Observability with eBPF

REDOCS'25 - GDR Sécurité Informatique

Paul Houssel[1,2] Kahina Lazri[1] Tristan D'audibert[1]

[1]Orange Research, Châtillon
[2]Institut Polytechnique de Paris, SAMOVAR, Télécom SudParis, Palaiseau

13^{th} of October 2025

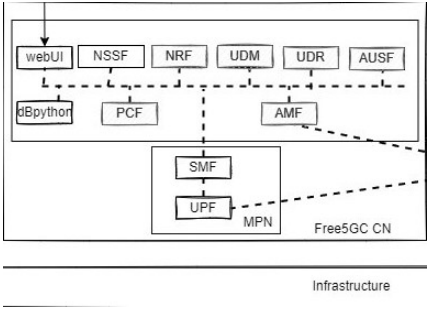
Context

5G Network Deployment at Orange

- 5G relies on several complementary virtual network functions:
  - AMF (Access and Mobility Management Function) : Authentication and handover between antennas
  - UPF (User Plane Function) : Internet traffic gateway

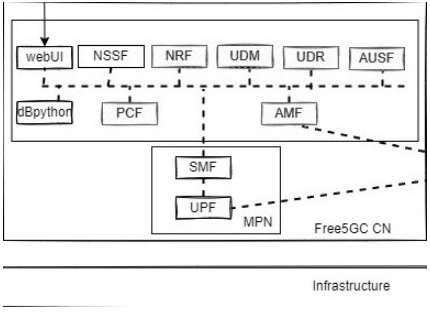
5G Network Deployment at Orange

- 5G relies on several complementary virtual network functions:
  - AMF (Access and Mobility Management Function) : Authentication and handover between antennas
  - UPF (User Plane Function) : Internet traffic gateway



5G Network Deployment at Orange

- 5G relies on several complementary virtual network functions:
  - AMF (Access and Mobility Management Function) : Authentication and handover between antennas
  - UPF (User Plane Function) : Internet traffic gateway



Orange deploys these 5G functions provided by equipment vendors (Ericsson, Nokia, etc.), who deliver precompiled binaries.

Observability and enforcement of 5G Networks

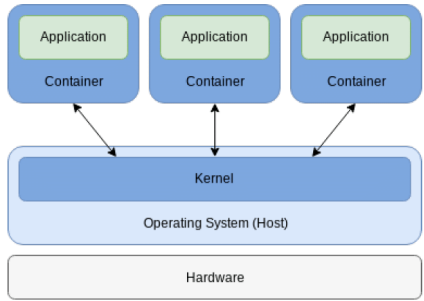
For its 5G network functions, Orange wants to:

- Monitor network function execution in real time
- Apply appropriate security policies

Observability and enforcement of 5G Networks

For its 5G network functions, Orange wants to:

- Monitor network function execution in real time
- Apply appropriate security policies



Kernel Telemetry Options

Location	Approach	efficiency	visibility	robustness	portability	safety	Example
kernel	integrated systems	●	●	●	❍	●	ftrace [1]
	kernel module	●	●	●	◗	❍	SE Linux [2]
	eBPF	●	●	●	◗	●	Tetragon [3]

Kernel Telemetry Options

Location	Approach	efficiency	visibility	robustness	portability	safety	Example
kernel	integrated systems	●	●	●	❍	●	ftrace [1]
	kernel module	●	●	●	◗	❍	SE Linux [2]
	eBPF	●	●	●	◗	●	Tetragon [3]

Kernel Telemetry Options

Location	Approach	efficiency	visibility	robustness	portability	safety	Example
kernel	integrated systems	●	●	●	❍	●	ftrace [1]
	kernel module	●	●	●	◗	❍	SE Linux [2]
	eBPF	●	●	●	◗	●	Tetragon [3]

Kernel Telemetry Options

Location	Approach	efficiency	visibility	robustness	portability	safety	Example
kernel	integrated systems	●	●	●	❍	●	ftrace [1]
	kernel module	●	●	●	◗	❍	SE Linux [2]
	eBPF	●	●	●	◗	●	Tetragon [3]

eBPF (extended Berkeley Packet Filter)

eBPF makes it possible to extend Linux kernel capabilities safely [4]:

- Deployment of restricted programs directly into the kernel

eBPF (extended Berkeley Packet Filter)

eBPF makes it possible to extend Linux kernel capabilities safely [4]:

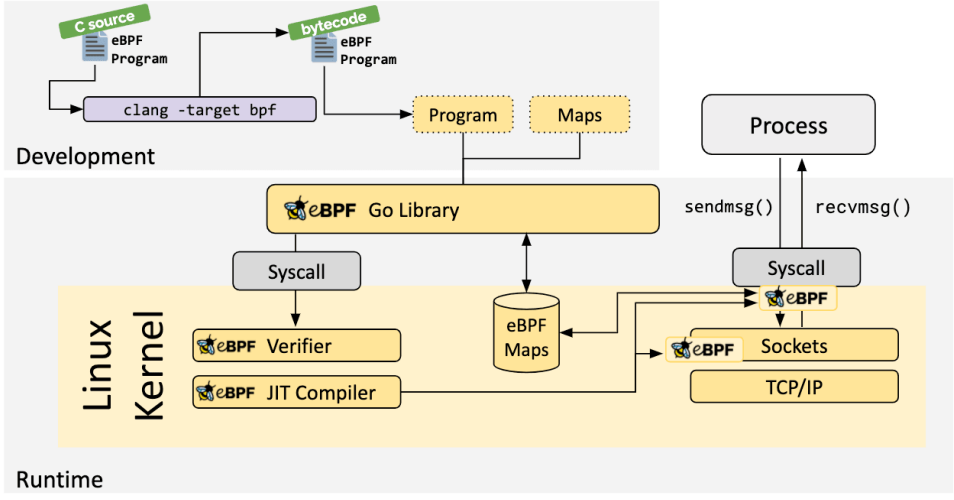
- Deployment of restricted programs directly into the kernel
- A formal verifier guarantees:
  - termination (no unbounded loops)
  - no memory leaks or arbitrary access
  - no deadlocks

eBPF

Types of eBPF Programs

Program type	Attach type	Granularity			
		system calls	kernel function	user-space function	Access Control
LSM	`LSM_MAC`	❍	❍	❍	●
tracepoint	`tracepoint`	●	◗	❍	❍
kprobe	`kprobe/kretprobe` `uprobe/uretprobe`	●	●	●	●
tracing	`fentry/fexit`	●	●	❍	●

eBPF [5]


C source
eBPF Program
bytecode
eBPF Program
clang -target bpf
Program
Maps
Development
Process
eBPF Go Library
sendmsg()
recvmsg()
Syscall
Syscall
eBPF
eBPF Maps
eBPF Verifier
eBPF JIT Compiler
eBPF Sockets
TCP/IP
Linux Kernel
Runtime

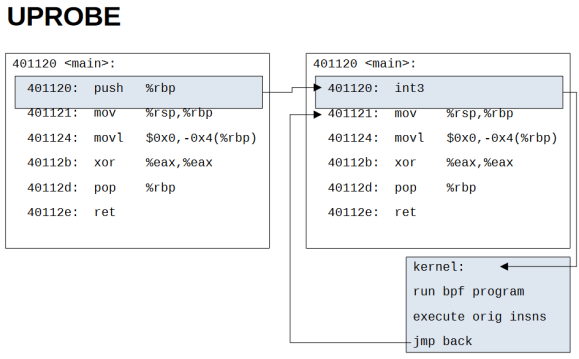
User-Space Observability with eBPF

With `uprobes`, eBPF can trace functions in user space:

```
SEC("uprobe//usr/bin/myapp:main.CreateUser")
int BPF_KPROBE(create_user)
{
    bpf_printk("user created !");
    return 0;
}
```

User-Space Observability with eBPF

1. The kernel replaces the target instruction with `INT3`
2. A CPU interrupt triggers execution of the eBPF program
3. Registers are copied and exposed to the eBPF context
4. The original instruction, which was removed, is then replayed



Observability of Compiled 5G Functions

- identify symbols from object files $\rightarrow$ `nm` tool
- Reconstruct data structures used $\rightarrow$ BTF
- Correctly interpret registers and memory pointers $\rightarrow$ BTF

BPF Type Format (BTF)

BTF describes type information:

- Produced by Pahole from DWARF data

BPF Type Format (BTF)

BTF describes type information:

- Produced by Pahole from DWARF data
- Allows linking eBPF code to the target kernel without requiring recompilation, and makes it portable across kernel versions

BPF Type Format (BTF)

BTF describes type information:

- Produced by Pahole from DWARF data
- Allows linking eBPF code to the target kernel without requiring recompilation, and makes it portable across kernel versions

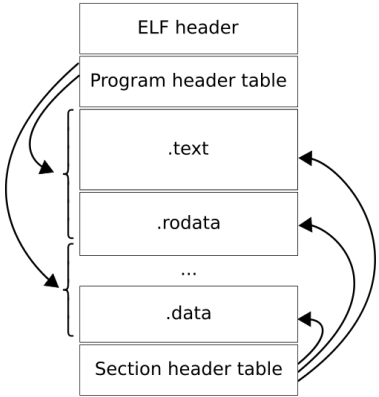
```
[1] INT 'int' size=4 bits_offset=0
  nr_bits=32 encoding=SIGNED
[2] STRUCT 'foo' size=8 vlen=2
  'f1' type_id=1 bits_offset=0
  'f2' type_id=1 bits_offset=32
```

