Pahole: Patching the 🐇 hole

Martin Horth, Ouail Derghal, Guilhem Jazeron,
Ludovic Paillat, Robin Theveniaut

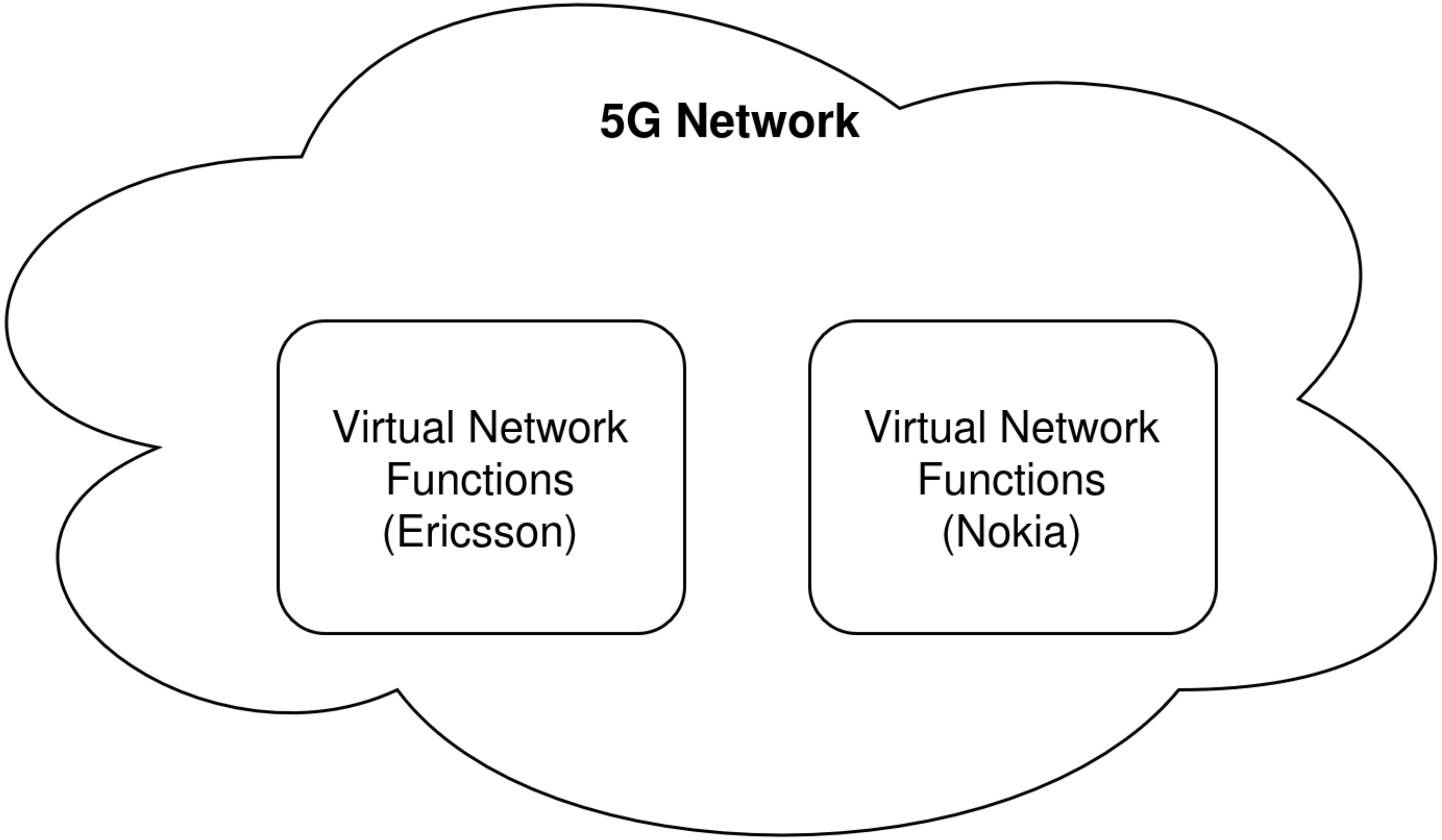
Supervised by: Paul Houssel, Kahina Lazri, Tristan d'Audibert

10 ANS REDOCS

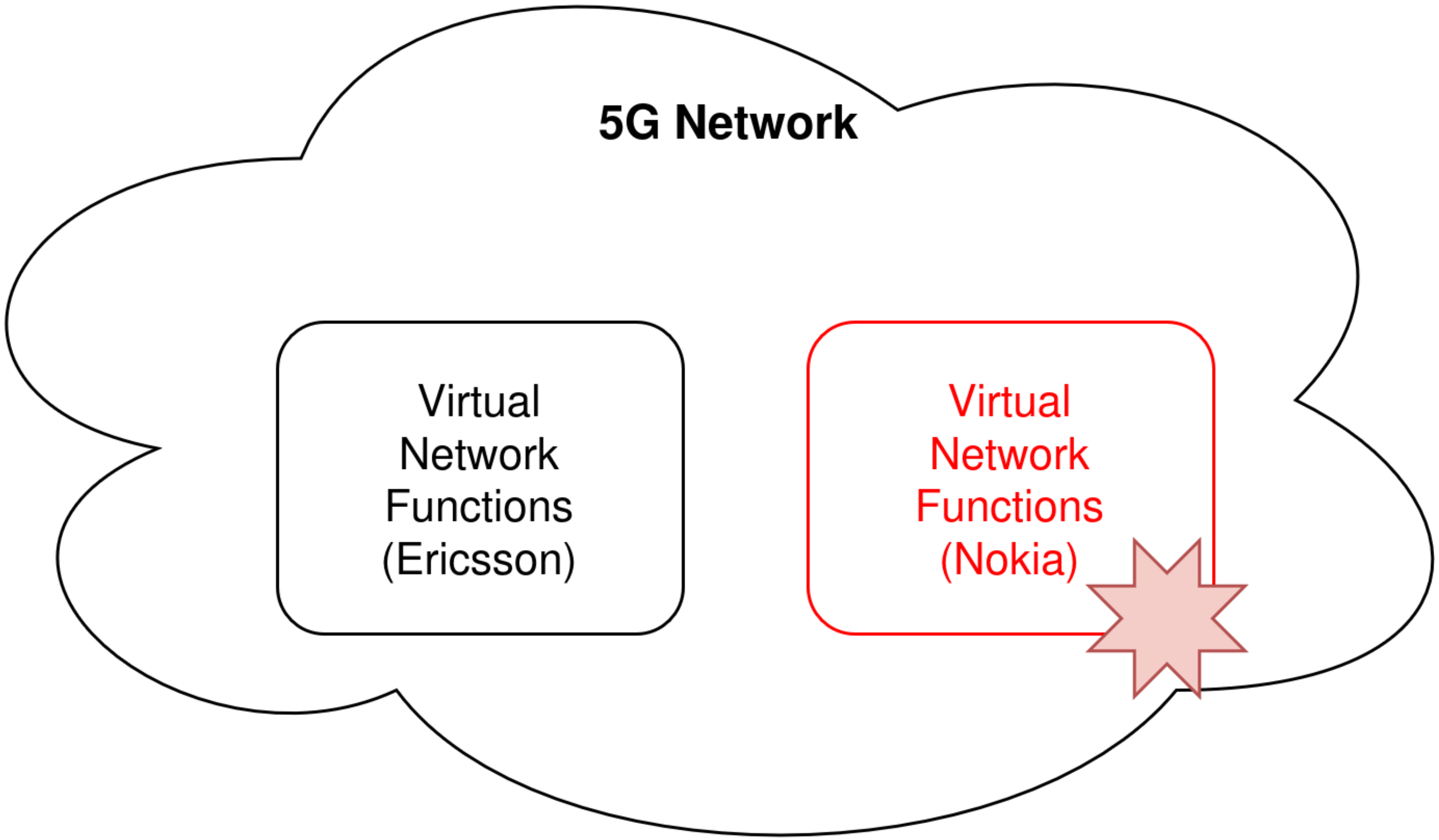
orange™

October 17th, 2025

Orange 5G Network


5G Network
Virtual Network Functions (Ericsson)
Virtual Network Functions (Nokia)

