
2      [...]
3      /* location of the probe point */
4      kprobe_opcode_t *addr;
5
6      /* Allow user to indicate symbol name of the probe point */
7      const char *symbol_name;
8
9      /* Offset into the symbol */
10     unsigned int offset;
11
12     /* Called before addr is executed. */
13     kprobe_pre_handler_t pre_handler;
14
15     /* Called after addr is executed, unless... */
16     kprobe_post_handler_t post_handler;
17
18     /*
19     /* ... called if executing addr causes a fault (eg. page fault).
20      * Return 1 if it handled fault, otherwise kernel will see it.
21      */
22     kprobe_fault_handler_t fault_handler;
23     [...]
24 };
```

Listing 1: Ausschnitt `struct kprobe`

Die Angabe von `addr` oder `symbol_name` ist zwingend erforderlich.
Soll die KProbe auf eine bekannte Speicheradresse gesetzt werden, so übergibt man diese Adresse als `addr` in dem Struct.
Möchte man stattdessen die KProbe auf ein benanntes Symbol setzen, so belegt man `symbol_name` mit dem Namen des Symbols. Falls auf eine bestimmte Adresse *nach* einem benannten Symbol gesetzt werden soll, gibt man mit `symbol_name` den Symbolnamen und mit `offset` den Versatz der KProbe hinter dem Aufruf des Symbols an.
In **3.5.1 kprobe** wird auf die Verwendung von `pre_handler()` und `post_handler()` eingegangen, mehr siehe **3.6 Beispiele**.
Alle Handlerfunktionen sind optional, sie müssen also für die korrekte Verarbeitung der KProbe nicht angegeben werden.

3.5 Varianten

KProbes sind Breakpoint[11]-basierte, dynamische Messpunkte im Kernel.
Es gibt aktuell drei verschiedene Varianten von *KProbes*

- *kprobe* – „Kernel Probe“, der einfachste Fall

[10] `linux-3.3.1/include/linux/kprobes.h:73`

[11] Breakpoint: Speziell markierte Stelle im Programmcode, an der der Prozessor die Ausführung des Codes unterbricht und (manuell oder automatisch) das Programm analysiert wird

- *jprobe* – „Jump Probe“, erweiterte *kprobe*, um Funktionsparameter sichtbar zu machen
- *kretprobe* – „Kernel Return Probe“, erweiterte *kprobe*, in der Rückgabewerte sichtbar sind

3.5.1 kprobe

Die einfache *kprobe* ist eine als Breakpoint auf eine Instruktion[12] an einer bestimmten Speicheradresse[13] gesetzte Probe.

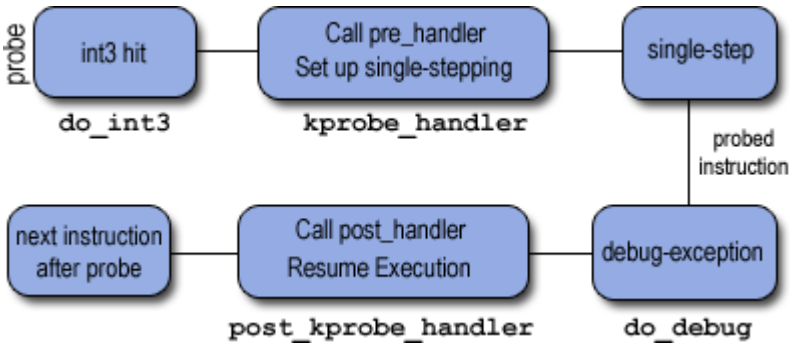

Abbildung 1: Ausführung einer KProbe[5]

Wird die geprobete Instruktion ausgeführt[14] wird über einen Interrupt[15] die Behandlung des Breakpoints[16] gestartet. Die Kernelfunktion `do_int3()`[17], wechselt in einen unterbrechungsfreien[18] Modus und ruft die Funktion `kprobe_handler()` auf, welche prüft, ob an der gebreakten Adresse eine *kprobe* gesetzt wurde. Ist dies der Fall, so sichert `kprobe_handler()` den aktuellen Registersatz[19] und den Stack[20] der CPU und ruft dann selbst eine benutzerdefinierte Funktion, `pre_handler()` auf, sofern sie definiert wurde.
In `pre_handler()` sind die Prozessorregister unmittelbar vor Ausführen der geprobeten Instruktion sichtbar und können bei Bedarf modifiziert werden. Nach vollständigem Abarbeiten von `pre_handler()` veranlasst `kprobe_handler()` den Prozessor, per Single-Stepping[21] eine Kopie der geprobeten Instruktion auszuführen.

[12] Instruktion: Anweisung an den Prozessor, in Maschinencode
[13] Speicheradresse: „Ort“, an dem sich ein Datum oder eine Instruktion im Speicher befindet
[14] heißt: der Befehlszähler ist gleich der Probe-Adresse
[15] Interrupt: kurze Hardware-Unterbrechung der CPU, ein Signal das sofort von der CPU bearbeitet werden muss
[16] x86-Architektur: `int3`
[17] do_int3() wird in `linux-3.3.1/arch/x86/kernel/traps.c:302` definiert
[18] unterbrechungsfrei: der Prozessor kann nicht durch einen weiteren Interrupt unterbrochen werden, sondern arbeitet die aktuellen Instruktionen am Stück ab
[19] Registersatz: alle Register, also alle Speicher die direkt in der CPU liegen
[20] Stack: vom Betriebssystem benutzter Speicher, um Speicherbereiche einzelner Prozesse/-Funktionen zu verwalten und zu schützen
[21] Single-Stepping: Ausführung von Maschinencode in einzelnen Schritten, nach jeder Instruktion stoppt der Prozessor und wartet auf ein Signal, den nächsten Schritt zu machen. Verantwortlich ist die Kernelfunktion `do_debug()`

Single-Stepping ist erforderlich, damit der Programmablauf nicht erneut unterbrochen werden kann und damit die *kprobe* Kontrolle über die Register und den Stack behält. Es ist zu beachten, dass die Instruktion nun im Kernelmodus ausgeführt wird, somit verhindert das Single-Stepping der Kopie auch, dass die Instruktion das gesamte Programm in den Kernelmodus verschiebt und es so privilegierten Status erhält.
Anschließend wird, da die CPU noch im Single-Stepping-Modus ist, `do_debug()`[22] per Auslösen einer Exception[23] die Kontrolle an die Funktion `post_kprobe_handler()` übergeben und so das Single-Stepping beendet.
`post_kprobe_handler()` ruft nun die optionale benutzerdefinierte Funktion `post_handler()` auf, welche genutzt werden kann, um die Registerinhalte nach Ausführen der Instruktion zu überwachen. Anschließend wird der Stack wiederhergestellt und der Programmablauf normal weitergeführt.
So lässt sich zum Beispiel über `printk()` im `pre_handler()` und `post_handler()` die bei der Ausführung der *kprobe* verbrachte Zeit messen.
Tritt während der Ausführung eine Exception, wie zum Beispiel ein *Page Fault*[24] auf, so wird die Funktion `fault_handler()` aufgerufen. In `fault_handler()` kann die Exception behandelt werden. Wird die Exception nicht in `fault_handler()` behandelt, muss die Funktion den Wert `0` zurückgeben, damit die Exception an den Kernel weitergeleitet wird.

3.5.2 jprobe

Eine *jprobe* ist, wie in **3.5 Varianten** bereits angemerkt, eine Erweiterung einer *kprobe*, in der zusätzlich noch die an die geprobete Funktion übergebenen Parameter sichtbar sind.
JProbes werden mit `register_jprobe(struct jprobe *jp)` im Kernel registriert, das `struct jprobe *jp` beinhaltet außer einem `struct kprobe *kp` noch `entry`, einen Zeiger auf eine benutzerdefinierte Funktion[25]:

