BFF: Foundational and Automated Verification of Bitfield-Manipulating Programs

Fengmin (Paul) Zhu Michael Sammler Rodolphe Lepigre
Derek Dreyer Deepak Garg

Max Planck Institute for Software Systems

December 9, 2022

Bit Operations Are Used to ...

- Perform efficient arithmetic computation
- Realize cryptography algorithms

Bit Operations Are Used to ...

- Perform efficient arithmetic computation
- Realize cryptography algorithms
- Bitfield Manipulation: manipulate the bitfields packed in integers, e.g., network packet headers, page table entries

Bit Operations Are Used to ...

- Perform efficient arithmetic computation
- Realize cryptography algorithms
- Bitfield Manipulation: manipulate the bitfields packed in integers, e.g., network packet headers, page table entries

Bitfield Manipulation: An Example

7	6	5	4	3	2	1	0
unused	owner				execute	write	read

Figure: A u8-integer encoding the metadata of a file header.

Bitfield Manipulation: An Example

7	6	5	4	3	2	1	0
unused	owner				execute	write	read

Figure: A u8-integer encoding the metadata of a file header.

Bitfield Manipulation: An Example

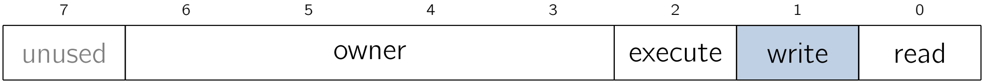


Figure: A `u8`-integer encoding the metadata of a file header.

Bitfield Manipulation: An Example

7	6	5	4	3	2	1	0
unused	owner				execute	write	read

Figure: A u8-integer encoding the metadata of a file header.

Bitfield Manipulation: An Example

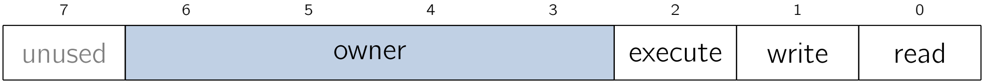


Figure: A `u8`-integer encoding the metadata of a file header.

Bitfield Manipulation: An Example

7	6	5	4	3	2	1	0
unused	owner				execute	write	read

Figure: A u8-integer encoding the metadata of a file header.

```c
#include <linux/bitops.h> // BIT macro
bool is_writeable(uint8_t header) {
  return (header & BIT(1)) != 0; // BIT(1) = 0b10
}
```

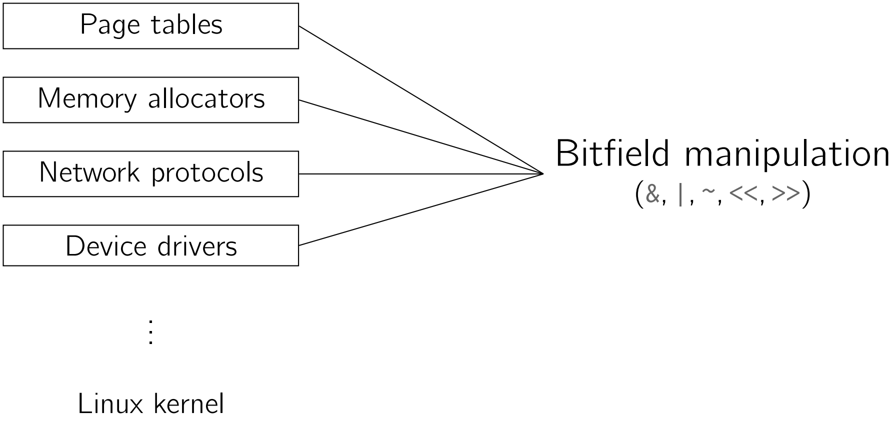
Bitfield Manipulation: An Example

7	6	5	4	3	2	1	0
unused	owner				execute	write	read

Figure: A u8-integer encoding the metadata of a file header.

```
#include <linux/bitops.h> // BIT macro
bool is_writeable(uint8_t header) {
  return (header & BIT(1)) != 0; // BIT(1) = 0b10
}
```

