BFF: Foundational and Automated Verification of Bitfield-Manipulating Programs

FENGMIN ZHU, MPI-SWS, Germany
MICHAEL SAMMLER, MPI-SWS, Germany
RODOLPHE LEPIGRE, MPI-SWS, Germany
DEREK DREYER, MPI-SWS, Germany
DEEPAK GARG, MPI-SWS, Germany

Low-level systems code often needs to interact with data, such as page table entries or network packet headers, in which multiple pieces of information are packaged together as bitfield components of a single machine integer and accessed via bitfield manipulations (*e.g.*, shifts and masking). Most existing approaches to verifying such code employ SMT solvers, instantiated with theories for bit vector reasoning: these provide a powerful hammer, but also significantly increase the trusted computing base of the verification toolchain.

In this work, we propose an alternative approach to the verification of bitfield-manipulating systems code, which we call BFF. Building on the RefinedC framework, BFF is not only highly *automated* (as SMT-based approaches are) but also *foundational*—*i.e.*, it produces a machine-checked proof of program correctness against a formal semantics for C programs, fully mechanized in Coq. Unlike SMT-based approaches, we do not try to solve the general problem of arbitrary bit vector reasoning, but rather observe that real systems code typically accesses bitfields using simple, well-understood programming patterns: the layout of a bit vector is known up front, and its bitfields are accessed in predictable ways through a handful of bitwise operations involving bit masks. Correspondingly, we center our approach around the concept of a *structured bit vector*—*i.e.*, a bit vector with a known bitfield layout—which we use to drive simple and predictable automation. We validate the BFF approach by verifying a range of bitfield-manipulating C functions drawn from real systems code, including page table manipulation code from the Linux kernel and the pKVM hypervisor.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; • **Theory of computation** → **Type theory**; **Automated reasoning**.

Additional Key Words and Phrases: bitfield manipulation, bit vectors, C programming language, refinement types, proof automation, Coq, Iris

ACM Reference Format:
Fengmin Zhu, Michael Sammler, Rodolphe Lepigre, Derek Dreyer, and Deepak Garg. 2022. BFF: Foundational and Automated Verification of Bitfield-Manipulating Programs. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 182 (October 2022), 26 pages. https://doi.org/10.1145/3563345

1 INTRODUCTION

Many systems programming applications written in C/C++ make use of *bit manipulation* operations—*e.g.*, bitwise logical operators (&, |, ~) and bit-shifting operators (<<, >>)—in order to precisely control the bit-level representation of data, manage space usage more efficiently, and interface with devices

Authors' addresses: Fengmin Zhu, MPI-SWS, Saarland Informatics Campus, Germany, paulzhu@mpi-sws.org; Michael Sammler, MPI-SWS, Saarland Informatics Campus, Germany, msammler@mpi-sws.org; Rodolphe Lepigre, MPI-SWS, Saarland Informatics Campus, Germany, lepigre@mpi-sws.org; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org; Deepak Garg, MPI-SWS, Saarland Informatics Campus, Germany, dg@mpi-sws.org.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2022 Copyright held by the owner/author(s).
2475-1421/2022/10-ART182
https://doi.org/10.1145/3563345

that dictate a particular bitwise data layout. Examples include the implementations of bitmaps, cryptographic primitives, and data compression algorithms.

A particularly important mode of use of bit manipulation operations is in interacting with *bit vectors containing structured data*, such as page table entries and network packet headers. *Logically*, such data takes the form of a record with multiple fields; but *physically*, the record is represented as a single machine integer, and its fields are represented as *bitfields* of the integer—*i.e.*, specific, pre-determined bit ranges of the integer with enough bits to store the field's data. Such bitfield representations are often used to store data compactly, even when the data representation is not mandated by specific hardware or communication protocols. For example, one might prefer to store file permission flags (read, write, execute) in three different bits of a byte, instead of using a struct with three boolean fields that would each take one byte.[1]

In order to verify correctness of programs with bitfield manipulation—or more generally any bit manipulation—a popular approach is to use SMT solvers in conjunction with a theory for bit vector reasoning [Barrett et al. 2010]. SMT solvers provide an effective hammer for this task, but they are also complex pieces of unverified software which significantly increase the trusted computing base (TCB) of the verification toolchain. For example, recent work has uncovered many critical bugs in widely-used SMT solvers like Z3 and CVC4 [Winterer et al. 2020b,a; Mansur et al. 2020; Park et al. 2021]. In the interest of building high-assurance verification artifacts for low-level systems code, it is therefore worth exploring alternative *foundational* approaches to the problem, wherein the verification of bitfield-manipulating programs is embedded in a general-purpose theorem prover like Coq or Isabelle, whose TCB (*e.g.*, the Coq kernel) is much smaller and better understood.