```
1 struct jprobe {
2     struct kprobe kp;
3     void *entry;    /* probe handling code to jump to */
4 };
```

Listing 2: Definition `struct jprobe`

Damit eine *JProbe* erfolgreich registriert werden kann, ist zwingend erforderlich, dass `addr` (siehe **3.4 `struct kprobe *kp`**) auf den *Entry Point*[26] der zu überwachenden Funktion zeigt.
`register_jprobe(struct jprobe *jp)` ruft selbst `register_jprobes(struct jprobe **jps, int num)` auf, welches die Probe registriert[27]:

[22] linux-3.3.1/arch/x86/kernel/traps.c:376
[23] Exception: Ausnahmesituation, über die bestimmte Informationen an andere Funktionen weitergereicht bzw. eskaliert werden
[24] Page Fault (auch Seitenfehler): eine solche Exception wird generiert, wenn ein Prozess auf einen Speicherbereich im virtuellen Speicher zugreift, der sich aktuell nicht im physischen Speicher befindet oder auf den der Prozess nicht zugreifen darf
[25] `linux-3.3.1/include/linux/kprobes.h:158`
[26] Entry Point: die Speicheradresse, an der eine Funktion beginnt. Beim Anfang der Funktionsausführung steht der Befehlszähler auf dieser Adresse
[27] `linux-3.3.1/kernel/kprobes.c:1591`

```
int __kprobes register_jprobes(struct jprobe **jps, int num)
{
    struct jprobe *jp;
    int ret = 0, i;

    if (num <= 0)
        return -EINVAL;
    for (i = 0; i < num; i++) {
        unsigned long addr, offset;
        jp = jps[i];
        addr = arch_deref_entry_point(jp->entry);