Bitfield Manipulation is Everywhere in Systems Programming



Bitfield Manipulation is Everywhere in Systems Programming

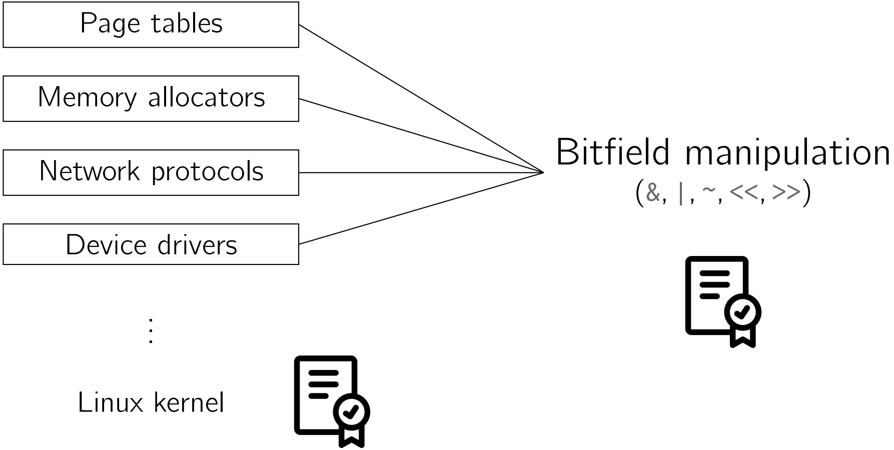
- Page tables
- Memory allocators
- Network protocols
- Device drivers

⋮

→ Bitfield manipulation
(&, |, ~, <<, >>)

Linux kernel

Bitfield Manipulation is Everywhere in Systems Programming



Two Goals, Simultaneously

Foundational

Automated

Two Goals, Simultaneously

Foundational

Proofs are machine-checkable in proof assistants

Automated

Two Goals, Simultaneously

Foundational	Proofs are machine-checkable in proof assistants
Automated	Proofs are mostly inferred

Our Starting Point: RefinedC

Automating the foundational verification of C code with refined ownership types [Sammler et al., PLDI 2021]

Our Starting Point: RefinedC

Automating the foundational verification of C code with refined ownership types [Sammler et al., PLDI 2021]

+ Bitfield manipulation support
(our work)

Refinement Types in RefinedC

$$\boxed{r \, @ \, \mathsf{tyConstr}\langle T_1, T_2, \ldots \rangle}$$

Refinement Types in RefinedC

r @ $\text{tyConstr}\langle T_1, T_2, \ldots \rangle$

Refinement Types in RefinedC

r @ tyConstr$\langle T_1, T_2, \ldots \rangle$

Refinement Types in RefinedC

$$\boxed{r \,@\, \text{tyConstr}\langle T_1, T_2, \ldots\rangle}$$

Builtin-types:

Integer type	$n \,@\, \text{int}\langle\alpha\rangle$
Boolean type	$b \,@\, \text{bool}$
Ownership type	$l \,@\, \&\text{own}\langle\tau\rangle$
$\ldots$	

$\{n\}$ where $n \in \mathbb{N}$ and $n \in \alpha$ ($\alpha \in \{\texttt{i32}, \texttt{u16}, \ldots\}$)

Refinement Types in RefinedC

$$r \,@\, \text{tyConstr}\langle T_1, T_2, \ldots \rangle$$

Builtin-types:

Integer type	$n \,@\, \text{int}\langle \alpha \rangle$
Boolean type	$b \,@\, \text{bool}$
Ownership type	$l \,@\, \&\text{own}\langle \tau \rangle$
...	

$\{b\}$ where $b \in \mathbb{B}$

Refinement Types in RefinedC

$$\boxed{r \mathbin{@} \mathrm{tyConstr}\langle T_1, T_2, \ldots\rangle}$$

Builtin-types:

Integer type	$n \mathbin{@} \mathrm{int}\langle\alpha\rangle$
Boolean type	$b \mathbin{@} \mathrm{bool}$
Ownership type	$l \mathbin{@} \&\mathrm{own}\langle\tau\rangle$
...	

$\{l\}$ where the location $l \mapsto v$ for some v of type τ

Refinement Types in RefinedC

$$\boxed{r \mathbin{@} \text{tyConstr}\langle T_1, T_2, \ldots\rangle}$$

Builtin-types:

Integer type	$n \mathbin{@} \text{int}\langle\alpha\rangle$
Boolean type	$b \mathbin{@} \text{bool}$
Ownership type	$l \mathbin{@} \&\text{own}\langle\tau\rangle$
$\ldots$	

Types are semantically defined (in Iris separation logic).

Refinement Types in RefinedC

$$\boxed{r \ @ \ \text{tyConstr}\langle T_1, T_2, \ldots \rangle}$$

Builtin-types:

Integer type	$n \ @ \ \text{int}\langle \alpha \rangle$
Boolean type	$b \ @ \ \text{bool}$
Ownership type	$l \ @ \ \&\text{own}\langle \tau \rangle$
...	

Types are semantically defined (in Iris separation logic).