In this paper, we present **BFF**, a new approach to the verification of bitfield-manipulating programs that is not only *automated* (as SMT-based approaches are) but also *foundational*. To achieve this combination of features, BFF builds on the recently introduced RefinedC framework [Sammler et al. 2021; Lepigre et al. 2022]. RefinedC employs a *refinement type* system for enforcing functional correctness properties for C data types and functions. Under the RefinedC approach, types are parameterized by a *refinement*, which provides additional information about the inhabitants of the type that is useful for driving automatic type checking. The soundness of the type system is then established via a *semantic soundness* proof [Ahmed et al. 2010; Jung et al. 2018], whereby RefinedC's typing rules are proven sound in Coq against a semantic model of RefinedC's types formalized in the Iris separation logic. This type-based approach is both compositional and automated, but also results in an end-to-end verification of C programs against a formal semantics for C programs, fully mechanized in Coq.

The key contribution of BFF over RefinedC is its support for verifying *bitfield manipulations*. Toward this end, we take a different tack than SMT-based approaches do. Rather than reducing the problem of verifying bitfield manipulation to the more general (and harder) problem of arbitrary bit vector reasoning, we instead observe that real systems code typically accesses bitfields using relatively simple, well-understood programming patterns: the layout of a bit vector (*i.e.*, its bitfield structure) is known up front, and its bitfields are accessed in predictable ways through a handful of bitwise operations involving *bit masks*.

Correspondingly, we center our approach around the concept of a *structured bit vector* (or SBV, for short)—*i.e.*, a bit vector with a known bitfield layout. We first formalize a simple typed language of *SBV descriptors* for describing the logical structure of SBVs and ensuring that standard operations involving bit masks preserve that structure. We make critical use of this SBV descriptor language

[1]The C language also supports declaring struct members with an explicit width in bits, but the ISO C standard does not guarantee such fields to be tightly packed, and hence this mechanism is not widely used for portability reasons. We ignore this feature throughout.

in BFF's internal automation: in particular, thanks to the clear logical structure of SBVs, it is easy to automatically decide whether two SBV descriptors encode the same machine integer.

To incorporate this automation into RefinedC, we rely on the hook provided by RefinedC's refinement types. Specifically, we first introduce a new *refinement type of structured bit vectors*, r@bitfield$\langle R\rangle$, which is a subtype of the integer type. This type is parameterized by a (Coq) record type R describing the bitfield layout of an SBV, and refined by a (Coq) record r of type R specifying the contents of the SBV's bitfields. This record-based representation provides a clean, high-level interface for users of BFF to describe the bitfield layout of their bit vectors. Internally, however, we translate r and R into a term and its corresponding type in our (lower-level) SBV descriptor language, which we can manipulate efficiently and on which we can then apply our decision procedure for SBV equality.

Put together, this approach enables us to extend the benefits of the RefinedC approach—automated and predictable, yet foundational verification—with support for common bitfield manipulation idioms. We validate the BFF approach by verifying a range of bitfield-manipulating C functions drawn from real systems code, including page table manipulation code from the Linux kernel and the pKVM hypervisor. Our work is formalized in Coq, and our artifact and repository are open access [Zhu et al. 2022].

Contributions.

- BFF: a new approach to the automatic and foundational verification of bitfield-manipulating programs based on (1) carefully characterizing the standard bitfield-manipulation patterns and (2) establishing that they preserve the abstraction of structured bit vectors. (§3)
- A formalization of structured bit vectors and their meta-theory, including operations for merging, extracting, and clearing, and a sound and complete equality checking procedure, all formally verified in Coq. (§4)
- An integration of BFF into RefinedC, a foundational and automated framework for C program verification. (§5, §6)
- An evaluation of the BFF automation using bitfield-manipulating programs from real-world systems code, including page-table management code drawn from the Linux kernel and the pKVM hypervisor. (§7)

We begin in §2 with an overview of the BFF approach by example, discuss related work in §8, and conclude in §9.

2 THE BFF APPROACH BY EXAMPLE

We start by demonstrating how the BFF approach enables verification of bitfield-manipulating programs through a small but illustrative example. Our example consists of two functions that manipulate page table entries (PTEs), both taken from pKVM, an in-development, open source hypervisor from Google [Deacon 2020; Edge 2020]. We first present the (simplified) C code of the two functions (§2.1). Then, we describe how the correctness of functions is *specified* in BFF (§2.2). Next, we explain how specifications are *automatically verified* by BFF (§2.3). Finally, we explain how BFF applies to programs that manipulate *nested* bitfields (§2.4).

