Andrea Manzini

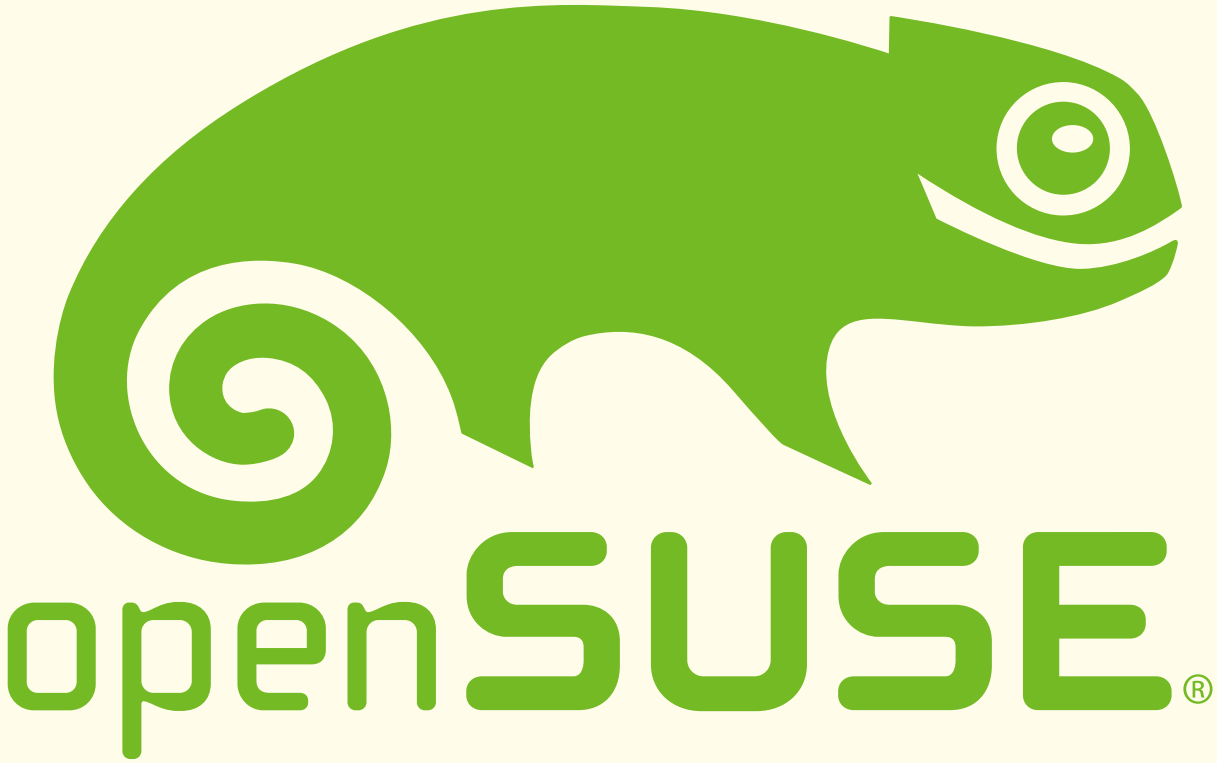
A scatola chiusa

Test Coverage

di binari nativi

con eBPF

Ottobre 2026



andrea.manzini@suse.com



Mi presento

Andrea Manzini
https://ilmanzo.github.io

- Veteran Unix Admin; BOFH
- Software engineer @SUSE
- Open Source Contributor 👈
- Package Maintainer 📦

andrea.manzini@suse.com

Mi occupo (anche) di testare

… ma quanto codice stiamo davvero testando?

Con i sorgenti in mano, è facile:

```
go test -cover ./...                # Go
gcc --coverage prog.c -o prog       # C/C++ (gcov)
```

Ricompili con la strumentazione, lanci i test, ottieni il report riga per riga.

Fine della storia? 🤔 Non proprio…

Quando la scatola è chiusa 📦

Test di sistema / integrazione (es. **openQA** su openSUSE):

- pacchetti RPM già compilati
- binari di terze parti, tool legacy
- demoni complessi (nginx, postgres, systemd…)

Ricompilare non è possibile.

> Domanda: durante questa suite di test, **quali funzioni sono state davvero eseguite?**

Test Coverage

Qualsiasi binario *ELF* è composto da funzioni!

Possiamo contare quante funzioni ci sono: **F**

Possiamo tracciare quante di queste vengono eseguite: **E**

E / F = Coverage!