Failure in the 5G Network

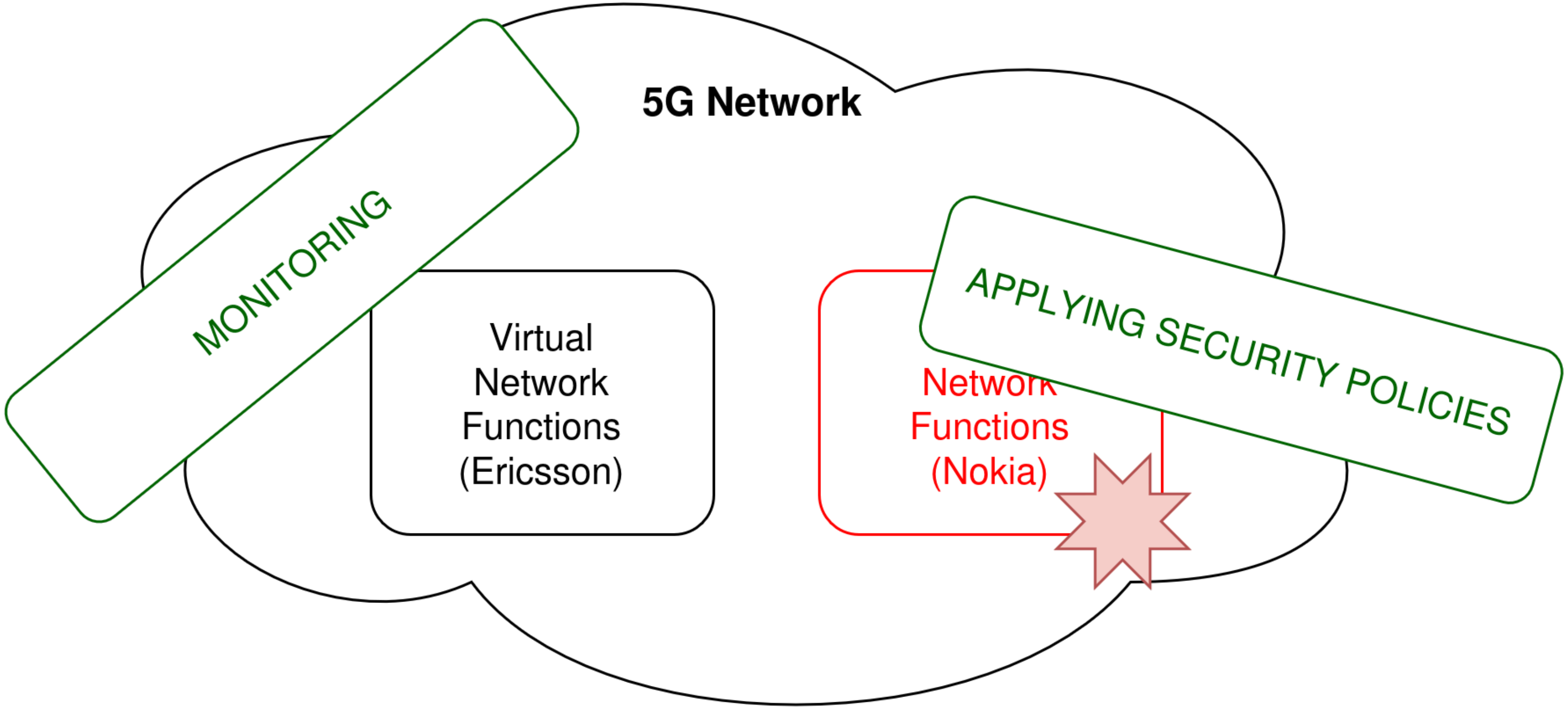


5G Network

Virtual Network Functions (Ericsson)

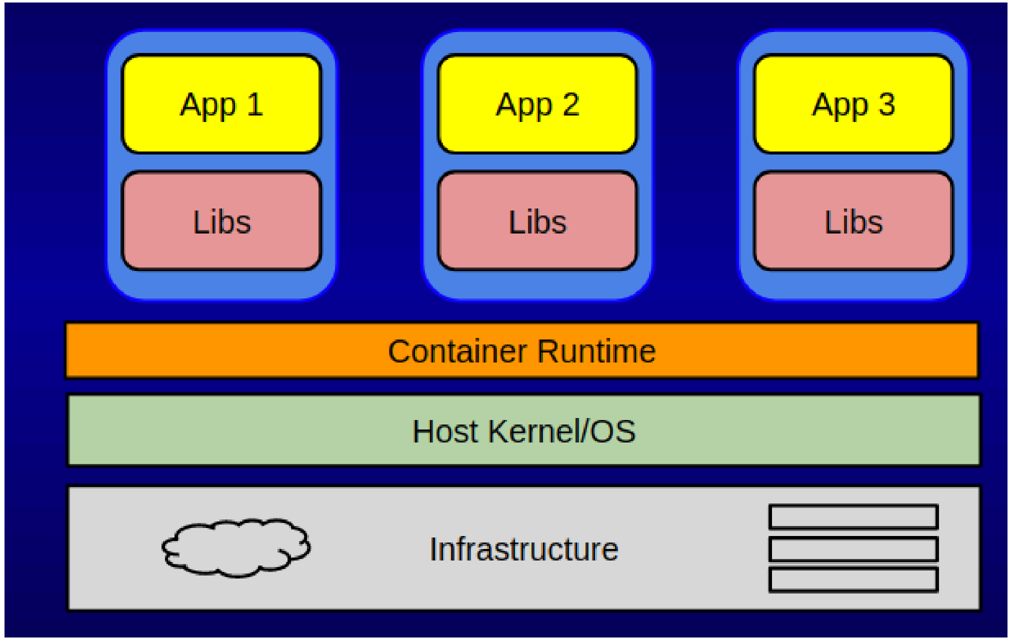
Virtual Network Functions (Nokia)

How we could avoid these threats


5G Network
MONITORING
Virtual Network Functions (Ericsson)
Network Functions (Nokia)
APPLYING SECURITY POLICIES

Observability from the Kernel

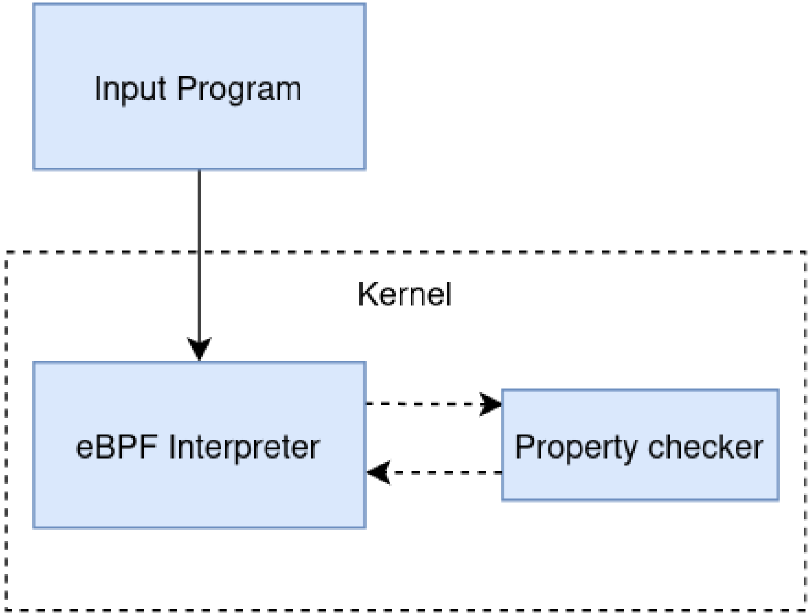
- Performance
- Visibility
- Robustness



eBPF

eBPF is a language that:

- Extends Linux kernel capabilities safely
- Deploys restricted programs directly into the kernel
- Ensures:
  - No unbounded loops
  - No memory leaks or arbitrary access
  - No deadlocks
  - ...

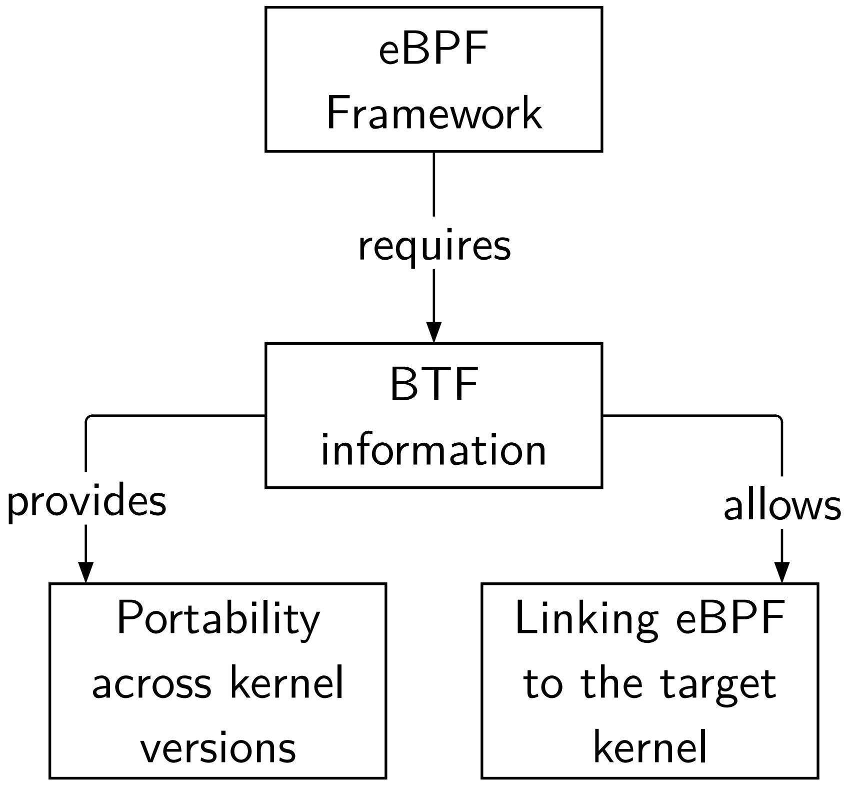


eBPF requirements

BTF (eBPF Type Format):