Typing rules are propositions and their soundness has been proven in Coq-Iris.

Refinement Types in RefinedC

$$r \mathbin{@} \text{tyConstr}\langle T_1, T_2, \ldots \rangle$$

Builtin-types:

Integer type	$n \mathbin{@} \text{int}\langle \alpha \rangle$
Boolean type	$b \mathbin{@} \text{bool}$
Ownership type	$l \mathbin{@} \&\text{own}\langle \tau \rangle$
...	

Types are semantically defined (in Iris separation logic).

Typing rules are propositions and their soundness has been proven in Coq-Iris.

Naive Approach: Reuse Integer Types & Typing Rules in RefinedC

Q: How to automatically discharge the proof obligations in bit vector theory?

A: SMT solvers with a bit vector decision procedure.

[1] Studies show the presence of bugs [Mansur et al., FSE 2020, Winterer et al., PLDI 2020]

Naive Approach: Reuse Integer Types & Typing Rules in RefinedC

Q: How to automatically discharge the proof obligations in bit vector theory?

A: SMT solvers with a bit vector decision procedure.

Q: Can we do "RefinedC + SMT solver"?

A1: Hard because SMT solvers are not foundational[1]

[1]Studies show the presence of bugs [Mansur et al., FSE 2020, Winterer et al., PLDI 2020]

Naive Approach: Reuse Integer Types & Typing Rules in RefinedC

Q: How to automatically discharge the proof obligations in bit vector theory?

A: SMT solvers with a bit vector decision procedure.

Q: Can we do "RefinedC + SMT solver"?

A1: Hard because SMT solvers are not foundational[1]

A2: Not ideal because bit vector theory is too big a hammer for verifying bitfield manipulation (only a fragment is needed)

[1] Studies show the presence of bugs [Mansur et al., FSE 2020, Winterer et al., PLDI 2020]

Key Observations & Ideas

In bitfield-manipulating programs:

Key Observations & Ideas

In bitfield-manipulating programs:

- Bit vectors have structure – we call them structured bit vectors (SBVs)

Key Observations & Ideas

In bitfield-manipulating programs:

- Bit vectors have structure – we call them structured bit vectors (SBVs)

- Developers know the structure

Key Observations & Ideas

In bitfield-manipulating programs:

- Bit vectors have structure – we call them structured bit vectors (SBVs)
- Developers know the structure
- Valid bitfield manipulations only use bit operations following restricted patterns

Key Observations & Ideas

In bitfield-manipulating programs:

- Bit vectors have structure – we call them structured bit vectors (SBVs)

  → **Refinement types for SBVs**
- Developers know the structure
- Valid bitfield manipulations only use bit operations following restricted patterns

Key Observations & Ideas

In bitfield-manipulating programs:

- Bit vectors have structure – we call them structured bit vectors (SBVs)
  - → **Refinement types for SBVs**
- Developers know the structure
  - → **User annotations for bitfields**
- Valid bitfield manipulations only use bit operations following restricted patterns

Key Observations & Ideas

In bitfield-manipulating programs:

- Bit vectors have structure – we call them structured bit vectors (SBVs)
  - → **Refinement types for SBVs**
- Developers know the structure
  - → **User annotations for bitfields**
- Valid bitfield manipulations only use bit operations following restricted patterns
  - → **Typing rules restricted to common bitfield manipulation patterns**

Key Observations & Ideas

In bitfield-manipulating programs:

- Bit vectors have structure – we call them structured bit vectors (SBVs)

  → **Refinement types for SBVs**

- Developers know the structure

  → **User annotations for bitfields**

- Valid bitfield manipulations only use bit operations following restricted patterns

  → **Typing rules restricted to common bitfield manipulation patterns**

Our Approach

+ Refinement types for SBVs
+ User annotations for bitfields
+ Typing rules restricted to common bitfield manipulation patterns

BFF: Foundational and automated verification for bitfield manipulation

Let's Verify is_writeable

```c
bool is_writeable(uint8_t header) {
  return (header & BIT(1)) != 0;
}
```

Structures, Structures

This is in the user's head:

7	6	5	4	3	2	1	0
unused	owner				execute	write	read

The user attaches the following annotation to the C code:

```
//@rc::bitfields FileInfo as u8
//@ read    : bool[0]
//@ write   : bool[1]
//@ execute : bool[2]
//@ owner   : int[3..6]
//@rc::end
```

Structures, Structures

This is in the user's head:

7	6	5	4	3	2	1	0
unused	owner				execute	write	read

The user attaches the following annotation to the C code:

```
//@rc::bitfields FileInfo as u8
//@ read    : bool[0]
//@ write   : bool[1]
//@ execute : bool[2]
//@ owner   : int[3..6]
//@rc::end
```

Structures, Structures

This is in the user's head:

7	6	5	4	3	2	1	0
unused	owner				execute	write	read

The user attaches the following annotation to the C code:

```
//@rc::bitfields FileInfo as u8
//@ read    : bool[0]
//@ write   : bool[1]
//@ execute : bool[2]
//@ owner   : int[3..6]
//@rc::end
```

Structures, Structures

This is in the user's head:

7	6	5	4	3	2	1	0
unused	owner				execute	write	read

The user attaches the following annotation to the C code:

```
//@rc::bitfields FileInfo as u8
//@ read    : bool[0]
//@ write   : bool[1]
//@ execute : bool[2]
//@ owner   : int[3..6]
//@rc::end
```

Structures, Structures

This is in the user's head:

7	6	5	4	3	2	1	0
unused	owner				execute	write	read

The user attaches the following annotation to the C code:

```
//@rc::bitfields FileInfo as u8
//@ read    : bool[0]
//@ write   : bool[1]
//@ execute : bool[2]
//@ owner   : int[3..6]
//@rc::end
```

generates →

```
Record FileInfo := {
  read : bool;
  write : bool;
  execute : bool;
  owner : Z
}.
(* and other auxiliary
   definitions & lemmas *)
```

Formal Specification via RefinedC Annotations

```
[[rc::parameters("h : FileInfo")]]
[[rc::args("h @ bitfield<FileInfo>")]]
[[rc::returns("{h.(write)} @ builtin_boolean")]]
bool is_writeable(uint8_t header) {
  return (header & BIT(1)) != 0;
}
```

$\forall h : \texttt{FileInfo}$, this function returns a Boolean value h.write.

Formal Specification via RefinedC Annotations

```
[[rc::parameters("h : FileInfo")]]
[[rc::args("h @ bitfield<FileInfo>")]]
[[rc::returns("{h.(write)} @ builtin_boolean")]]
bool is_writeable(uint8_t header) {
  return (header & BIT(1)) != 0;
}
```

`rc::parameters`: declare universal-quantified (Coq) variables.

Formal Specification via RefinedC Annotations

```
[[rc::parameters("h : FileInfo")]]
[[rc::args("h @ bitfield<FileInfo>")]]
[[rc::returns("{h.(write)} @ builtin_boolean")]]
bool is_writeable(uint8_t header) {
  return (header & BIT(1)) != 0;
}
```

`rc::args`: assign refinement types to input arguments.

A New Refinement Type for SBVs

$$\boxed{r \; \texttt{@} \; \mathrm{bitfield}\langle R \rangle}$$

- parameter: a record type R

A New Refinement Type for SBVs

$$\boxed{r \; @ \; \text{bitfield}\langle R \rangle}$$

- parameter: a record type R
- refinement: a record term r of type R

A New Refinement Type for SBVs

$$\boxed{r \,@\, \mathrm{bitfield}\langle R \rangle}$$

- parameter: a record type R
- refinement: a record term r of type R
- semantically represents an integer type $[\![r]\!] \,@\, \mathrm{int}\langle \alpha_R \rangle$

A New Refinement Type for SBVs

$$\boxed{r \mathbin{@} \mathsf{bitfield}\langle R \rangle}$$

- parameter: a record type R
- refinement: a record term r of type R
- semantically represents an integer type $[\![r]\!] \mathbin{@} \mathsf{int}\langle \alpha_R \rangle$

$$[\![\cdot]\!] : \mathsf{SBV} \to \mathbb{Z}$$

Formal Specification via RefinedC Annotations

```
[[rc::parameters("h : FileInfo")]]
[[rc::args("h @ bitfield<FileInfo>")]]
[[rc::returns("{h.(write)} @ builtin_boolean")]]
bool is_writeable(uint8_t header) {
  return (header & BIT(1)) != 0;
}
```

Intuitively, let header $= [\![h]\!]$.

Formal Specification via RefinedC Annotations

```
[[rc::parameters("h : FileInfo")]]
[[rc::args("h @ bitfield<FileInfo>")]]
[[rc::returns("{h.(write)} @ builtin_boolean")]]
bool is_writeable(uint8_t header) {
  return (header & BIT(1)) != 0;
}
```

rc::returns: assign a refinement type to the return value.