Demo 1 🎬 F ed E dal vivo

`sample`: un programma C++ già compilato, per noi una scatola chiusa.

```
./demo1.sh
```

1. **F**: quante funzioni contiene
2. **E**: tre "test" in fila, dopo ognuno ricalcoliamo **E / F**
3. Report HTML con le funzioni mai chiamate

Da osservare: **F = 101**, poi la copertura sale a ogni test.

Vicolo cieco #1: GDB + Valgrind 🐌

```
gdb -ex 'info functions' -ex quit /usr/bin/gzip   # tutte le funzioni
valgrind --tool=callgrind /usr/bin/gzip -h        # quelle chiamate
```

- Valgrind **'emula' la CPU**: funziona su binari non modificati
- Con i pacchetti `-debuginfo` ottieni anche i nomi delle funzioni
- `gzip`: da ~4% a ~46% di coverage in 4 test… ma
- **~30× più lento**: i test con rete o timeout falliscono

Vicolo cieco #2: Intel PIN 🔧

Dynamic Binary Instrumentation: inietta codice a runtime nei blocchi base.

- Flessibile e potente
- SDK C++ **proprietario**, ~150 MB
- Solo **x86_64** (niente ARM64, niente RISC-V)
- Overhead ancora alto: 5–10×

Il repo nasce così: nel 2023 come **fork di un pintool** di un altro sviluppatore, ripreso da me nel 2025 (wrapper, report in Go…).

Riassunto dei candidati

Approccio	Serve il sorgente?	Overhead	Note
`gcov` / `go -cover`	**sì**	basso	granularità riga
GDB + Valgrind	no	~30×	emula la CPU
Intel PIN	no	5–10×	SDK proprietario, solo x86_64
eBPF `uprobe_multi`	no	**~1–2%**	kernel ≥ 6.6, x86_64 + ARM64

eBPF in 60 secondi ⏱️

piccoli programmi caricati a runtime **dentro il kernel**, agganciati a eventi.

- Non è un modulo kernel
- un **verifier** li controlla (niente loop infiniti, niente accessi a memoria a caso, niente sleep o mutex)
- Si agganciano a syscall, tracepoint, funzioni del kernel… e **funzioni dei programmi utente**
- Parlano con lo userspace tramite **map** e **ring buffer**

Pensa a JavaScript nel browser: codice "altrui" che gira in un programma grosso, con un vigile che lo controlla prima di farlo partire.

uprobe: un breakpoint che non si ferma 🚦

Il kernel sostituisce la prima istruzione della funzione con un'**istruzione di breakpoint** (es. `int3`):

trap (eccezione) → il tuo handler eBPF → si riprende come se nulla fosse

- Niente `ptrace`, niente `LD_PRELOAD`
- Il file su disco non viene toccato

La rivoluzione di `uprobe_multi` (Linux 6.6+):

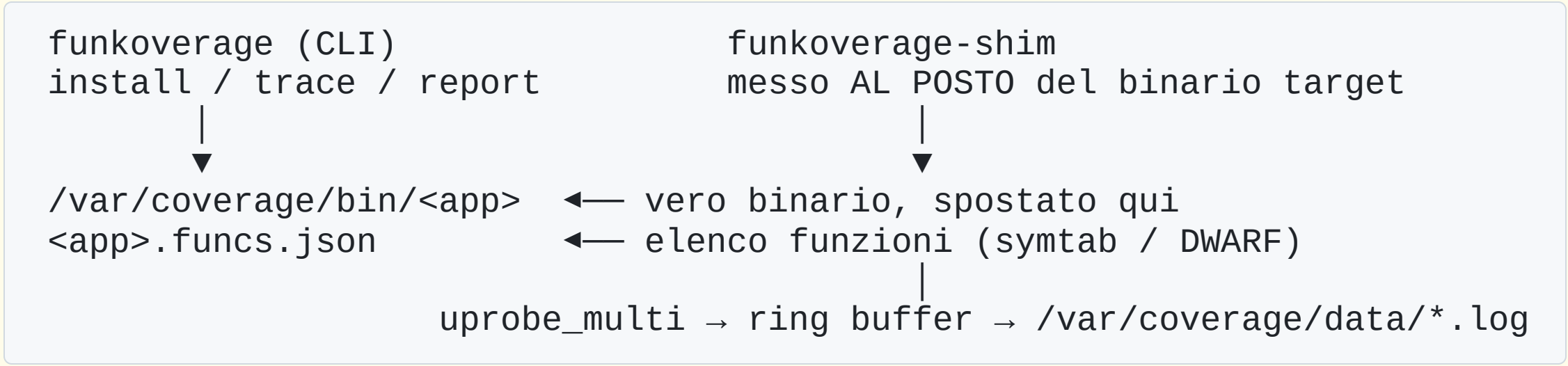
Per tracciare la coverage dobbiamo agganciare *TUTTE* le funzioni di un eseguibile.