- Provides types information for BPF programs
- Needed for monitoring

⇒ How can we get it?



eBPF Framework

requires

BTF information

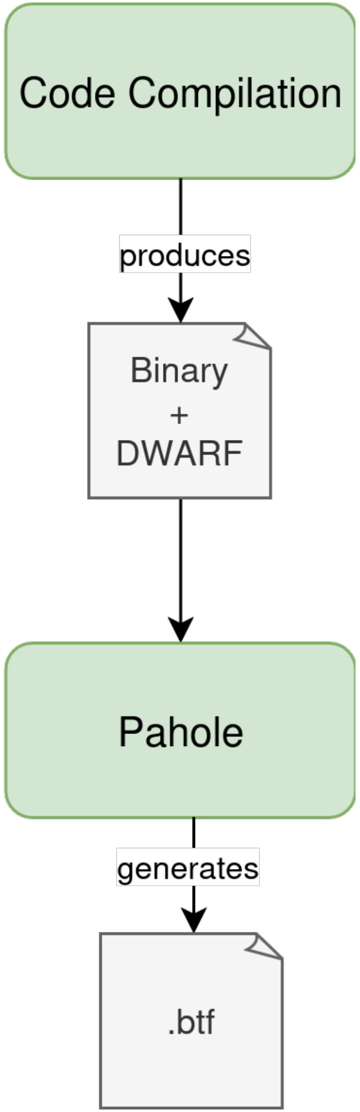
provides

allows

Portability across kernel versions

Linking eBPF to the target kernel

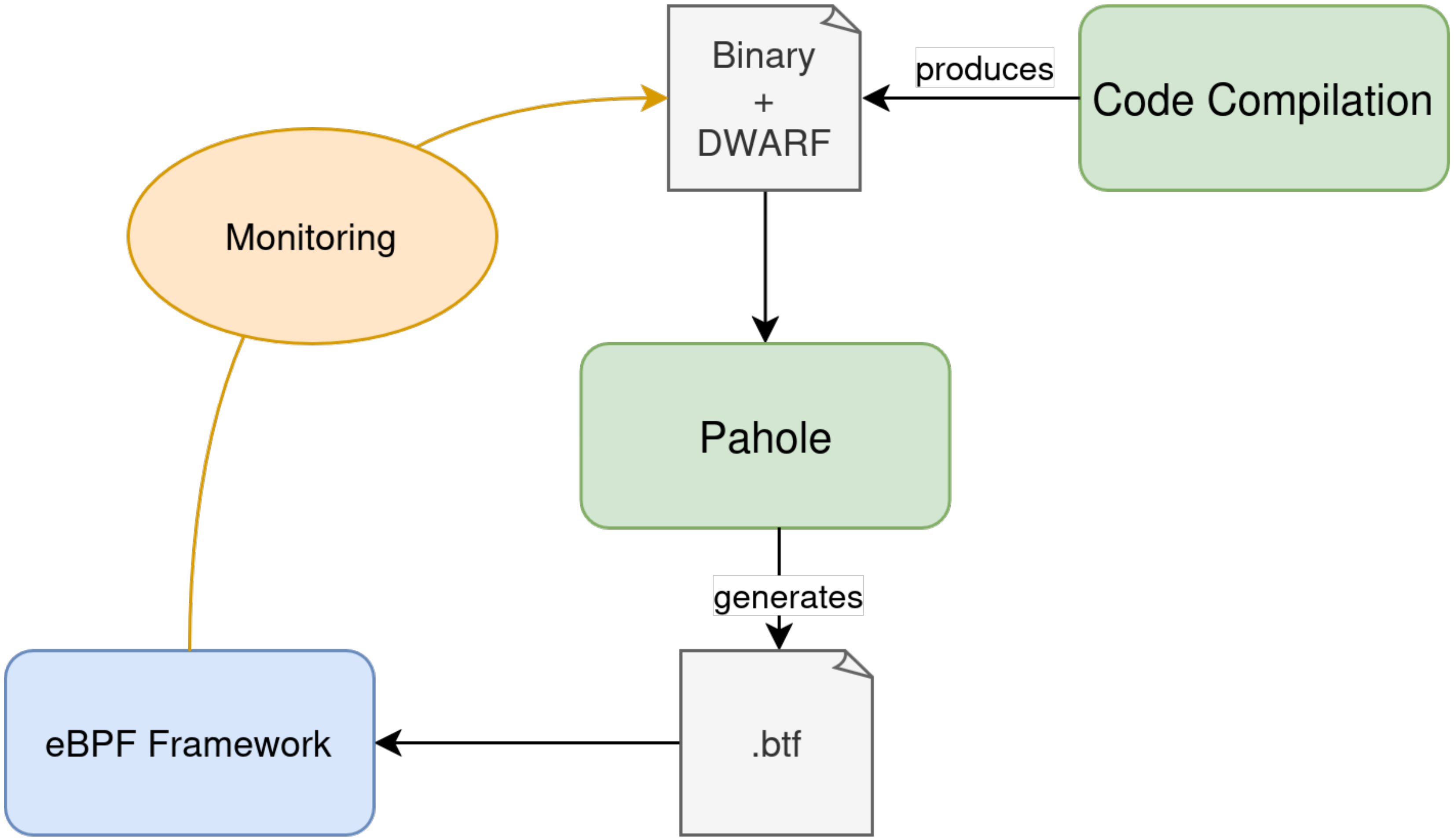
Using Pahole



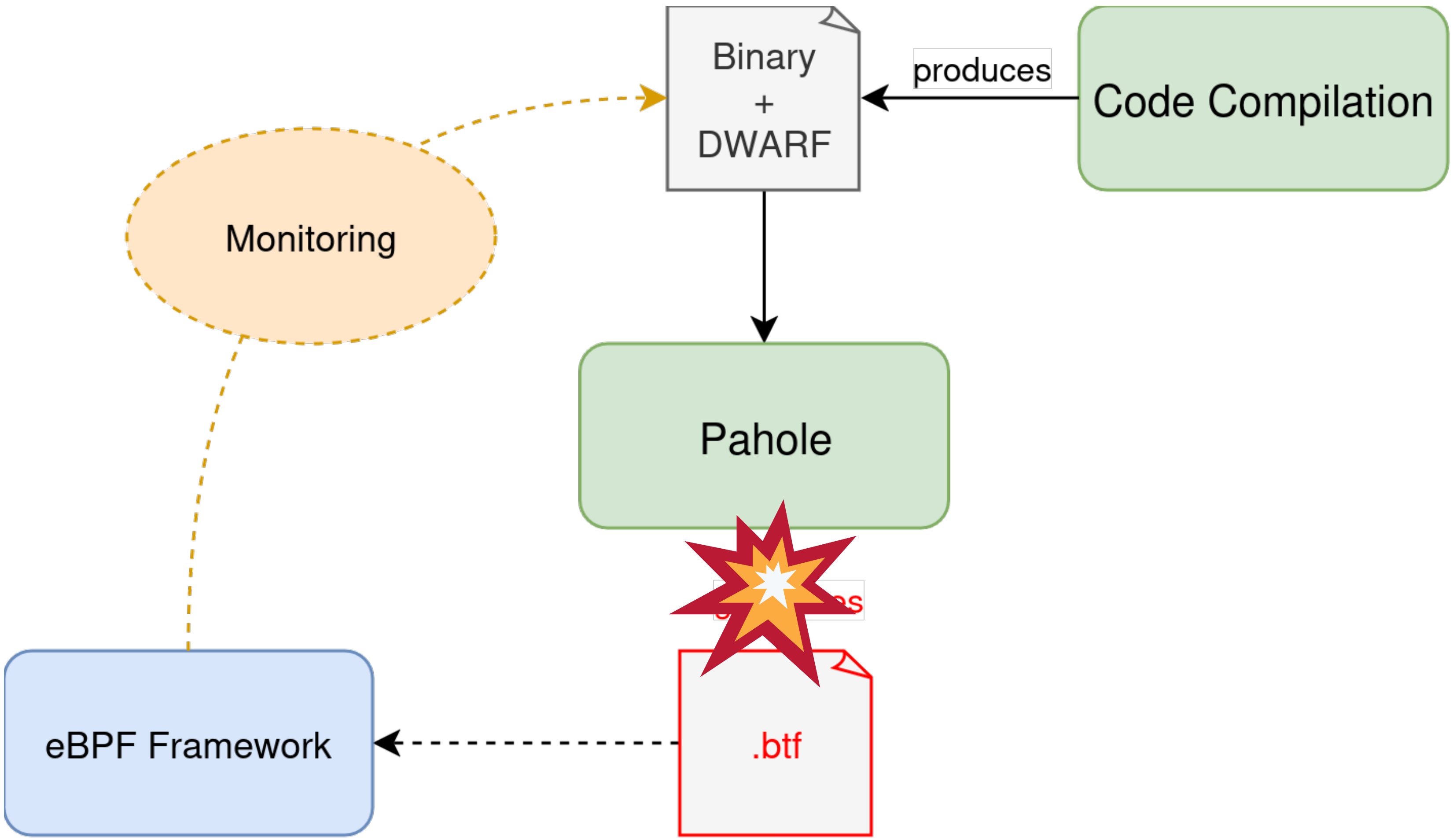
Pahole tool: translates DWARF information into BTF information

DWARF: debugging information produced during compilation

Pahole C projects BTF generation



Pahole failure on Go projects 💥



Contributions

1. Identification of the issue preventing Pahole to analyze Go programs
2. Unraveling the faulty algorithm and bug-fix
3. Evaluation of the fix

Identifying the issue 🔎

When does the issue happen ?