Formal Specification via RefinedC Annotations

```
[[rc::parameters("h : FileInfo")]]
[[rc::args("h @ bitfield<FileInfo>")]]
[[rc::returns("{h.(write)} @ builtin_boolean")]]
bool is_writeable(uint8_t header) {
  return (header & BIT(1)) != 0;
}
```

`rc::returns`: assign a refinement type to the return value.

Now, BFF takes over the verification (automatically for this example).

A Template for Typing Rules

$$\frac{e_1 \triangleright_{\mathrm{e}} r_1 \ @\ \mathsf{bitfield}\langle R\rangle \qquad e_2 \triangleright_{\mathrm{e}} r_2 \ @\ \mathsf{bitfield}\langle R\rangle \qquad ?P}{(e_1\ op\ e_2) \triangleright_{\mathrm{e}} ?r \ @\ \mathsf{bitfield}\langle R\rangle}$$

where

- $e \triangleright_{\mathrm{e}} \tau$: type judgment on C-expressions
- $op \in \{\&, |, \sim, <<, >>\}$

A Template for Typing Rules

$$\frac{e_1 \rhd_e r_1 \mathbin{@} \text{bitfield}\langle R\rangle \qquad e_2 \rhd_e r_2 \mathbin{@} \text{bitfield}\langle R\rangle \qquad ?P}{(e_1\ op\ e_2) \rhd_e\ ?r \mathbin{@} \text{bitfield}\langle R\rangle}$$

where

- $e \rhd_e \tau$: type judgment on C-expressions
- $op \in \{\&, |, \sim, <<, >>\}$
- $?P$: additional restriction of bitfield manipulation

A Template for Typing Rules

$$\frac{e_1 \triangleright_{\mathrm{e}} r_1 \;@\; \mathrm{bitfield}\langle R\rangle \qquad e_2 \triangleright_{\mathrm{e}} r_2 \;@\; \mathrm{bitfield}\langle R\rangle \qquad ?P}{(e_1 \; op \; e_2) \triangleright_{\mathrm{e}} \; ?r \;@\; \mathrm{bitfield}\langle R\rangle}$$

where

- $e \triangleright_{\mathrm{e}} \tau$: type judgment on C-expressions
- $op \in \{\&, |, \sim, <<, >>\}$
- $?P$: additional restriction of bitfield manipulation
- $?r$: the resulting SBV s.t.

$$[\![?r]\!] = [\![op]\!]([\![e_1]\!], [\![e_2]\!])$$

A Template for Typing Rules

$$\frac{e_1 \rhd_{\mathrm{e}} r_1 \text{ @ bitfield}\langle R\rangle \qquad e_2 \rhd_{\mathrm{e}} r_2 \text{ @ bitfield}\langle R\rangle \qquad ?P}{(e_1\ op\ e_2) \rhd_{\mathrm{e}} ?r \text{ @ bitfield}\langle R\rangle}$$

where

- $e \rhd_{\mathrm{e}} \tau$: type judgment on C-expressions
- $op \in \{$&, |, ~, <<, >>$\}$
- $?P$: additional restriction of bitfield manipulation
- $?r$: the resulting SBV s.t.

  $$[\![?r]\!] = [\![op]\!]([\![e_1]\!], [\![e_2]\!])$$

  computed without any bitwise operators anymore

Masking Bitfields via Bitwise-AND

$$\frac{e_1 \triangleright_{\mathrm{e}} r_1 \text{ @ bitfield}\langle R \rangle \qquad e_2 \triangleright_{\mathrm{e}} r_2 \text{ @ bitfield}\langle R \rangle \qquad \text{is\_mask}(r_2)}{(e_1 \mathbin{\&} e_2) \triangleright_{\mathrm{e}} (r_1 \searrow r_2) \text{ @ bitfield}\langle R \rangle}$$

The rhs is a mask, where is_mask(r) iff:
for every field f of r, the value of f is either all-zero or all-one

Masking Bitfields via Bitwise-AND

$$\frac{e_1 \triangleright_e r_1 \mathbin{@} \mathsf{bitfield}\langle R\rangle \qquad e_2 \triangleright_e r_2 \mathbin{@} \mathsf{bitfield}\langle R\rangle \qquad \mathsf{is\_mask}(r_2)}{(e_1 \mathbin{\&} e_2) \triangleright_e (r_1 \searrow r_2) \mathbin{@} \mathsf{bitfield}\langle R\rangle}$$

Partially define an extraction operation $r_1 \searrow r_2$ if $\mathsf{is\_mask}(r_2)$:
extract from r_1 the bitfields specified by r_2.