        /* Verify probepoint is a function entry point */
        if (kallsyms_lookup_size_offset(addr, NULL, &offset) &&
            offset == 0) {
            jp->kp.pre_handler = setjmp_pre_handler;
            jp->kp.break_handler = longjmp_break_handler;
            ret = register_kprobe(&jp->kp);
        } else
            ret = -EINVAL;

        if (ret < 0) {
            if (i > 0)
                unregister_jprobes(jps, i);
            break;
        }
    }
    return ret;
}
EXPORT_SYMBOL_GPL(register_jprobes);

int __kprobes register_jprobe(struct jprobe *jp)
{
    return register_jprobes(&jp, 1);
}
EXPORT_SYMBOL_GPL(register_jprobe);
```

Listing 3: Implementierung von `register_jprobe()`

Über `register_jprobes()` können so mehrere jprobes mit einer Zeile registriert werden, wenn ein Array von Pointern auf JProbes (als `struct jprobe *jps`) angelegt und übergeben wird. Gleiches gilt auch für `register_kprobes()` und `register_kretprobes()`.

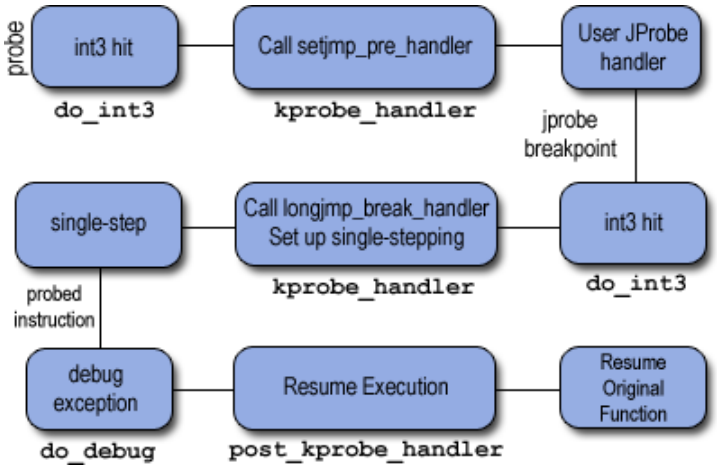

Abbildung 2: Ausführung einer JProbe[5]

Wie wir sehen, werden *JProbes* als normale *KProbes* registriert, wobei `pre_handler()` und `post_handler()` auf die fest definierten Funktionen `setjmp_pre_handler()` bzw. `longjmp_break_handler()` gesetzt werden.
Sobald die *jprobe* ausgelöst[28] wird, wird von der bereits bekannten Kernelfunktion `kprobe_handler()` zum Sichern aller Register und einer „großzügigen“[29] Portion des Stacks die Funktion `setjmp_pre_handler()`[30] aufgerufen. Dort wird der Befehlszeiger auf `entry` aus `struct jprobe *jp` gesetzt, an dieser Adresse muss sich eine Funktion mit derselben Signatur[31] wie die geprobete Funktion befinden:

```
1  int __kprobes setjmp_pre_handler(struct kprobe *p, struct pt_regs *regs)
2  {
3      struct jprobe *jp = container_of(p, struct jprobe, kp);
4      unsigned long addr;
5      struct kprobe_ctlblk *kcb = get_kprobe_ctlblk();
6
7      kcb->jprobe_saved_regs = *regs;
8      kcb->jprobe_saved_sp = stack_addr(regs);
9      addr = (unsigned long)(kcb->jprobe_saved_sp);
10
11     /*
12      * As Linus pointed out, gcc assumes that the callee
13      * owns the argument space and could overwrite it, e.g.
14      * tailcall optimization. So, to be absolutely safe
15      * we also save and restore enough stack bytes to cover
16      * the argument area.
17      */
18     memcpy(kcb->jprobes_stack, (kprobe_opcode_t *)addr,
19            MIN_STACK_SIZE(addr));
20     regs->flags &= ~X86_EFLAGS_IF;
21     trace_hardirqs_off();
22     regs->ip = (unsigned long)(jp->entry);
23     return 1;
24 }
```

Listing 4: `setjmp_pre_handler()`

In dieser Funktion sind nun die übergebenen Parameter sichtbar. Änderungen sind zwar möglich, werden später aber rückgängig gemacht. Wichtig ist hier, dass unmittelbar vor dem Verlassen von dieser Funktion die Funktion `jprobe_return` aufgerufen wird. `jprobe_return` löst einen Breakpoint aus, welcher von `kprobe_handler()` bearbeitet wird, indem über die Funktion `longjmp_break_handler()` Register und Stack auf den vorherigen Zustand zurücksetzt werden:

[28] das heißt: sobald der Befehlszähler auf der definierten Adresse steht

[29] Die aufgerufene Funktion besitzt ihre Argumente, kann sie also ändern. Damit dies keine negativen Auswirkungen auf den weiteren Programmverlauf hat, sichert `kprobes_handler()` bis zu `MAX_STACK_SIZE` vom Stack. Auf i386 sind das 64 Bytes

[30] `linux-3.3.1/arch/x86/kernel/kprobex.c:1039`

[31] Signatur: Rückgabewert und Parameter (Anzahl + Typ)