- Simple / toy programs in `C` and `Go`: OK ✅
- On more advanced Go programs (e.g. Kubernetes), depends:
  - With `gcc-go` compiler, OK ✅
    - Incompatible with newer go version
  - With `Go` standard compiler, nope ❌

Investigation on Kubernetes’ side

In `github.com/prometheus/common` dependency

go

expfmt/text_parse.go#L37

```go
// A stateFn is a function that represents a state in a state machine. By
// executing it, the state is progressed to the next state. The stateFn returns
// another stateFn, which represents the new state. The end state is represented
// by nil.
type stateFn func() stateFn
```

⇒ standard construction in Go for text parsers (based on state machines)

Minimal non-working code

go

```go
package main

type Foo func() Foo
```

⇒ Why is this a problem?

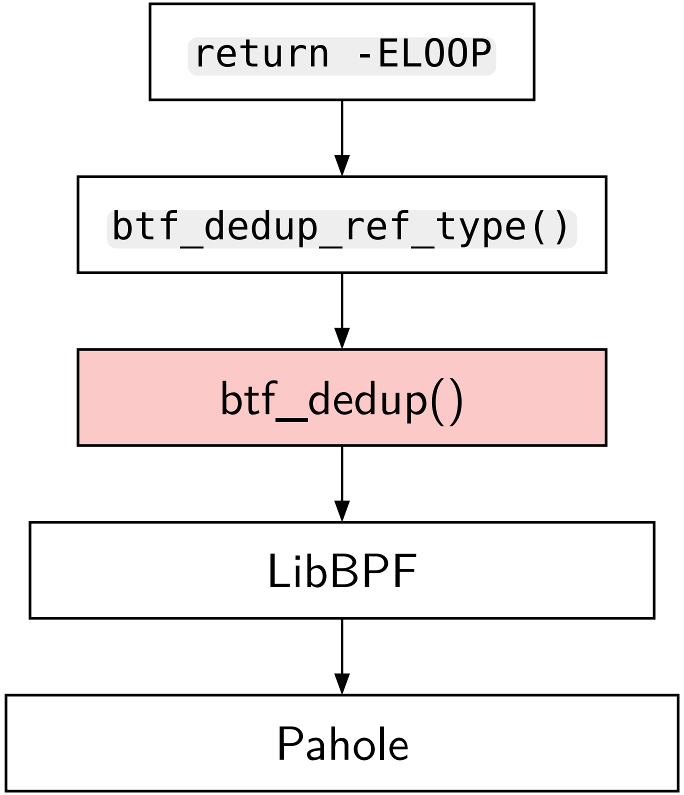
```go

func main() {
    bar()
}

func bar() Foo {
    return nil
}
```

Investigation on Pahole's side

- `-V` (verbose) option of Pahole gives `ELOOP` error
- Error comes from a dependency called `LibPBF`
- More precisely a function called `btf_dedup()`

⇒ Points out to the `dedup` algorithm of `LibBPF`

⇒ Let's investigate... 🔎



`return -ELOOP`

`btf_dedup_ref_type()`

btf_dedup()

LibBPF

Pahole

Resolution

Some details on BTF...

BTF stores information about symbols:

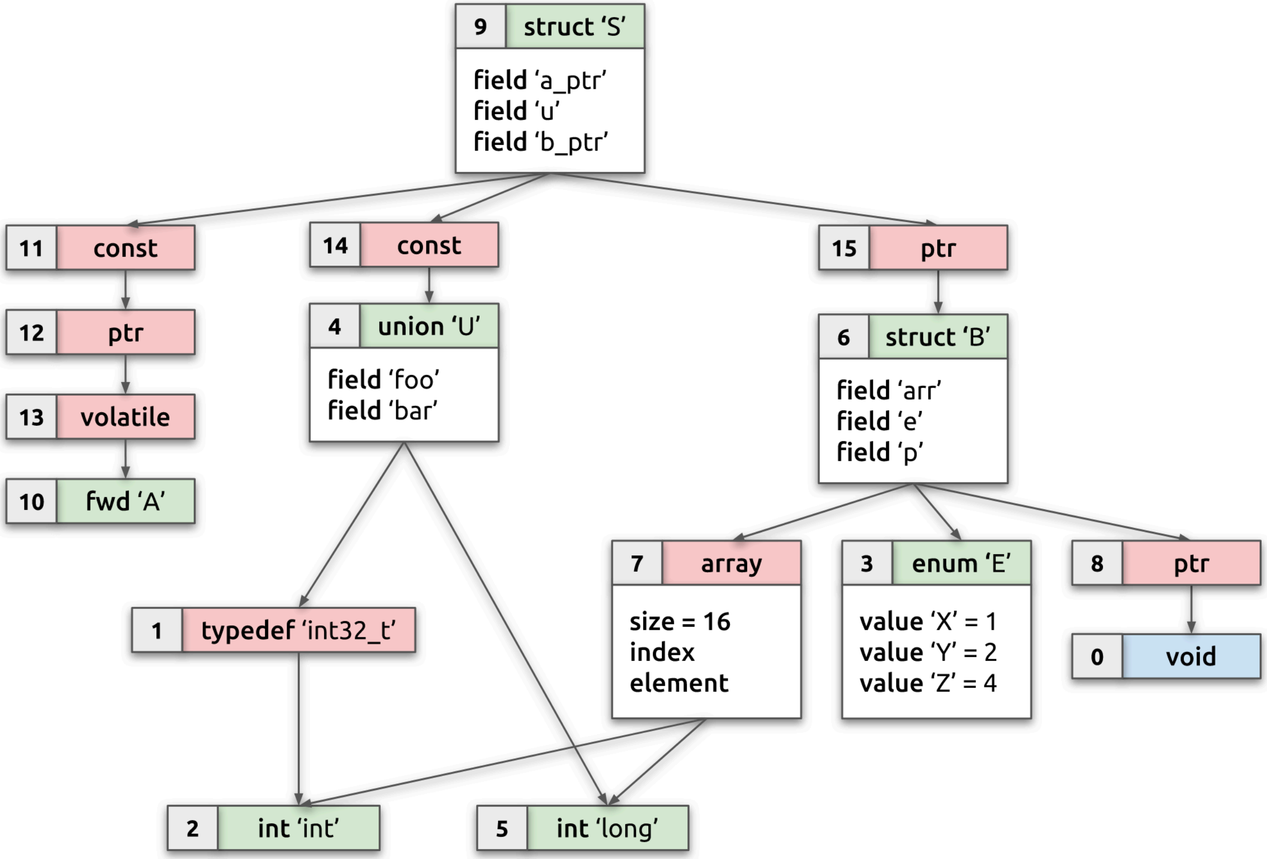
- A `KIND` variable
  - `BTF_KIND_TYPEDEF, BTF_KIND_INT, BTF_KIND_FUNC_PROTO`...
- A `name`
  - `int a` will have `name = a`
- A `typeid`: the unique identifier of a type
  - Allows to reference other types

Specific `KINDS` have specific variables:

`BTF_KIND_STRUCT` has `fields`, `BTF_FUNC` has `return_type`...