2.1 Manipulating Page Table Entries

Our two example functions, shown in Fig. 1, manipulate page-table entries (PTEs) of the Armv8 architecture. *Concretely*, a PTE is a 64-bit unsigned integer, and is represented by the C type `pte_t`, defined on line 1. *Logically*, however, a PTE contains five *bitfields* that span specific, contiguous but non-overlapping bits of the PTE. For example, the first bitfield is 1-bit wide (at position 0) and codes whether the whole PTE is valid or not. There is also a 36-bit page address bitfield spanning

```
1  typedef u64 pte_t;
2  #define PTE_VALID            BIT(0)
3  #define PTE_TYPE             BIT(1)
4  #define PTE_LEAF_ATTR_LO     GENMASK(11, 2)
5  #define PTE_ADDR_MASK        GENMASK(47, 12)
6  #define PTE_LEAF_ATTR_HI     GENMASK(63, 51)
7
8  bool pte_valid(pte_t pte) { return pte & PTE_VALID; }
9
10 void set_valid_leaf_pte(pte_t *ptep, u64 pa, pte_t attr) {
11   pte_t pte = pa & PTE_ADDR_MASK;
12   pte |= attr & (PTE_LEAF_ATTR_LO | PTE_LEAF_ATTR_HI);
13   pte |= PTE_VALID;
14   *ptep = pte;
15 }
```

Fig. 1. C code of two page table manipulation functions from the pKVM hypervisor (simplified).

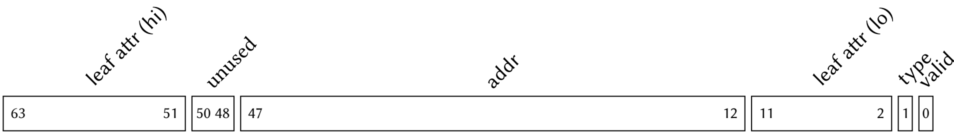

Fig. 2. Logical bitfields of a page table entry.

bit positions 12–47. Bits 48–50 of the PTE are unused and are not a part of any logical bitfield. The entire layout of the bitfields in a PTE is depicted in Fig. 2. Such types as `pte_t`, which contain bitfields with a known structure, we refer to as *structured bit vector* or SBV types.

Bitfield masks. Lines 2-6 of Fig. 1 define five *bitfield mask*s, one for each of the five logical bitfields of the PTE (including the page address). A bitfield mask for a bitfield is a constant integer, which contains 1s at the bit positions spanned by the bitfield and 0s at other positions. (The standard Linux macro `BIT(n)` expands to a bitfield mask for a single-bit bitfield located at position `n`, while `GENMASK(n,m)` returns a bitfield mask for a bitfield spanning the bit positions `n` to `m`.)

Bitfield-manipulating functions. Next, we describe two functions that manipulate PTEs. The function `pte_valid` extracts the 1-bit validity bitfield using a bitwise `&` between the PTE and the bitfield mask `PTE_VALID` for the valid field. The result is implicitly cast to the return type `bool`.

Note how the function `pte_valid` relies only on the *logical structure* of the PTE; the concrete layout of the PTE (specifically, that the valid bitfield is at position 0) is completely hidden behind the definition of the bit mask `PTE_VALID`. The exact same function would be correct if the valid bitfield were moved to a different position in `pte_t` and the bitfield mask `PTE_VALID` were correspondingly redefined. Using the bitwise `&` operator with a pre-defined bitfield mask to extract the corresponding bitfield is actually *the* standard programming pattern for extracting a bitfield from an SBV.