```
int __kprobes longjmp_break_handler(struct kprobe *p, struct pt_regs *
    regs)
{
    struct kprobe_ctlblk *kcb = get_kprobe_ctlblk();
    u8 *addr = (u8 *) (regs->ip - 1);
    struct jprobe *jp = container_of(p, struct jprobe, kp);

    if ((addr > (u8 *) jprobe_return) &&
        (addr < (u8 *) jprobe_return_end)) {
        if (stack_addr(regs) != kcb->jprobe_saved_sp) {
            struct pt_regs *saved_regs = &kcb->jprobe_saved_regs;
            printk(KERN_ERR
                   "current sp %p does not match saved sp %p\n",
                   stack_addr(regs), kcb->jprobe_saved_sp);
            printk(KERN_ERR "Saved registers for jprobe %p\n", jp);
            show_registers(saved_regs);
            printk(KERN_ERR "Current registers\n");
            show_registers(regs);
            BUG();
        }
        *regs = kcb->jprobe_saved_regs;
        memcpy((kprobe_opcode_t *)(kcb->jprobe_saved_sp),
               kcb->jprobes_stack,
               MIN_STACK_SIZE(kcb->jprobe_saved_sp));
        preempt_enable_no_resched();
        return 1;
    }
    return 0;
}
```

Listing 5: `longjmp_break_handler()`

Da die *jprobe* direkt als *kprobe* implementiert ist, wird die geprobete Instruktion wie eine normale *kprobe* verarbeitet.

3.5.3 kretprobe

KRretProbes werden benutzt, um den Rückgabewert einer Funktion sichtbar zu machen. Sie werden im Kernel über `register_kretprobe(struct kretprobe *rp)`[32] eingebunden, welches eine *kprobe* auf den *Entry Point* der zu überwachenden Funktion setzt:

```
int __kprobes register_kretprobe(struct kretprobe *rp)
{
    int ret = 0;
    struct kretprobe_instance *inst;
    int i;
    void *addr;

    [...]

    rp->kp.pre_handler = pre_handler_kretprobe;
    rp->kp.post_handler = NULL;
    rp->kp.fault_handler = NULL;
    rp->kp.break_handler = NULL;
    [...]
}
```

Listing 6: Aussschnitt von `register_kretprobe()`

Sobald diese *KProbe* auslöst, wird die Rücksprungadresse der Funktion gesichert und durch ein „Trampolin“[33] ersetzt.
`pre_handler_kretprobe()` selbst ruft `arch_prepare_kretprobe()`[34] auf, welches die Sicherung und Ersetzung der Returnadresse übernimmt:

[32] `linux-3.3.1/kernel/kprobes.c:1699`
[33] Trampolin: eine spezielle Instruktion (NOP), auf der eine weitere KProbe registriert wurde
[34] linux-3.3.1/arch/kernel/kprobes.c:445

```
1  void __kprobes arch_prepare_kretprobe(struct kretprobe_instance *ri,
2                        struct pt_regs *regs)
3  {
4      unsigned long *sara = stack_addr(regs);
5
6      ri->ret_addr = (kprobe_opcode_t *) *sara;
7
8      /* Replace the return addr with trampoline addr */
9      *sara = (unsigned long) &kretprobe_trampoline;
10 }
```

Listing 7: `arch_prepare_kretprobe`

In Zeile 8 wird die Returnadresse durch die des Trampolins ersetzt. Das Trampolin `kretprobe_trampoline()` ist direkt in Assembler programmiert, auf ein Listing wird hier verzichtet. Es pusht (fast) alle Register auf den Stack, ruft die Funktion `trampoline_handler()` auf und stellt anschließend den Registersatz wieder her.
Wird das Trampolin getroffen, unterbricht die CPU und KProbes' `trampoline_handler()` ruft eine im `struct kretprobe` benutzerdefinierte Handlerfunktion[35] `entry_handler()` auf, die ähnlich wie `pre_handler()` in **3.5.1 kprobe** benutzt werden kann.
Die auf den *Entry Point* gesetzte *kprobe* wird verarbeitet, heißt die CPU arbeitet die Instruktion an `addr` per Single-Stepping ab, anschließend wird die vollständige geprobete Funktion normal ausgeführt.
Erst beim Erreichen der Rücksprungadresse löst die anfangs gesetzte KProbe aus, und die im `struct kretprobe` definierte Funktion `handler()` wird aufgerufen.
Die Rücksprungadresse der hier aufgerufenen Funktion wird durch `kretprobe_trampoline()` und `trampoline_handler()`[36] auf die ursprüngliche Rücksprungadresse der geprobeten Funktion gesetzt, gleichzeitig wird die hier gesetzte KProbe deregistriert:

```
1  static __used __kprobes void *trampoline_handler(struct pt_regs *regs)
2  {
3      [...]
4
5      kretprobe_assert(ri, orig_ret_address, trampoline_address);
6
7      [...]
8
9      kretprobe_hash_unlock(current, &flags);
10
11     hlist_for_each_entry_safe(ri, node, tmp, &empty_rp, hlist) {
12         hlist_del(&ri->hlist);
13         kfree(ri);
14     }
15     return (void *)orig_ret_address;
16 }
```

Listing 8: Ausschnitt `trampoline_handler()`

In `handler()` sind der `struct kretprobe *rp` und die CPU-Register sichtbar, Rückgabewerte werden üblicherweise im Register `EAX` abgelegt[7].
Nach Rückkehr aus `handler()` werden Stack und Register auf den ursprünglichen Wert zurückgesetzt, eventuelle Änderungen des Rückgabewerts durch die *kretprobe* sind also nicht von Bestand.

[35] Wie in **3.4** `struct kprobe *kp` erwähnt, alle Handlerfunktionen sind optional
[36] `linux-3.3.1/arch/kernel/kprobes.c:705`

3.6 Beispiele

Weitere Beispiele zu allen Varianten von *kprobes* finden sich im Kernel-Quelltext, unter `linux-3.3.1/samples/kprobes/`, dort liegt auch eine `Makefile`.
Hier eine *kprobe*, die einen `pre_handler()` ausführt, sobald ein Prozess forkt:

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kprobes.h>

/* For each probe you need to allocate a kprobe structure */
static struct kprobe kp = {
    .symbol_name    = "do_fork",
};

/* kprobe pre_handler: called just before the probed instruction is
    executed */
static int handler_pre(struct kprobe *p, struct pt_regs *regs)
{
    printk(KERN_INFO "pre_handler: p->addr = 0x%p, ip = %lx, flags = 0x%
        lx\n",
           p->addr, regs->ip, regs->flags);
    return 0;
}

static int __init kprobe_init(void)
{
    int ret;
    kp.pre_handler = handler_pre;

    ret = register_kprobe(&kp);
    if (ret < 0) {
        printk(KERN_INFO "register_kprobe failed, returned %d\n", ret);
        return ret;
    }
    printk(KERN_INFO "Planted kprobe at %p\n", kp.addr);
    return 0;
}

static void __exit kprobe_exit(void)
{
    unregister_kprobe(&kp);
    printk(KERN_INFO "kprobe at %p unregistered\n", kp.addr);
}

module_init(kprobe_init)
module_exit(kprobe_exit)
MODULE_LICENSE("GPL");
```

kp_dofork.c

Die zugehörige Makefile (hier wird auch `kp_helloworld.c` erwähnt, welches in **4.5 Vergleich mit Kernel Probes** angegeben ist):

```
obj-m := kp_helloworld.o kp_dofork.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
default:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
clean:
    rm -f *.mod.c *.ko *.o modules.order Module.symvers .*.cmd
    rm -rf .tmp_versions/
```

Makefile

4 Systemtap

Systemtap[8] ist der Name sowohl eines Instrumentierungswerkzeugs als auch der zugehörigen Skriptsprache. Es setzt auf die Kernel Probes-API auf, heißt es benutzt intern kprobe, jprobe und kretprobe um den Kernel zu instrumentieren. Mit Systemtap ist es schnell und sicher möglich, Daten zu *Events*, auch *Probepunkte* genannt, aus dem Kernel zu sammeln und zu verarbeiten. *Events* können sowohl Timer als auch Kernelfunktionen sein. Beim Auslösen eines Events wird der zugehörige *Handler* aufgerufen, in dem die gewünschten Daten abgegriffen und verarbeitet werden können.

4.1 Geschichte

Systemtap[37] wurde als Reaktion auf IBMs Dprobes und SUNs 2005 veröffentlichtes DTrace-Toolkit entwickelt. DTrace ermöglicht eine relativ leichte und ausführliche Instrumentierung des Solaris-Kernels, was bis zu dem Zeitpunkt unter Linux nicht möglich war.
Da DTrace unter der CDDL[38] veröffentlicht wurde, einer zwar „freien“, jedoch mit der GPL[39] inkompatiblen Lizenz, konnte DTrace nicht ohne weiteres auf Linux portiert und in den offiziellen Kernel integriert werden. Als Alternative dazu werden ebenfalls seit 2005 die Projekte *Systemtap* und *LTTng*[40] entwickelt, LTTng soll nicht Teil dieser Ausarbeitung sein.

4.2 Grundlagen

Systemtap setzt auf dynamische und statische Instrumentierung des Kernels, dynamische Instrumentierung wird per *Kernel Probes* realisiert, statische durch *Tracepoints*. Die genaue Implementierung von Tracepoints ist ebenfalls nicht Teil dieser Arbeit.
Dem Benutzer wird durch Bereitstellung einer einfachen, aber relativ mächtigen, Skriptsprache ein Werkzeug in die Hand gegeben mit dem sich der Kernel durch wenige Zeilen Code instrumentieren lässt.
Systemtap-Skriptdateien haben üblicherweise die Dateiendung `.stp`, der Interpreter wird mit `stap` aufgerufen:

```
1 probe begin {
2     print("hello world!\n")
3     exit()
4 }
```

Listing 9: `hello-world.stp`

Der Aufruf durch `stap hello-world.stp` gibt nach kurzer Zeit die Zeile `"hello world!"` auf der Konsole aus, bevor `stap` beendet.