Approccio	Syscall per agganciare 8.000 funzioni	Tempi di avvio
uprobe classiche	8.000 (una per ogni singolo offset)	Secondi (lento)
`uprobe_multi`	**1 singola syscall** con un array di offset	**Istantaneo**

Il segreto: il BPF Cookie 🍪

Invece di avere 8.000 programmi eBPF, ne carichiamo **uno solo**. Al momento dell'attach in blocco, assegniamo a ciascun offset un ID univoco (il *cookie*). Quando scatta la *trap*, il kernel passa l'ID al nostro handler, che così riconosce all'istante quale funzione è stata chiamata.

Architettura: due binari, un impostore 🥷



Come funziona ?

`sudo funkoverage install /usr/bin/gzip`:

1. sposta il vero `gzip`
2. elenca le funzioni
3. copia lo shim in `/usr/bin/gzip`
4. `setcap` (CAP_BPF...)

Fase di esecuzione:

1. L'utente lancia il programma normalmente (es. `gzip file.txt`)
2. Quello che viene eseguito è il NOSTRO programma (shim)
3. Lo *shim* lancia un *helper* in background per attivare eBPF
4. Lo *shim* fa `exec` verso il **vero binario** (quindi il PID non cambia)
5. L'helper raccoglie le informazioni in background in modo trasparente

In dettaglio

Mentre il kernel gestisce le *trap* a tempo di record, il processo in userspace:

1. **Mappa le funzioni**: Analizza il binario ELF (Symtab/DWARF) per trovare gli offset delle funzioni.
2. **Prepara il terreno**: Crea le strutture (Mappe), carica il programma eBPF e lo aggancia tramite `uprobe_multi`.
3. **Ascolta**: Resta in attesa dormiente degli eventi (i "cookie") dal **Ring Buffer**.
4. **Salva i risultati**: Traduce l'ID nell'evento nel nome originale della funzione e aggiorna il log.

Essendo guidato dagli eventi (event-driven), non c'è *polling*. L'uso della CPU in userspace è praticamente nullo.

Lato kernel, ci basta *"almeno una volta"*

Alla coverage basta sapere **se** una funzione viene eseguita.

```
SEC("uprobe.multi/probe")
int trace_uprobe(struct pt_regs *ctx)  // (semplificato)
{
    __u32 idx = (__u32)bpf_get_attach_cookie(ctx);   // chi sono?
    __u64 *flag = bpf_map_lookup_elem(&seen, &idx);
    if (!flag) return 0;

    // Compare-And-Swap atomico: 0→1 vince UNA sola CPU
    if (__sync_val_compare_and_swap(flag, 0ULL, 1ULL) != 0)
        return 0;  // già vista: esco subito

    /* solo la PRIMA volta: scrivo l'evento nel ring buffer */
}
```

Eventi verso lo userspace ≤ numero di funzioni, **qualunque sia il runtime**.

Demo 2 🎬 su software vero

```
./demo2.sh
```

1. **curl**: quali funzioni SSL usa? Trace al volo, senza installare nulla
2. **gzip**: shim permanente, tre esecuzioni di fila
3. Report HTML

Da osservare: `gzip -h` ≈ **3%** → percorso d'errore ≈ **11%** → compress/decompress ≈ **46%**. Il tutto in millisecondi.

Il fantasma di `dlopen()` 👻

`ldd` elenca solo le librerie dichiarate. Ma nginx, OpenSSL, PAM, i database caricano **plugin a runtime**: all'installazione non esistono ancora.

La soluzione, *event-driven* (niente polling!):

1. **uretprobe su `dlopen()`**: quando ritorna con successo, evento speciale nel ring buffer
2. Lo shim rilegge `/proc/<pid>/maps` e scopre la `.so` appena mappata
3. Ne legge i simboli e **aggancia le uprobe al volo** (JIT)
4. Tutto prima che il programma chiami `dlsym()`

Prova su nginx: **6.400+ funzioni** di libcrypto/libssl agganciate al volo.