Our second function `set_valid_leaf_pte` illustrates bitfield manipulation further. It generates a new PTE by first extracting the addr bitfield from a given PTE (`pa`), then extracting two other fields—leaf attr (hi) and leaf attr (lo)—from a different given PTE (`attr`), and finally setting these extracted bitfields as well as the valid bitfield in the output PTE. The function uses the above pattern of `&`-ing with a bitfield mask to extract bitfields from the given PTEs and uses the `|` operator to

```
1  //@rc::bitfields Pte as u64
2  //@  pte_valid        : bool[0]
3  //@  pte_type         : int[1]
4  //@  pte_leaf_attr_lo : int[2 ..11]
5  //@  pte_addr         : int[12..47]
6  //@  pte_leaf_attr_hi : int[51..63]
7  //@rc::end
8
9  //@rc::typedef pte_t = bitfield<Pte>
10
11 [[rc::parameters("pte : Pte")]]
12 [[rc::args("pte @ pte_t")]]
13 [[rc::returns("{pte.(pte_valid)} @ builtin_boolean")]]
14 static bool pte_valid(pte_t pte) { /* code unchanged */ }
15
16 //@rc::inlined_prelude
17 //@Definition svl_pte pa attr := {|
18 //@  pte_addr  := pa.(pte_addr);
19 //@  pte_valid := true;
20 //@  pte_type  := 0;
21 //@  pte_leaf_attr_lo := attr.(pte_leaf_attr_lo);
22 //@  pte_leaf_attr_hi := attr.(pte_leaf_attr_hi);
23 //@|}.
24 //@rc::end
25
26 [[rc::parameters("p: loc", "old, pa, attr: Pte")]]
27 [[rc::args("p @ &own<old @ pte_t>", "pa @ pte_t", "attr @ pte_t")]]
28 [[rc::ensures("own p: {svl_pte pa attr} @ pte_t")]]
29 static void set_valid_leaf_pte(pte_t *ptep, u64 pa, pte_t attr) { /* code unchanged */ }
```

Fig. 3. Adding specifications for Fig. 1 using RefinedC annotations.

set bitfields in the PTE pointed by the first argument. This use of the `|` operator is the standard programming pattern for setting bitfields.

We list some key takeaways that we exploit in BFF. First, even though bitfields are *concretely* just bit ranges packed into integer types like `pte_t`, such *structured bit vector* or SBV integer types have a higher-level *logical* structure, which specifies which bitfields exist. It is this logical structure that most programmers working with bitfields care about and that most functions manipulate logically. Second, *bitfield masks*, which are typically defined once for each SBV type like `pte_t`, map the logical list of bitfields in a structured bit vector type to their concrete bit ranges. Third, all subsequent manipulation (extraction, setting, unsetting) of bitfields is done at the granularity of entire bitfields using bitfield masks, and relies only on the logical layout (defined by the masks), not the concrete layout. Finally, only a handful of programming patterns that rely on bit masks and bitwise operators are used to perform nearly all bitfield manipulation (two patterns with `&` and `|` shown above; the remaining are enumerated in §3).

2.2 BFF Specifications

Next, we describe how functional correctness is *specified* in BFF. We would like to specify bitfield-manipulating code in terms of its effects on *logical* bitfields, not the *concrete* bit layout because, as noted above, programmers typically think about bitfield manipulation in terms of the logical structure. As an example, to a first approximation, we specify the function `pte_valid` as "return `true` if the `PTE_VALID` field is set and `false` otherwise" (the precise specification is in Fig. 3). In contrast,

a specification based on the concrete bit layout would be "return `true` if the lowest (index 0) bit is 1 and `false` otherwise". Clearly, the BFF specification is higher-level and closer to how most programmers would think of what the `pte_valid` function does.

In the following, we explain BFF specifications using the code of Fig. 1 as an example. Broadly speaking, in BFF, a programmer needs to provide two kinds of specifications—for SBV types and for functions. Both kinds of specifications for our example are shown in Fig. 3 and explained below.

SBV specifications. For every SBV type like `pte_t`, the programmer specifies—once and for all—which bitfields the type holds (the logical structure) and the concrete bit ranges of each of those bitfields (the mapping from the logical to the concrete structure). This is done with a `rc::bitfields` annotation as on lines 1-7 of Fig. 3. This specification provides the following information to BFF:

(1) The width of the SBV (here, `u64`, which is our notation for "64 bits wide").
(2) The names of all the bitfields (the logical structure).
(3) The logical type of the values contained in each bitfield. Here, the bitfield `pte_valid` has logical type `bool` and the remaining four bitfields logically contain `integers`.
(4) The bit ranges of each of the bitfields (the concrete structure). These are specified via `[m .. n]`, indicating the bit range from position m to n (inclusive). (We use `[m]` as shorthand for `[m .. m]`.)
(5) A name for this specification, which is `Pte` in this example (line 1).