[37] http://sourceware.org/systemtap
[38] Common Development and Distribution License, eine Lizenz von SUN
[39] GNU General Public License, eine freie Lizenz von der Free Software Foundation. Linux steht unter der GPL
[40] http://lttng.org/

4.2.1 Workflow

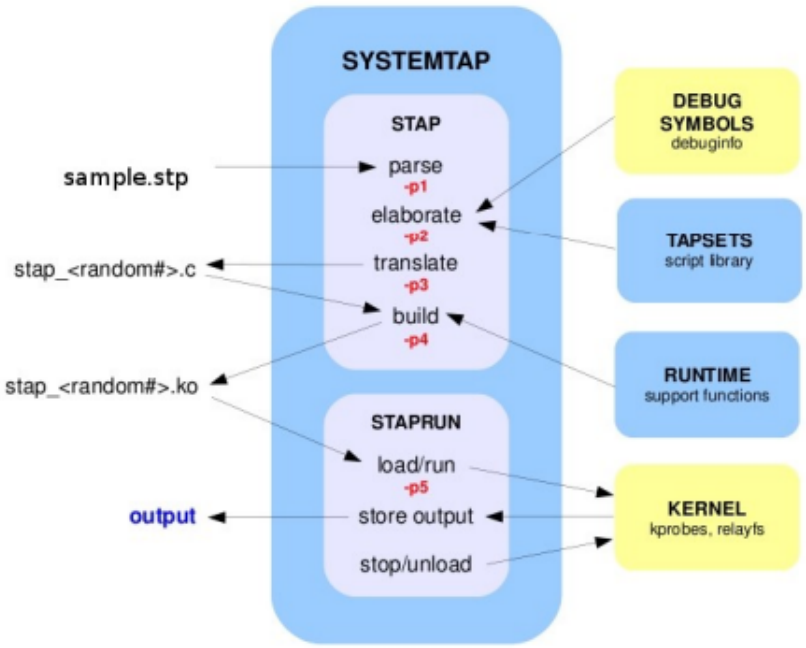

Abbildung 3: Ausführung eines Systemtap-Skripts[9]

Die Ausführung eines Systemtap-Skripts findet in 5 Schritten statt:

1. *Parsen*; das übergegebene Skript und eventuell vorhandene *Tapsets* werden eingelesen, ein Parsebaum wird generiert.
2. *Elaborieren*; Symbolnamen[41] werden rekursiv aufgelöst, nicht auflösbare Symbolnamen werden in den bereits eingelesenen Tapsets gesucht. Wenn eine Datei eines Tapsets ein Symbol auflöst wird die Datei vollständig in das Skript integriert. Der Interpreter versucht, alle Probes und Funktionen zu optimieren.
3. *Übersetzen*; das im Speicher geparste und ergänzte Skript wird vom Interpreter in C-Code umgewandelt und in temporären Dateien abgelegt, zusätzlich wird eine `Makefile`[42] generiert, mit der das Kernelmodul gebaut wird.
4. *Kompilieren*; der Interpreter führt `make` im temporären Verzeichnis aus und erstellt so das Kernelmodul.
5. *Ausführen*; das erstellte Modul wird in den Kernel geladen, die Instrumentierung wird durchgeführt. Anschließend entläd Systemtap das Modul und räumt auf[43].

[41] Symbolnamen hier: Variablen, Funktionen und Probepunkte
[42] Makefile: Datei, die Regeln zum Kompilieren (wie Compilerparameter) eines Programms beinhaltet
[43] temporäre Dateien werden entfernt, Logdateien zusammengefasst und gefiltert, ...

4.3 Skriptsprache

Im Folgenden wird eine kurze, unvollständige Einführung in die Skriptsprache geboten. Die aktuelle Dokumentation ist unter `http://sourceware.org/systemtap/documentation.html` zu finden.

4.3.1 Probepunkte

Man unterteilt *Probepunkte* oder *Events* in synchrone und asynchrone Punkte. Synchrone Probepunkte bezeichnet auf spezifischen Kernelcode, also Instruktionsadressen des Kernels, gesetzte Probes. Unter asynchronen Probepunkten werden Probes verstanden, die durch Timer, Zähler und ähnliche Konstrukte ausgelöst werden.

Einige Beispielevents:

`begin`: Start der Systemtap-Session, also sobald das kompilierte Modul geladen wurde.

`end`: Ende der Systemtap-Session, unmittelbar bevor das Modul entladen wird.

`timer.ms(200)`: Ein Timer, der alle 200 ms die Probe auslöst.