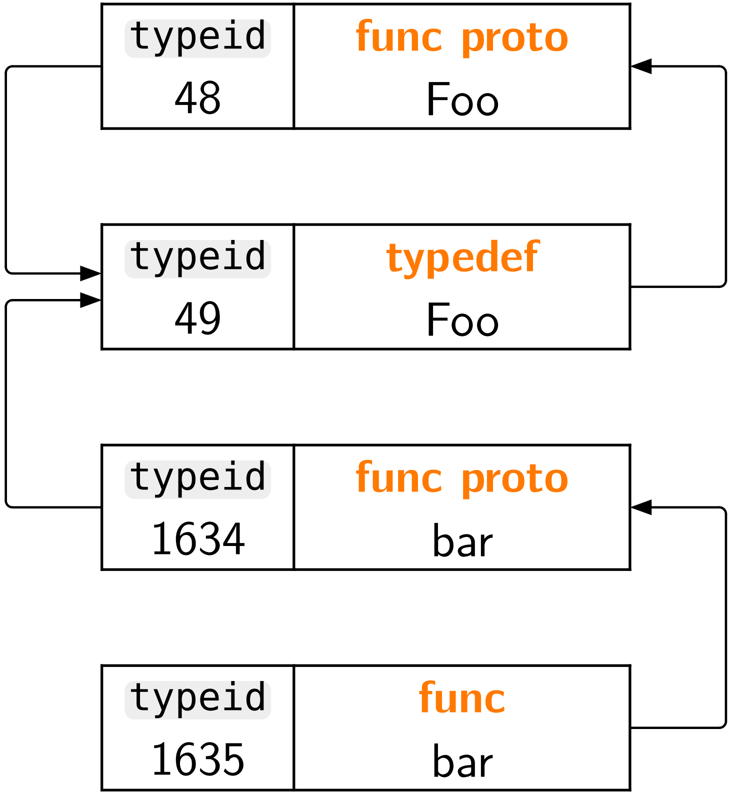
Example of BTF type structure

```
typedef int int32_t;

enum E {
    X = 1,
    Y = 2,
    Z = 4
};

union U {
    int32_t foo;
    long bar;
};

struct A;

struct B {
    long arr[16];
    enum E e;
    void* p;
};

struct S {
    volatile struct A* const a_ptr;
    const union U u;
    struct B* b_ptr;
};

int main() {
    struct S s;
    return 0;
}
```



Diving into our minimal non-working example

```go
package main

type Foo func() Foo

func main() {
    bar()
}

func bar() Foo {
    return nil
}
```

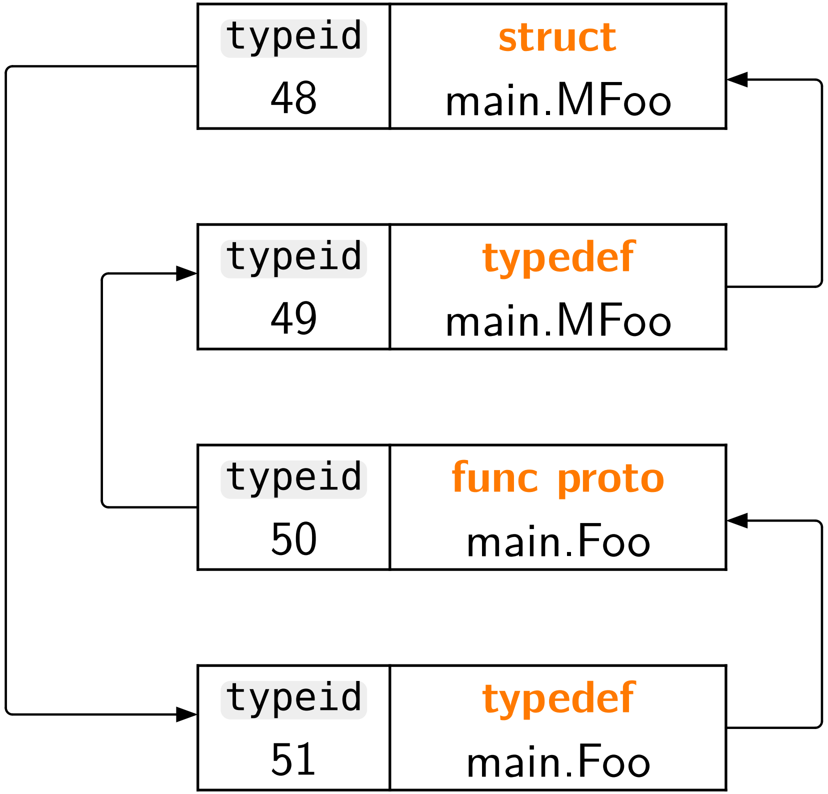


There is a loop in the BTF type graph!

⇒ Do all kind of loops make pahole fail?

But not all loops fail...

```go
package main

type MFoo struct {f Foo}
type Foo func() MFoo

func main() {
	bar()
}

func bar() MFoo {
	return MFoo {f: nil}
}
```



⇒ When loops are caused by **structs**, it doesn't fail!

Back to the code!

The error we identified:

bash

```bash
$ pahole --btf_encode_detached=simple_program_go.btf simple_program_go
simple_program_go
btf_encoder__encode: btf__dedup failed!
Failed to encode BTF
```

`btf__dedup` failed... 🤔

⇒ What does `dedup` even mean?

Digging again and again...

Keep digging (🪏 😓) a bit, we find some more info:

C

libbpf/src/btf.c

```c
/*
  * Deduplicate BTF types and strings.
  ...
  * Struct/union deduplication is the most critical part and algorithm for
  * deduplicating structs/unions is described in greater details in comments for
  * `btf_dedup_is_equiv` function.
  */
int btf__dedup(struct btf *btf, const struct btf_dedup_opts *opts);
```

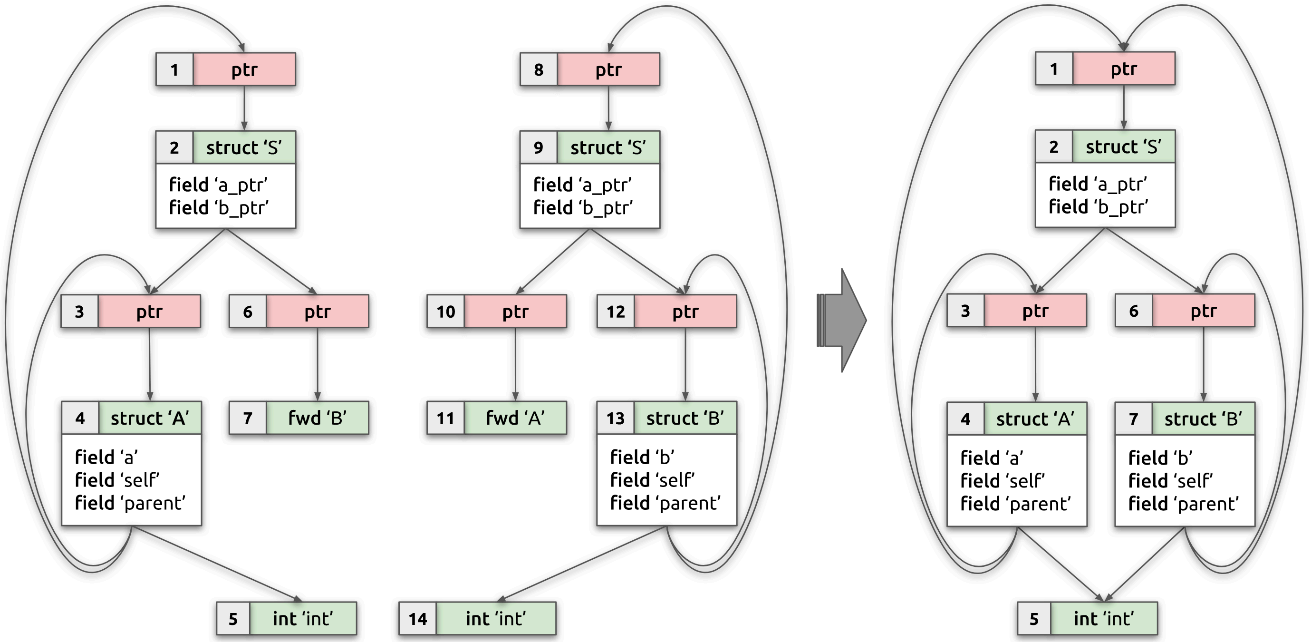
It means... **deduplication** 🤯

Unraveling deduplication mysteries

Deduplication sounds like the name of a dark magic chant 🧙 ⇒ let's make some sense out of it

Unraveling deduplication mysteries

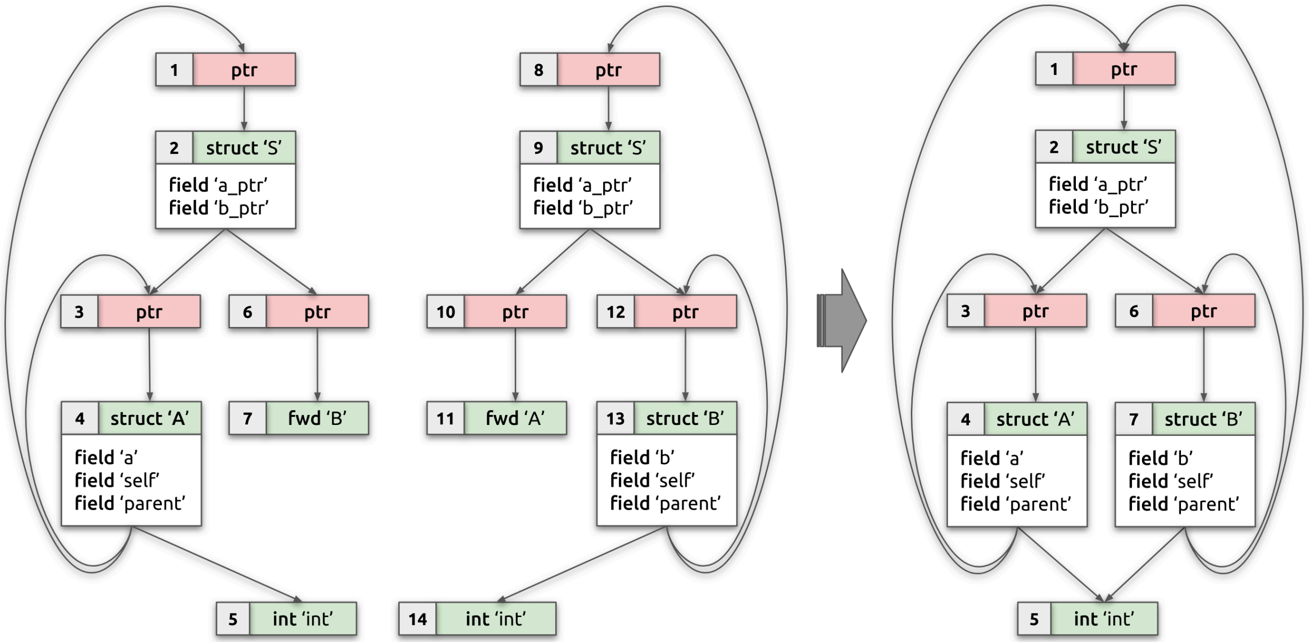
Deduplication sounds like the name of a dark magic chant 🧙 ⇒ let's make some sense out of it



Compression by deduplication

Unraveling deduplication mysteries

Deduplication sounds like the name of a dark magic chant 🧙 ⇒ let's make some sense out of it



Compression by deduplication

Same types, different `typeid`: duplicated

- **Huge space impact**
- In kernel land, size and speed are critical

⇒ Remove the duplicates...