Masking Bitfields via Bitwise-AND

$$\frac{e_1 \triangleright_e r_1 \mathbin{@} \text{bitfield}\langle R\rangle \qquad e_2 \triangleright_e r_2 \mathbin{@} \text{bitfield}\langle R\rangle \qquad \text{is\_mask}(r_2)}{(e_1 \mathbin{\&} e_2) \triangleright_e (r_1 \searrow r_2) \mathbin{@} \text{bitfield}\langle R\rangle}$$

Partially define an extraction operation $r_1 \searrow r_2$ if $\text{is\_mask}(r_2)$:
extract from r_1 the bitfields specified by r_2.

Lemma

If $\text{is\_mask}(r_2)$, *then* $[\![r_1 \searrow r_2]\!] = [\![r_1]\!] \mathbin{\&} [\![r_2]\!]$.

Masking Bitfields via Bitwise-AND

$$\frac{e_1 \rhd_{\mathrm{e}} r_1 \mathbin{@} \mathrm{bitfield}\langle R \rangle \qquad e_2 \rhd_{\mathrm{e}} r_2 \mathbin{@} \mathrm{bitfield}\langle R \rangle \qquad \mathrm{is\_mask}(r_2)}{(e_1 \mathbin{\&} e_2) \rhd_{\mathrm{e}} (r_1 \searrow r_2) \mathbin{@} \mathrm{bitfield}\langle R \rangle}$$

Partially define an extraction operation $r_1 \searrow r_2$ if $\mathrm{is\_mask}(r_2)$:
extract from r_1 the bitfields specified by r_2.

Lemma

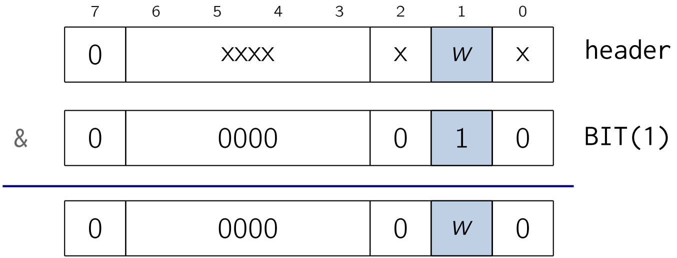
If $\mathrm{is\_mask}(r_2)$, *then* $[\![r_1 \searrow r_2]\!] = [\![r_1]\!] \mathbin{\&} [\![r_2]\!]$.

No bit operations required in the definition of $\searrow$!

Masking Bitfields via Bitwise-AND

$$\frac{e_1 \triangleright_{\mathrm{e}} r_1 \mathbin{@} \mathrm{bitfield}\langle R \rangle \qquad e_2 \triangleright_{\mathrm{e}} r_2 \mathbin{@} \mathrm{bitfield}\langle R \rangle \qquad \mathrm{is\_mask}(r_2)}{(e_1 \mathbin{\&} e_2) \triangleright_{\mathrm{e}} (r_1 \searrow r_2) \mathbin{@} \mathrm{bitfield}\langle R \rangle}$$

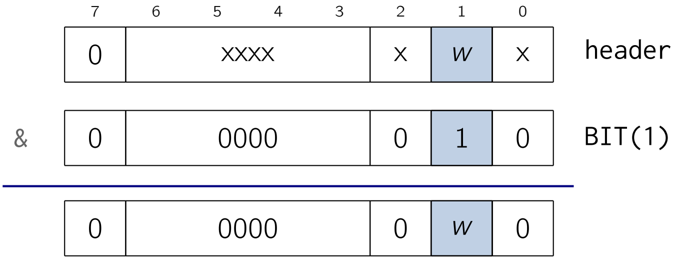
`header & BIT(1)`



Masking Bitfields via Bitwise-AND

$$\frac{e_1 \rhd_e r_1 \;@\; \text{bitfield}\langle R\rangle \qquad e_2 \rhd_e r_2 \;@\; \text{bitfield}\langle R\rangle \qquad \cancel{\text{is\_mask}(r_2)}}{(e_1 \,\&\, e_2) \rhd_e (r_1 \searrow r_2) \;@\; \text{bitfield}\langle R\rangle}$$

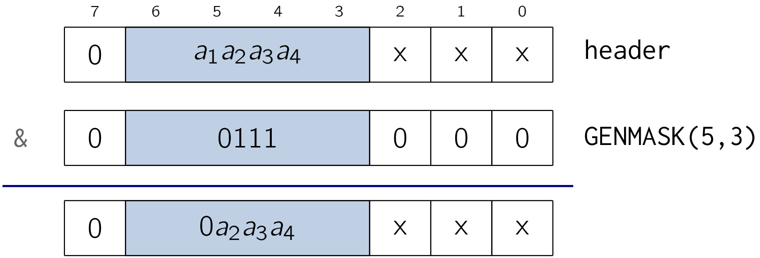
`header & BIT(1)`



Programmers Can Make Mistakes

The user expects to extract the owner field,

`header & GENMASK(5,3)`



but this expression does incomplete extraction.