`kernel.function("sys_open")`: Eintrittspunkt von `sys_open()` im Kernel, löst aus sobald diese Funktion aufgerufen wird.

Mehr Probepunkte werden in `man 3 stapprobes` aufgeführt.

```
1 probe timer.ms(200) { ... }
```

Listing 10: Verwendung von Probepunkten

4.3.2 Vordefinierte Variablen

Vordefinierte Variablen können als Funktionsaufrufe betrachtet werden[44], die den Wert der gewünschten Variablen als Zeichenkette oder Zahl zurückgeben.

Beispiele:

`tid()`: Gibt die aktuelle Thread-ID als Zahl zurück

`execname()`: Gibt den Namen des laufenden Prozesses zurück, wie er unter `ps` erscheint.

`cpu()`: Gibt als Zahl zurück, auf welcher CPU die Probe ausgeführt wird.

`gettimeofday_s()`: Gibt einen Zeitstempel zurück, der Sekunden seit *Epoch*[45] entspricht.

`$$vars`: Gibt, sofern verfügbar, Informationen über an der aktuellen Codestelle sichtbare lokale Variablen an.

Mehr vordefinierte Variablen findet man in `man 3 stapfuncs`.

[44] In der Systemtap-Dokumentation ist von „SystemTap Functions“ die Rede.
[45] Epoch, auch Unixzeit: Zeit in Sekunden, die seit dem 1. Januar 1970 00:00 verstrichen ist

4.3.3 Ausgabe

Systemtap bietet verschiedene Funktionen, um Daten als Text ausgeben zu lassen:
`print(var)` ist die einfachste Methode, um den Inhalt der Variablen `var` auf einer Zeile auszugeben. Zu beachten ist, dass `var` vom Typ Zahl oder Zeichenkette sein muss.
`printf("format", argument1, argument2, ...)` erlaubt die formatierte Ausgabe von Variablen, die Formatierung erfolgt ähnlich wie in C durch die Platzhalter `%d` für Zahlen und `%s` für Zeichenketten.

Beispiel:

```
probe syscall.open {
    print("Der Syscall open() wurde benutzt:")
    printf("%s(%d) open\n", execname(), pid())
}
```

Listing 11: Verwendung von `print()` und `printf()`

Weitere Ausgabefunktionen werden in `man 1 stap`, Sektion `PRINTING` beschrieben.

4.3.4 Kontroll- und Datenstrukturen

Systemtap bietet verschiedene Kontrollstrukturen, mit denen der Programmfluss gesteuert werden kann.
Kommentare werden sowohl in *C*- (`"/* ... */"`), *C++*- (`"// ... "`), als auch in *Shell*-Notation (`"# ... "`) erkannt.

Blöcke bezeichnen zusammengehörende Codezeilen, sie werden in geschweifte Klammern eingefasst: `{ ... }`.

Funktionen sind benannte Blöcke, ein Rückgabewert ist nicht unbedingt erforderlich. Rekursive Funktionen sind erlaubt.

```
function trace_common() {
    printf("%d %s(%d)", gettimeofday_s(), execname(), pid());
}
```

Listing 12: Eine nützliche Funktionsdefinition

Variablen sind entweder lokal oder global sichtbar, also nur in dem Block in dem sie definiert wurden oder im gesamten Systemtap-Skript:

```
foo = "hello world" // ein lokaler String
global array[400] // ein global sichtbares Array mit 400 Elementen
a <<< delta timestamp // ein Aggregate, Aggregates werden im Abschnitt "
    Aggregates" behandelt
```

Listing 13: Verschiedene Variablendefinitionen

4.3.5 „Target Variables“

In synchronen Probepunkten sind „Target Variables“ verfügbar, sie werden genutzt um an der aktuellen Codestelle sichtbare Variablen einzusehen.
Zur Veranschaulichung werden am Beispiel die in der Kernelfunktion `vfs_read()`[46] sichtbaren Variablen ausgegeben und die Bedeutung der Ausgabe erläutert:

```
# stap -L 'kernel.function("vfs_read")'
kernel.function("vfs_read@fs/read_write.c:348") $file:struct file* \
  $buf:char* $count:size_t $pos:loff_t*
```

Wir sehen, dass `vfs_read()` in `fs/read_write.c`, Zeile 348[47] definiert wird. In der Funktion sind folgende Variablen sichtbar:

- `file`, Datentyp `struct file*`, enthält den Zeiger auf die Beschreibung der zu lesenden Datei
- `buf`, Datentyp `char*`, enthält den Zeiger auf den Speicherbereich im Userspace, in den die Inhalte der Datei geschrieben werden sollen
- `count`, Datentyp `size_t`, die Anzahl zu lesender Bytes
- `pos`, Datentyp `loff_t*`, das Offset, also die Position innerhalb der Datei, ab der sie gelesen werden soll

```
1 probe kernel.function("vfs_read") {
2     printf("Zu lesende Bytes: %d\n", $count)
3 }
```