🥁 **deduplicate**!

Importance of deduplication

⇒ The dedup algorithm is **critical**:

Type of debug info	Size
DWARF type descriptors	121.31MB
BTF type descriptors	101.7MB
Deduplicated BTF type descriptors	0.97MB

Example of deduplication on the Linux kernel,
from Andrii Nakryiko's Blog (designer of dedup algorithm)

Importance of deduplication

⇒ The dedup algorithm is **critical**:

Type of debug info	Size
DWARF type descriptors	121.31MB
BTF type descriptors	101.7MB
Deduplicated BTF type descriptors	0.97MB

Example of deduplication on the Linux kernel,
from Andrii Nakryiko's Blog (designer of dedup algorithm)

⇒ Cannot just remove the deduplication step to solve the bug

Quick overview of the dedup algorithm

Main idea of algorithm ⇒ establishing **equivalences**

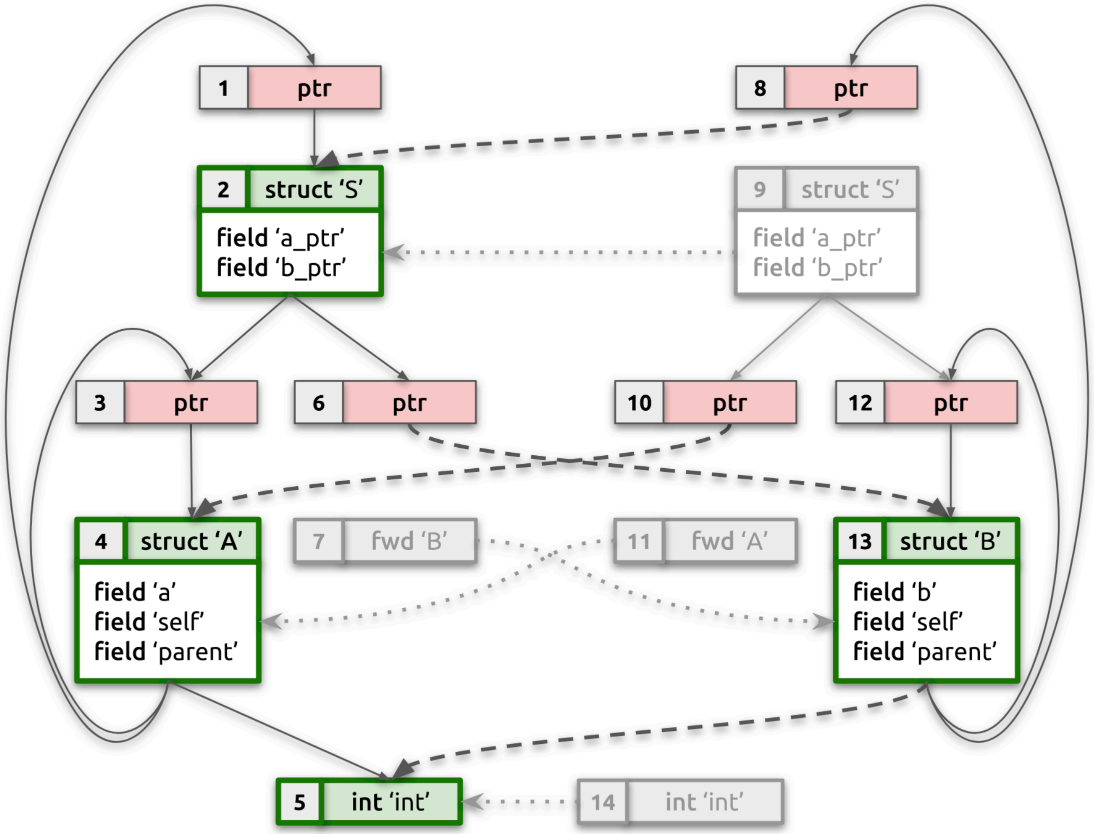
Quick overview of the dedup algorithm

Main idea of algorithm ⇒ establishing **equivalences**

1. Define **non-reference** and **reference types**:
   - reference types: make references to other types
     - `BTF_KIND_PTR, BTF_KIND_TYPEDEF`...
   - non-reference types:
     - The rest: `BTF_KIND_INT, BTF_KIND_STRING`...

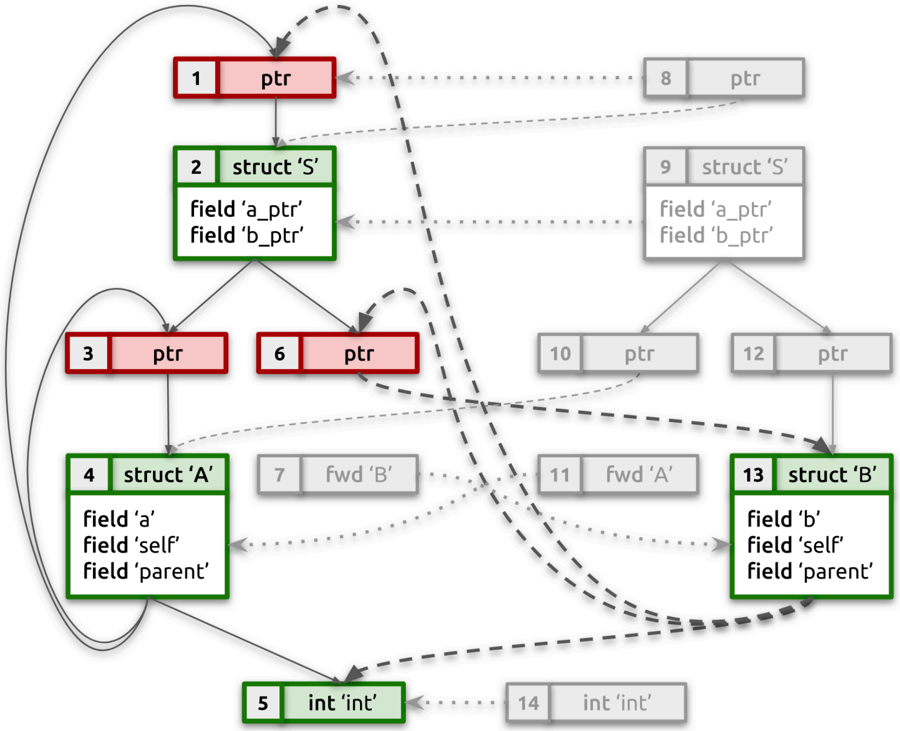
Quick overview of the dedup algorithm

1. Define **non-reference** and **reference** types.
2. Check equivalence of **non-reference** types:
   - for instance, all `int` symbols are equivalent...
   - equivalences between `typeid` are stored