War story: tre lezioni dal campo ⚔️

- 🪪 **Il PID è sacro.** Lo shim lanciava il demone come *figlio*: `systemd` (`sd_notify`), `pg_ctl` , `rpcbind` controllano il PID e impazziscono.
  - Fix: un helper in background attacca eBPF, lo shim fa `exec` **nello stesso PID**
- 🧯 **Mai rompere l'host.** `udev` + seccomp bloccano `perf_event_open()` : lo shim usciva con errore, `nfsrahead` sostituito da uno stub che fallisce, readahead NFS spento in silenzio.
  - Regola: **fail-open**, avvisa ma esegui *sempre* il binario vero
- 🤫 **Sbagliare in silenzio è il peggio.** Numeri falsi senza alcun errore:
  - `glibc-hwcaps` : agganciato il file sbagliato → **0% di coverage**
  - symlink/SONAME: libreria contata due volte
  - `report` letto mentre l'helper scriveva ancora → `flock`

Limiti onesti 🙋

- Solo **function** coverage (nessun dato su righe e branch)
- Funzioni *inline*: conto la funzione che le ingloba
- Servono sempre i simboli: `.symtab` o i pacchetti `-debuginfo`
- **NSS** (`getpwnam`, `libnss_*.so`): glibc usa un `dlopen` interno non esportato → invisibili
- Kernel **≥ 6.6** con BTF; solo x86_64 e ARM64
- L'installazione richiede root

Dentro openQA 📊

`coverage_setup` → **test openQA invariati** → `coverage_report`

- Installa shim e debuginfo, poi gira la suite di sempre
- Con Zoltán Balogh: dal PoC su 7 binari a **141 binari di sistema**, oltre **74.000 funzioni**
- Il report esce come artefatto del job

A destra: report aggregato (dati di esempio del repo)

Aggregate Coverage Report

Generated at: 2026-10-17 19:01:25 CEST

Total Coverage

- **Total Functions:** 40825
- **Total Executed:** 7304
- **Average Coverage:** 17.89%

Image ▲	Total Functions	Called Functions	Coverage
bzip2	37	9	24.3%
cpio	142	57	40.1%
curl	169	52	30.8%
dig	102	45	44.1%
iostat	32	13	40.6%
kmod	201	123	61.2%
libacl.so.1	45	0	0.0%
libaom.so.3	53	0	0.0%
libavif.so.16	99	0	0.0%
libbrotlicommon.so.1.2.0	9	0	0.0%
libbrotlidec.so.1.2.0	14	1	7.1%
libbz2.so.1.0.6	24	13	54.2%
libcom_err.so.2	17	0	0.0%

Come si legge un numero basso 🤷

Media tipica sui binari di sistema: **12–17%.** I test non sono scritti per quei binari.

Caso	Coverage	Perché
`gzip`	3% con `-h`, 46% con uso reale	stesso binario, test diversi
`lp`	0%	una sola funzione tracciabile (`usage`), raggiunta solo dagli errori
nginx (`nginx -t`)	4,6%	801 funzioni su 17.544, librerie di sistema comprese

Il valore sta nella **baseline** e nel **trend**: se la coverage scende tra due build, c'è codice nuovo che nessun test tocca.

E adesso? Idee (e dove serve una mano) 🛠️

- **Basic-block coverage**: uprobe sui salti, via DWARF
- **Diff-coverage**: "il commit tocca X e Y: i test le hanno eseguite?"
- **Call graph / flamegraph** con `bpf_get_stack()`
- Export **LCOV / Cobertura** (GitLab, Codecov, Sonar)
- Supporto **RISC-V**

Contribuire: build, test, prova su un binario tuo (meglio in una VM Tumbleweed di scorta!) e guarda le issue **good first issue**.

> *Remember: fun is the first thing (fun - coverage).*

Grazie per l'attenzione!

Andrea Manzini

https://ilmanzo.github.io

Domande e (forse) risposte



andrea.manzini@suse.com

Risorse

- 📦 Repo: https://github.com/ilmanzo/BinaryCoverage
- 📝 Serie di blog "How much code are you testing?": https://ilmanzo.github.io
  - Go `-cover` · GDB+Valgrind · Intel PIN · **eBPF** · `dlopen()` JIT
- 🔭 Alternativa eBPF: xcover (FOSDEM 2026), di @maxgio92
- 🧪 openQA: https://openqa.opensuse.org
- 🐝 eBPF: https://ebpf.io