Merging Bitfields via Bitwise-OR

$$\frac{e_1 \rhd_e r_1 \ \texttt{@}\ \mathsf{bitfield}\langle R\rangle \qquad e_2 \rhd_e r_2 \ \texttt{@}\ \mathsf{bitfield}\langle R\rangle \qquad r_1 \ \#\#\ r_2}{(e_1 \mid e_2) \rhd_e (r_1 \cup r_2)\ \texttt{@}\ \mathsf{bitfield}\langle R\rangle}$$

The bitfields specified in the two SBVs must be disjoint.

Merging Bitfields via Bitwise-OR

$$\frac{e_1 \triangleright_e r_1 \mathbin{@} \mathsf{bitfield}\langle R\rangle \qquad e_2 \triangleright_e r_2 \mathbin{@} \mathsf{bitfield}\langle R\rangle \qquad r_1 \mathbin{\#\#} r_2}{(e_1 \mid e_2) \triangleright_e (r_1 \cup r_2) \mathbin{@} \mathsf{bitfield}\langle R\rangle}$$

Partially define a merging operation $r_1 \cup r_2$ if $r_1 \mathbin{\#\#} r_2$:
merge (take the union of) the specified bitfields.

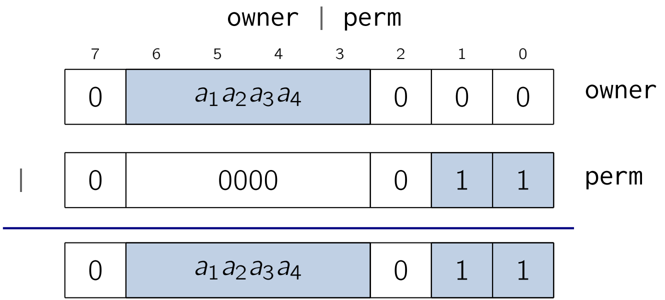
Lemma

If $r_1 \mathbin{\#\#} r_2$, then $[\![r_1 \cup r_2]\!] = [\![r_1]\!] \mid [\![r_2]\!]$.

No bit operations required in the definition of $\cup$!

Merging Bitfields via Bitwise-OR

$$\frac{e_1 \triangleright_e r_1 \;@\; \mathsf{bitfield}\langle R \rangle \qquad e_2 \triangleright_e r_2 \;@\; \mathsf{bitfield}\langle R \rangle \qquad r_1 \;\#\#\; r_2}{(e_1 \mid e_2) \triangleright_e (r_1 \cup r_2) \;@\; \mathsf{bitfield}\langle R \rangle}$$



A More Complicated Example

The above rules apply to the function taken from PKVM page table entry code:

```
void set_valid_leaf_pte(pte_t *ptep, u64 pa, pte_t attr) {
  pte_t pte = pa & PTE_ADDR_MASK;
  pte |= attr & (PTE_LEAF_ATTR_LO | PTE_LEAF_ATTR_HI);
  pte |= PTE_VALID;
  *ptep = pte;
}
```

➔ *See §2 of our paper*

More Typing Rules

In addition to masking & merging bitfields:

- Setting bitfields via `|`
- Clearing bitfields via `~` and `&`
- Reading bitfield values via `>>`
- Loading bitfield values via `<<`

➜ *See §3 of our paper*

Case Studies

Codebase	Lines of annotation/code	Side conditions	
		manual	total
#1 pgtable	0.83	0	75
#2 x86_pgtable	0.64	0	17
#3 tcp_input	0.79	0	0
#4 mt7601u	0.26	3	63
Total	0.42	3	155

Mostly automated; reasonable amount of annotations

➜ *See §7 of our paper*

Technical Issue on Implementing Typing Rules

$$\frac{e_1 \rhd_{\mathrm{e}} r_1 \mathbin{@} \mathrm{bitfield}\langle R\rangle \qquad e_2 \rhd_{\mathrm{e}} r_2 \mathbin{@} \mathrm{bitfield}\langle R\rangle \qquad \mathrm{is\_mask}(r_2)}{(e_1 \mathbin{\&} e_2) \rhd_{\mathrm{e}} (r_1 \searrow r_2) \mathbin{@} \mathrm{bitfield}\langle R\rangle}$$

Technical Issue on Implementing Typing Rules

$$\frac{e_1 \triangleright_e r_1 \mathbin{@} \text{bitfield}\langle R\rangle \qquad e_2 \triangleright_e r_2 \mathbin{@} \text{bitfield}\langle R\rangle \qquad \text{is\_mask}(r_2)}{(e_1 \mathbin{\&} e_2) \triangleright_e (r_1 \searrow r_2) \mathbin{@} \text{bitfield}\langle R\rangle}$$

Issue: since R is generic (user-defined), operators such as

$$\searrow : \forall (R : \text{Type}), R \to R \to R$$

are hard to implement in Coq.