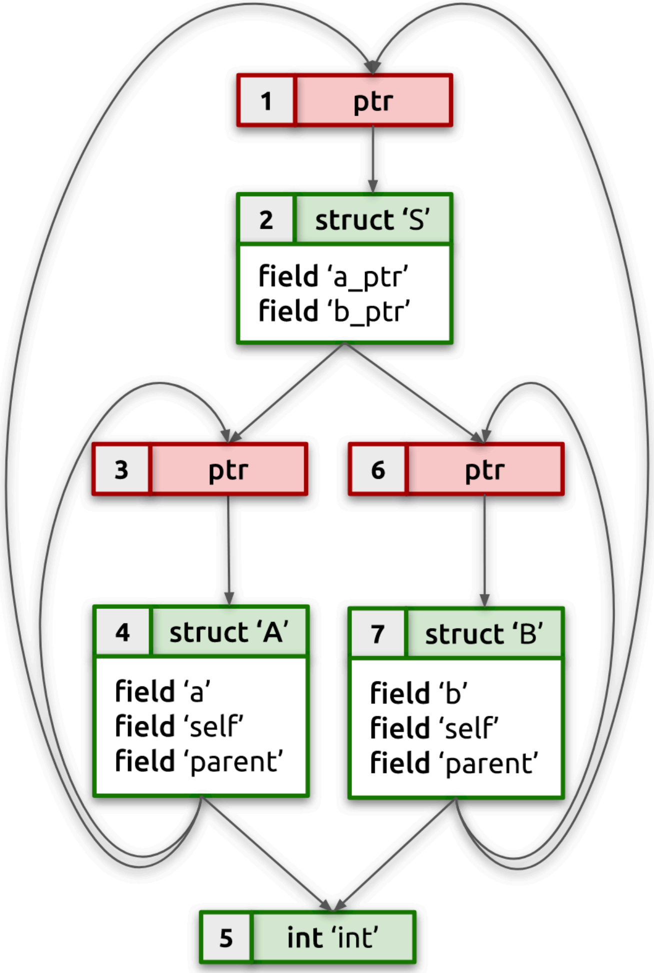
Quick overview of the dedup algorithm

1. Define **non-reference** and **reference** types.
2. Check equivalence of **non-reference** types.
3. Check equivalence of **reference** types:
   - check if they have same name
   - check if the (non-reference) types they refer to were detected as equivalent



Quick overview of the dedup algorithm

1. Define **non-reference** and **reference** types
2. Check equivalence of **non-reference** types
3. Check equivalence of **reference** types
4. Merge all equivalent type symbols and assign them new `typeid`



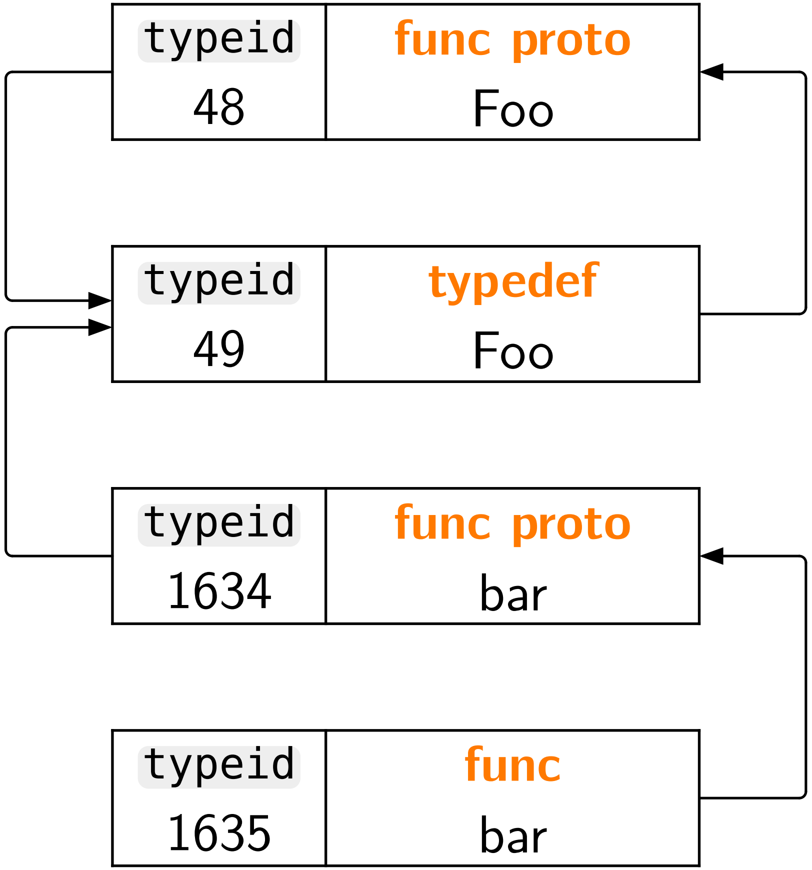
Importance of handling loops

If we look again at the example causing the bug...

The algorithm just cannot resolve this!

- `typeid = 49` needs to resolve equivalences of:
    - `typeid = 48` which needs to resolve equivalences of:
        - `typeid = 49` needs to resolve equivalences of:
            - `typeid = 48` which needs to resolve equivalences of...

⇒ You get it: 🌀 🌀 🌀 🌀 🌀



typeid	
48	func proto Foo
49	typedef Foo
1634	func proto bar
1635	func bar

But it sometimes works...

Remember:

- The **struct** case did not cause a problem... Even if there is a loop

c

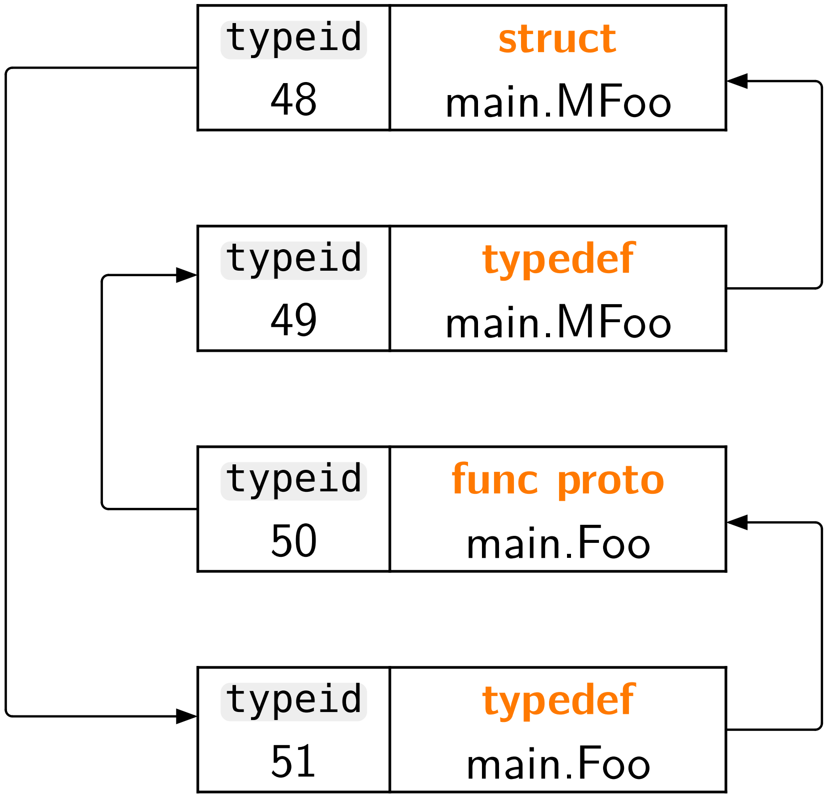
```c
libbpf/src/btf.c:btf_dedup_struct_type
eq = btf_dedup_is_equiv(d, type_id, cand_id);
```

- In the function `btf_dedup_struct_type`, there is `btf_dedup_is_equiv`

c

```c
Documentation of btf_dedup_is_equiv
// Check equivalence of BTF type graph formed
by candidate struct/union
```

- **struct** appears again... 🔎



The maintainers left hints

C

libbpf/src/btf.c:btf_dedup_ref_type

```c
// Recursion will always terminate at either primitive or
// struct/union type, at which point we can "unwind" chain of reference types
// one by one. There is no danger of encountering cycles because in C type
// system the only way to form type cycle is through struct/union
```

⇒ Look closely... 🔎

The maintainers left hints

C

libbpf/src/btf.c:btf_dedup_ref_type

```c
// Recursion will always terminate at either primitive or
// struct/union type, at which point we can "unwind" chain of reference types
// one by one. There is no danger of encountering cycles because in C type
// system the only way to form type cycle is through struct/union
```

⇒ Look closely... 🔎

The maintainers left hints

C

libbpf/src/btf.c:btf_dedup_ref_type

```c
// Recursion will always terminate at either primitive or
// struct/union type, at which point we can "unwind" chain of reference types
// one by one. There is no danger of encountering cycles because in C type
// system the only way to form type cycle is through struct/union
```

⇒ Look closely... 🔎

⚠ They were right... But we are not in C world! ⚠

⇒ Explains why binaries from `gccgo` did not cause issues: they get compiled like `C` binaries

And finally, the patch 🩹

- After all this hard work ⚒️...
- We got to the heart of the issue ⇒ easy to solve our problem

And finally, the patch 🩹

- After all this hard work ⚒️...
- We got to the heart of the issue ⇒ easy to solve our problem

⇒ Handle the case of `BTF_KIND_TYPEDEF` separately, mimicking how `BTF_KIND_STRUCT` is already handled

C

libbpf/src/btf.c

```c
static int btf_dedup_typedef_type(struct btf_dedup *d, __u32 type_id)
...
    // Reuse the recursive equivalence checking used for structs
    eq = btf_dedup_is_equiv(d, type_id, cand_id);
...
```