DWARF (Debug With Arbitrary Record Format)

DWARF: the debugging format for compiled programs

- Describes the compiled program's structure for debugging (maps machine code to source code)
- Used by debuggers such as GDB or LLDB
- Included in ELF files (`.debug_*` sections)

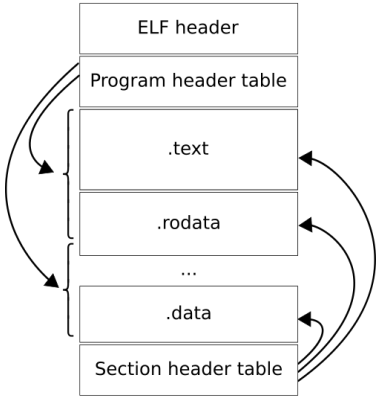
ELF (Executable Linkable Format)



Standard file format used for executables:

- `.text` : program executable code
- `.data` : initialized variables

ELF (Executable Linkable Format)



Standard file format used for executables:

- `.text` : program executable code
- `.data` : initialized variables
- `.debug_info` : types, variables, functions
- `.debug_line` : source code lines $\leftrightarrow$ address mapping
- ...

DWARF (Debug With Arbitrary Record Format)

Information defined by DWARF:

- Source–binary mapping: source line $\leftrightarrow$ machine address
- Variables and types: names, types, memory locations
- Functions and calls: signatures, return values, call stack
- Variable locations: register or stack

BTF (BPF File Format)

Format describing types and data structures used by the kernel and eBPF programs.

Utility for eBPF

- Allows correct parsing of CPU registers:
  - kernel internal data structures
  - user program data structures
- Platform-agnostic $\rightarrow$ no recompilation needed

Pahole - Usage example (C)

```c
struct Example {
  int id;
  char name[32];
};

int maFonctionTest(struct Example *e) {
  printf("ID: %d, Name: %s\n", e->id, e->name);
  return 0;
}
int main(void) {
  struct Example e = {1, "BTF Example"};
  maFonctionTest(&e);
  return 0;
}
```

Pahole - Usage example (C)

```
$ ./simple_program_c
ID: 1, Name: BTF Example
```

Pahole - Usage example (C)

```
$ ./simple_program_c
ID: 1, Name: BTF Example

$ nm ./simple_program_c | grep maFonctionTest
0000000000001180 T maFonctionTest
```

Pahole - Usage example (C)

```
$ ./simple_program_c
ID: 1, Name: BTF Example

$ nm ./simple_program_c | grep maFonctionTest
0000000000001180 T maFonctionTest

$ pahole --btf_encode_detached=./simple_program_c.btf ./simple_program_c
```

Pahole - Usage example (C)

```
$ ./simple_program_c
ID: 1, Name: BTF Example

$ nm ./simple_program_c | grep maFonctionTest
0000000000001180 T maFonctionTest

$ pahole --btf_encode_detached=./simple_program_c.btf ./simple_program_c
```

```
$ bpftool btf dump file simple_program_c.btf | grep -A 2 Example
[11] STRUCT 'Example' size=36 vlen=2
        'id' type_id=7 bits_offset=0
        'name' type_id=12 bits_offset=32
```

Pahole - Usage example (Go)

```go
type Example struct {
    ID   int
    Name string
}

func maFonctionTest(e *Example) int {
    fmt.Printf("ID: %d, Name: %s\n", e.ID, e.Name)
    return 0
}

func main() {
    e := Example{
        ID:   1,
        Name: "BTF Example",
    }
    maFonctionTest(&e)
}
```

Pahole - Usage example (Go)

```
$ ./simple_program_go
ID: 1, Name: BTF Example
```

Pahole - Usage example (Go)

```
$ ./simple_program_go
ID: 1, Name: BTF Example
```

```
$ nm ./simple_program_go | grep maFonctionTest
00000000004b0940 T main.maFonctionTest
```

Pahole - Usage example (Go)

```
$ ./simple_program_go
ID: 1, Name: BTF Example
$ nm ./simple_program_go | grep maFonctionTest
00000000004b0940 T main.maFonctionTest
$ pahole --btf_encode_detached=./simple_program_go.btf ./simple_program_go
```

Pahole - Usage example (Go)

```
$ ./simple_program_go
ID: 1, Name: BTF Example
$ nm ./simple_program_go | grep maFonctionTest
00000000004b0940 T main.maFonctionTest
$ pahole --btf_encode_detached=./simple_program_go.btf ./simple_program_go
```

```
$ bpftool btf dump file simple_program_go.btf | grep -A 2 Example
[1857] STRUCT 'main.Example' size=24 vlen=2
        'ID' type_id=76 bits_offset=0
        'Name' type_id=90 bits_offset=64
```