Solution: A New Type Refined by Terms

$$t \; @ \; \text{bfterm}\langle \sigma \rangle$$

- parameter: a signature σ

```
//@rc::bitfields FileInfo as u8
//@ read    : bool[0]
//@ write   : bool[1]
//@ execute : bool[2]
//@ owner   : int[3..6]
//@rc::end
```

$\xrightarrow{\text{generates}}$

$\sigma_{\texttt{FileInfo}} \triangleq$
$[\langle 0, 1\rangle, \langle 1, 1\rangle, \langle 2, 1\rangle, \langle 3, 4\rangle]$

(range format: $\langle \mathit{offset}, \mathit{width}\rangle$)

Solution: A New Type Refined by Terms

$$\boxed{t \ @ \ \mathsf{bfterm}\langle\sigma\rangle}$$

- parameter: a signature σ
- refinement: a term t of sort σ

```
Definition header : FileInfo :=
  {| read := r; write := w; execute := x; owner := o |}.
```

$\xrightarrow{\text{represented by}}$

$$t_{\text{header}} \triangleq [\langle 0,1\rangle \mapsto r, \langle 1,1\rangle \mapsto w, \langle 2,1\rangle \mapsto x, \langle 3,4\rangle \mapsto o] : \sigma_{\texttt{FileInfo}}$$

Solution: A New Type Refined by Terms

$$\boxed{t \mathbin{@} \text{bfterm}\langle\sigma\rangle}$$

- parameter: a signature σ
- refinement: a term t of sort σ
- translation rule: $r \mathbin{@} \text{bitfield}\langle R\rangle \xrightarrow{\text{desugar to}} t_r \mathbin{@} \text{bfterm}\langle\sigma_R\rangle$

Solution: A New Type Refined by Terms

$$\boxed{t \mathbin{@} \mathrm{bfterm}\langle\sigma\rangle}$$

- parameter: a signature σ
- refinement: a term t of sort σ
- translation rule: $r \mathbin{@} \mathrm{bitfield}\langle R\rangle \xrightarrow{\text{desugar to}} t_r \mathbin{@} \mathrm{bfterm}\langle\sigma_R\rangle$

The typing rule used in verification:

$$\frac{e_1 \triangleright_e t_1 \mathbin{@} \mathrm{bfterm}\langle\sigma\rangle \qquad e_2 \triangleright_e t_2 \mathbin{@} \mathrm{bfterm}\langle\sigma\rangle \qquad \mathrm{is\_mask}(t_2)}{(e_1 \mathbin{\&} e_2) \triangleright_e (t_1 \searrow t_2) \mathbin{@} \mathrm{bfterm}\langle\sigma\rangle}$$

Solution: A New Type Refined by Terms

$$\boxed{t \mathbin{@} \mathrm{bfterm}\langle\sigma\rangle}$$

- parameter: a signature σ
- refinement: a term t of sort σ
- translation rule: $r \mathbin{@} \mathrm{bitfield}\langle R\rangle \xrightarrow{\text{desugar to}} t_r \mathbin{@} \mathrm{bfterm}\langle\sigma_R\rangle$

The typing rule used in verification:

$$\frac{e_1 \rhd_e t_1 \mathbin{@} \mathrm{bfterm}\langle\sigma\rangle \qquad e_2 \rhd_e t_2 \mathbin{@} \mathrm{bfterm}\langle\sigma\rangle \qquad \mathrm{is\_mask}(t_2)}{(e_1 \mathbin{\&} e_2) \rhd_e (t_1 \searrow t_2) \mathbin{@} \mathrm{bfterm}\langle\sigma\rangle}$$

Summary

Key Insight: typical bitfield manipulation operates on the logical, high-level structure of fields that are packed into integers/SBVs.

Implementation in RefinedC: new types for SBVs, typing rules with soundness proofs, meta-theory of SBV terms.

Our webpage:

`https://plv.mpi-sws.org/refinedc/bff`