🎉 And voilà! 🎉

Evaluation

1. Soundness ✅
2. Performance overhead 🔥

Soundness of Deduplication patch

c

a.c

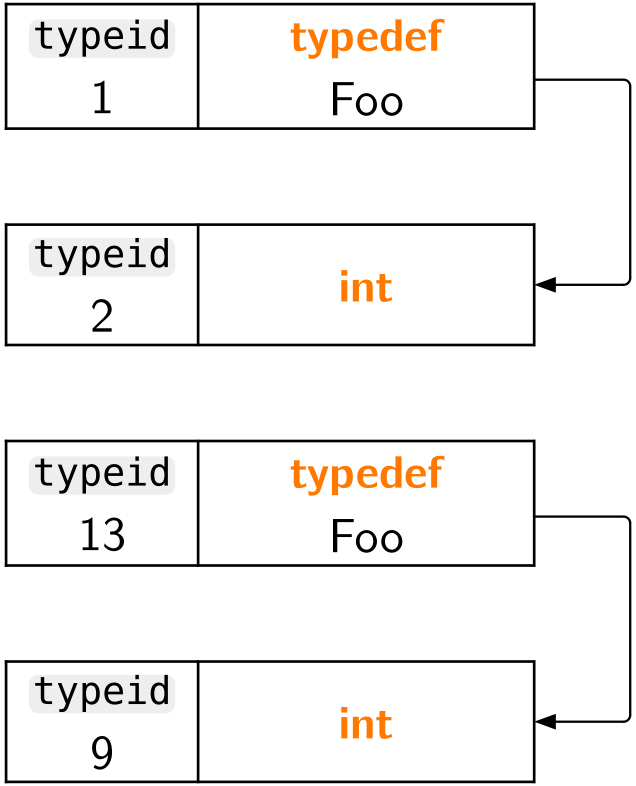
```c
typedef int foo;
foo add(foo a, foo b);

int main() {
    foo a = 5;
    foo b = 10;
    printf("%d", add(a,b));
    return 0;
}
```

c

b.c

```c
typedef int foo;

foo add(foo a, foo b){
    return a+b;
}
```

Without deduplication we get:



Soundness of Deduplication patch

c

a.c

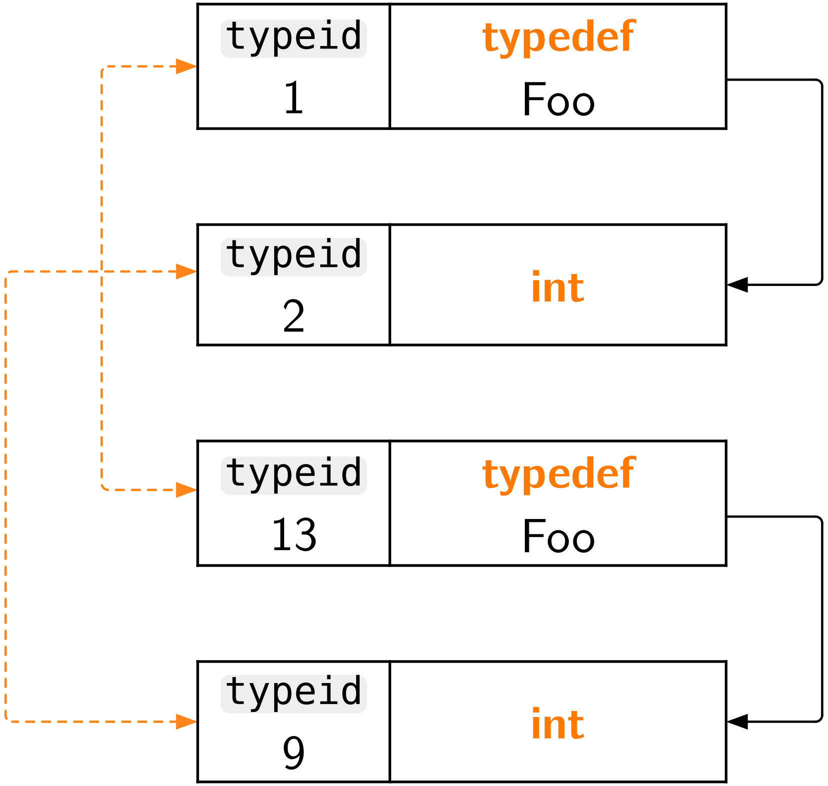
```c
typedef int foo;
foo add(foo a, foo b);

int main() {
    foo a = 5;
    foo b = 10;
    printf("%d", add(a,b));
    return 0;
}
```

c

b.c

```c
typedef int foo;

foo add(foo a, foo b){
    return a+b;
}
```

Without deduplication we get:



typeid 1 | typedef Foo

typeid 2 | int

typeid 13 | typedef Foo

typeid 9 | int

Soundness of Deduplication patch

c

a.c

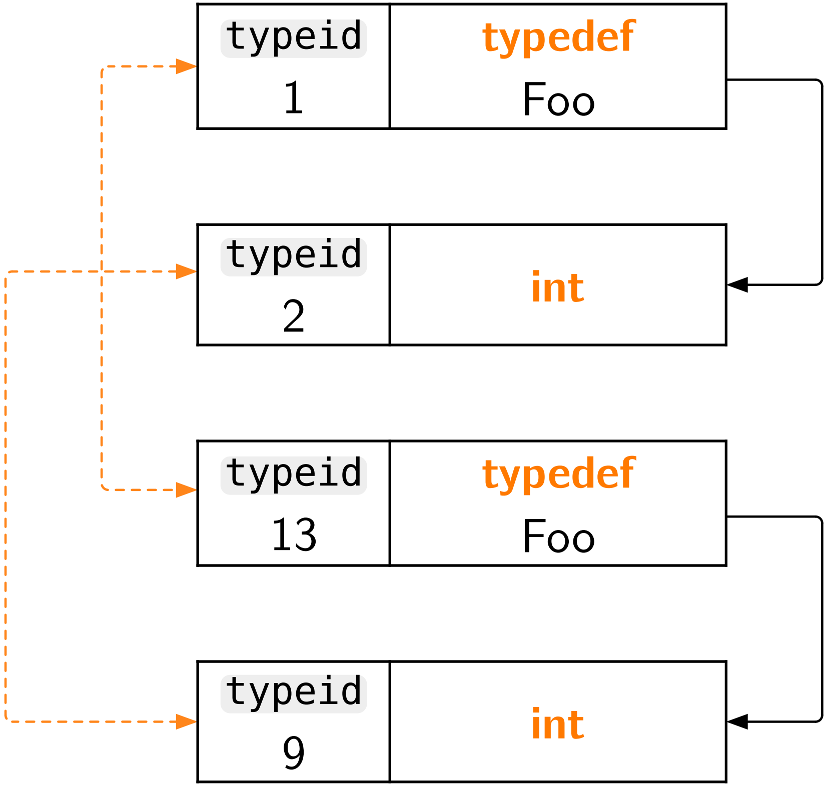
```c
typedef int foo;
foo add(foo a, foo b);

int main() {
    foo a = 5;
    foo b = 10;
    printf("%d", add(a,b));
    return 0;
}
```

c

b.c

```c
typedef int foo;

foo add(foo a, foo b){
    return a+b;
}
```

Without deduplication we get:



⇒ Let’s verify if typedefs are still dedupped

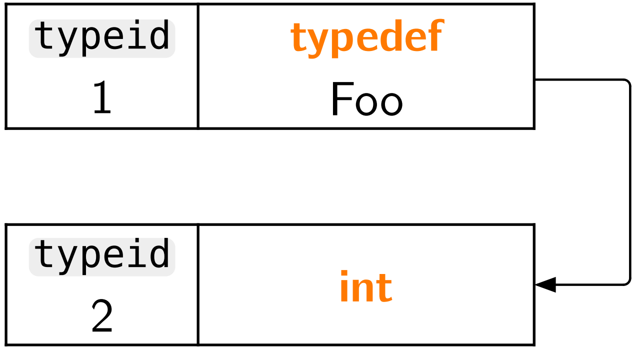
Soundness of Deduplication patch

a.c

```c
typedef int foo;
foo add(foo a, foo b);

int main() {
    foo a = 5;
    foo b = 10;
    printf("%d", add(a,b));
    return 0;
}
```

b.c

```c
typedef int foo;

foo add(foo a, foo b){
    return a+b;
}
```



Soundness of Deduplication patch

Executable binary	Original Version	Patched Version
readelf (C)	BTF file with same hash	
nm (C)	BTF file with same hash	
objdump (C)	BTF file with same hash	
lf (Go)	BTF file with same hash	
podman (Go)	crash 💥	👍
kubeadm (Go)	crash 💥	👍
kubectl (Go)	crash 💥	👍

Performance Overhead

Program	lf (Go)	readelf (C)	nm (C)	objdump (C)
Performance overhead (CPU cycles) ± 0.5%	+1.24%	+0.32%	+0.17%	+0.35%

Conclusion

Contributions:

1. We identified the issue preventing Pahole to analyze Go programs
2. We unravelled of the BTF deduplication algorithm and resolution of its limitations
3. We evaluated our patch of Pahole

Conclusion

Contributions:

1. We identified the issue preventing Pahole to analyze Go programs
2. We unravelled of the BTF deduplication algorithm and resolution of its limitations
3. We evaluated our patch of Pahole

Future work:

- Verify the existence of other recursive types in Go, and in other languages
- Extend verification of soundness
- Submit patch to the community