Problem Statement

Limitations of Pahole for the Go Language

Limitations of Pahole for the Go Language

- For simple Go programs: BTF generation works

Limitations of Pahole for the Go Language

- For simple Go programs: BTF generation works
- Pahole systematically fails on complex Go binaries $\rightarrow$ inconsistencies between DWARF and libbpf

Limitations of Pahole for the Go Language

- For simple Go programs: BTF generation works
- Pahole systematically fails on complex Go binaries $\rightarrow$ inconsistencies between DWARF and libbpf
- Go is widely used in the telecommunications industry

Limitations of Pahole for the Go Language

```
$ make kubeadm DBG=1
$ _output/bin/kubeadm


      KUBEADM
      Easily bootstrap a secure Kubernetes cluster

      Please give us feedback at:
      https://github.com/kubernetes/kubeadm/issues

...
```

Limitations of Pahole for the Go Language

```
$ make kubeadm DBG=1
$ _output/bin/kubeadm


      KUBEADM
      Easily bootstrap a secure Kubernetes cluster

      Please give us feedback at:
      https://github.com/kubernetes/kubeadm/issues

...
```

```
$ pahole --btf_encode_detached=./kubeadm.btf _output/bin/kubeadm
btf_encoder__encode: btf__dedup failed!
Failed to encode BTF
```

Limitations of Pahole for the Go Language

```
$ make kubectl DBG=1
$ _output/bin/kubectl
kubectl controls the Kubernetes cluster manager.

 Find more information at: https://kubernetes.io/docs/reference/kubectl/
...
```

Limitations of Pahole for the Go Language

```
$ make kubectl DBG=1
$ _output/bin/kubectl
kubectl controls the Kubernetes cluster manager.

 Find more information at: https://kubernetes.io/docs/reference/kubectl/
...
```

```
$ pahole --btf_encode_detached=./kubectl.btf _output/bin/kubectl
btf_encoder__encode: btf__dedup failed!
Failed to encode BTF
```

Objectives

Work Plan

Investigate the structure of DWARF data generated by the Go compiler to understand the limitations of libbpf, which Pahole uses to generate BTF files.

Work Plan

Investigate the structure of DWARF data generated by the Go compiler to understand the limitations of libbpf, which Pahole uses to generate BTF files.

1. Identify differences in results by testing other Go compilers (such as `gccgo`)

Work Plan

Investigate the structure of DWARF data generated by the Go compiler to understand the limitations of libbpf, which Pahole uses to generate BTF files.

1. Identify differences in results by testing other Go compilers (such as `gccgo`)
2. Incrementally test increasingly complex Go programs to pinpoint incompatibility causes:
   - Understand how the Go compiler generates DWARF data
   - Identify blocking points within Pahole

Work Plan

Investigate the structure of DWARF data generated by the Go compiler to understand the limitations of libbpf, which Pahole uses to generate BTF files.

1. Identify differences in results by testing other Go compilers (such as `gccgo`)
2. Incrementally test increasingly complex Go programs to pinpoint incompatibility causes:
   - Understand how the Go compiler generates DWARF data
   - Identify blocking points within Pahole
3. (Adapt Pahole to ensure full compatibility with Go)

Technical Environment

1. Local installation (Linux ≥ 6.0)
2. Preconfigured virtual machine (VirtualBox)

All resources available at:
https://gitlabev.imtbs-tsp.eu/paul.houssel/redocs25-orange

Useful References

- BTF documentation: https://docs.kernel.org/bpf/btf.html
- DWARF V4 specification: https://dwarfstd.org/doc/DWARF4.pdf
- Introduction to the DWARF format: https://dwarfstd.org/doc/Debugging using DWARF-2012.pdf
- Problematic deduplication algorithm: https://nakryiko.com/posts/btf-dedup/
- Tools for parsing DWARF data:
  1. `objdump`
  2. `readelf`

Observability with eBPF

REDOCS'25 - GDR Sécurité Informatique

Paul Houssel[1,2] Kahina Lazri[1] Tristan D'audibert[1]

[1]Orange Research, Châtillon
[2]Institut Polytechnique de Paris, SAMOVAR, Télécom SudParis, Palaiseau

13^{th} of October 2025

References I

- “ftrace - Function Tracer — The Linux Kernel documentation,” 2008.
- S. Smalley, C. Vance, and W. Salamon, Implementing SELinux as a Linux security module.
  NAI Labs Report, 2001.
- “Tetragon - eBPF-based Security Observability and Runtime Enforcement,” 2022.
- B. Gbadamosi, L. Leonardi, T. Pulls, T. Høiland-Jørgensen, S. Ferlin-Reiter, S. Sorce, and A. Brunström, “The eBPF Runtime in the Linux Kernel,” Oct. 2024.
  arXiv:2410.00026 [cs].
- “What is ebpf? an introduction and deep dive into the ebpf technology.”