Listing 14: Beispiel zu *Target Variables*

Ein `stap`-Aufruf um sich alle Variablen einer bestimmten Art, wie zum Beispiel *alle* sichtbaren Variablen *aller* Kernelfunktionen, anzeigen zu lassen:
`stap -L 'kernel.functions("*")' | less`

4.3.6 Aggregates

Aggregates sind spezielle Variablen, die sich ähnlich wie eine automatische Liste oder ein Stack verhalten und eine Sammlung von Werten sind. Man kann ausschließlich Werte zu ihnen hinzufügen.
Auf Aggregates sind verschiedene Funktionen, wie die Berechnung des Durchschnittwerts, anwendbar:

```
1 global count_read
2
3 probe kernel.function("vfs_read") {
4     count_read <<< $count
5 }
6
7 probe end {
8     print(@avg(count_read))
9 }
```

Listing 15: Berechnung des Durchschnittswerts an gelesenen Bytes

Mehr zu Aggregates in `man 1 stap`, Sektion `STATISTICS`.

[46] `linux-3.3.1/fs/read_write.c:364`
[47] Das Beispiel wurde unter Fedora mit Linux 3.1.0-7 aufgerufen, daher die unterschiedliche Zeilenausgabe

4.4 Sicherheit

Systemtap bietet im Vergleich zu Kernel Probes stark erhöhte Sicherheitsmechanismen. Wenn man bei der Programmierung einer *kprobe* einen schwerwiegenden Fehler macht ist die Stabilität des gesamten Systems gefährdet, da man fehlerhaften Code mit Kernelprivilegien ausführt.
Mit Systemtap wäre das nicht passiert.
Systemtap führt eine rudimentäre Überprüfung der Skriptdatei vor dem Ausführen durch, außerdem wird beim Erzeugen des *C*-Codes unter anderem darauf geachtet, dass keine Null-Pointer dereferenziert werden und dass keine Divisionen durch 0 erfolgen.
Endlosschleifen und -rekursion stellen auch keine große Gefahr dar, da jede Schleife und jeder Aufruf in Systemtap eine beschränkte Ausführungszeit hat. Wird diese Zeit überschritten wird zur Sicherheit das Modul aus dem Kernel entfernt.
Durch Systemtap kann auch nicht der RAM volllaufen, da sämtlicher benötigter Speicher bereits bei der Modulinitialisierung allokiert wird, eine nachträgliche dynamische Allokation ist nicht möglich.

4.5 Vergleich mit Kernel Probes

Ein Schnellvergleich von Systemtap und Kernel Probes:

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kprobes.h>

static struct kprobe kp = {
        .symbol_name    = "do_fork",
};

static int __init kprobe_init(void)
{
        int ret;

        ret = register_kprobe(&kp);
        if (ret < 0) {
                printk(KERN_INFO "register_kprobe failed, returned %d\n"
                        , ret);
                return ret;
        }
        printk(KERN_INFO "Hello, World!");
        return 0;
}

static void __exit kprobe_exit(void)
{
        unregister_kprobe(&kp);
}

module_init(kprobe_init)
module_exit(kprobe_exit)
MODULE_LICENSE("GPL");
```

Listing 16: *Hello, World!* als Kernel Probe

```
probe begin {
    print("Hello, World!\n")
    exit()
}
```

Listing 17: *Hello, World!* als Systemtap-Skript

Literatur

[1] *LWN 1999 Linux Timeline*; `http://lwn.net/1999/features/Timeline/?month=all`

[2] *`man 8 dprobes`* – *„Dynamic Probes“*; `http://dprobes.sourceforge.net/documentation/man/dprobes/#LICENSE`; LICENSE

[3] *Projektseite „Dynamic Probes“*; `http://sourceforge.net/projects/dprobes/`

[4] *TracingBook : Binary Instrumentation : Dynamic Binary Instrumentation*; `http://lttng.org/tracingwiki/index.php/Dynamic_Binary_Instrumentation`

[5] *An introduction to KProbes*; `http://lwn.net/Articles/132196/`

[6] Die vollständige Dokumentation zu KProbes findet sich im Linux-Quelltext, in der Datei `linux-3.3.1/Documentation/kprobes.txt`; Ferner wurden sämtliche relevanten Quelldateien zu KProbes (u.a. `linux-3.3.1/kernel/kprobes.c` sowie `linux-3.3.1/arch/x86/kprobes.c`) für diese Ausarbeitung betrachtet

[7] *Calling conventions for different C++ compilers and operating systems*; Fog, Agner; `http://agner.org./optimize/calling_conventions.pdf`

[8] *Die offizielle Systemtap-Dokumentation*; `http://sourceware.org/systemtap/documentation.html`; Sowie zugehörige manpages.

[9] *Anwendungen tracen mit Oprofile und Systemtap*; `http://www.admin-magazin.de/Das-Heft/2010/03/Anwendungen-tracen-mit-Oprofile-und-Systemtap`; ADMIN 03/2010