Our BFF frontend automatically converts every such `rc::bitfields` specification of an SBV into a *Coq record type* of the same name, which contains only the logical structure of the SBV, and a separate *signature*, which contains the concrete layout. For the purposes of this section, only the Coq record type is relevant, and we show it below. The signature is relevant internally in BFF's type system for verification, and is introduced in §2.3.

```
Record Pte :=
  { pte_valid : bool; pte_type : Z; pte_leaf_attr_lo : Z; pte_addr : Z; pte_leaf_attr_hi : Z }.
```

Note how the frontend has mapped the logical types of bitfield values `bool` and `int` to the Coq types `bool` and `Z`, respectively. An instance of this record type is a logical representation of the *contents* of a `pte_t` SBV, structured into the contents of each bitfield.

Technically, record types generated by the frontend (such as `Pte` above) are instances of a new BFF-defined Coq type class `BitfieldDesc`. This type class provides a representation function that maps a record of the type to a C integer representing the record's bitfields concretely. Our frontend actually generates this entire type class instance from the `rc::bitfields` annotation, including this representation function.

Function specifications. To specify functions, we rely on RefinedC's refinement types. In RefinedC, types can be *refined* by a Coq element of a Coq type. The RefinedC type `t` refined by the element `e` (of the specific Coq type) is written `e @ t`. A C expression `e` has this refined type if `e` has type `t` and it represents the element `e` logically. The mapping between logical and concrete values is specified in the semantic definition of the refined type. For example, the RefinedC `int` type can be refined by Coq integers `Z`: For `z` in `Z`, `z @ int` is the (singleton) type containing only one C integer, namely, the one that represents the Coq integer `z`.

Our insight is that to reason about an SBV type, we can refine it with elements of the corresponding Coq record type generated from its specification. For example, on line 9, we specify that the RefinedC type `pte_t`—the RefinedC analogue of the C type `pte_t`—is refined by records of the type `Pte`. We do this by defining `pte_t` as an alias for the type `bitfield<Pte>`. Here, `bitfield<R>` is a new BFF type that is parameterized by a Coq type `R`, which must be an instance of the type class `BitfieldDesc`. The type `bitfield<R>` can be refined by elements of type `R`, and this refinement is defined so that for any `r: R`, the RefinedC type `r @ bitfield<R>` is the singleton set of the C integer that concretely

represents the logical bitfields in the record `r`. (The semantic definition of `r @ bitfield<R>` relies on the representation function of `R` provided by the type class `BitfieldDesc`.)

Hence, by declaring `pte_t` as `bitfield<Pte>`, for every `pte : Pte` (in Coq), the RefinedC/BFF type `pte @ pte_t` contains exactly the one C integer which concretely represents the logical bitfield record `pte`. Such a type gives us a one-to-one mapping between C SBVs and Coq records representing their bitfields logically. Function specifications then relate concrete inputs to outputs by relating the corresponding Coq records.

Building on this idea, Fig. 3 (Lines 11-13) show the full specification of the function `pte_valid`. Recall that this function returns true if the `pte_valid` bitfield is set and false otherwise. In the specification, `rc::parameters` is RefinedC notation for *universally quantified* parameters that stand for Coq values, `rc::args` is the RefinedC annotation for the refined types of the function arguments in order, and `rc::returns` specifies the refined type of the value returned by the function. Here, the specification says that for any Coq value `pte` (of the record type `Pte`), if the function's argument is of type `pte @ pte_t`, *i.e.*, the argument is the concrete C representation of the structured bitfield record `pte`, then the result has type `pte.pte_valid @ builtin_boolean`, *i.e.*, it is the concrete C representation of the logical (Coq) boolean `pte.pte_valid`.

It is instructive to note how this specification relates the concrete C input to the concrete C output by specifying the logical output (`pte.pte_valid`) in terms of the logical input (`pte`). The *singleton* semantics of the refinement types `pte @ pte_t` and `pte.pte_valid @ builtin_boolean` imply that the concrete C values correspond to these logical values, so this fully specifies the function's correctness.

The second function `set_valid_leaf_pte` is specified similarly, but is more nuanced. Here, the first argument is a pointer to a `pte_t`, where the output is to be written. The output is obtained by combining fields from the second and third arguments, which have the C type `pte_t`.

In RefinedC, a pointer is refined by an abstract location of a predefined Coq type `loc`. An additional important concept is that of logical ownership of pointers: To successfully type-check a pointer written in RefinedC, the pointer must be owned by the code being executed. Ownership is represented by the type modifier `&own<t>`, which represents an owned pointer pointing to something of type `t`. Finally, a function's side effects on state are specified using the `rc::ensures` clause, which provides a predicate on the state at the end of the function.

The specification of `set_valid_leaf_pte` says that if the first argument is an *owned* pointer `p` pointing to a value of type `old @ pte_t` (`old` is irrelevant here since it will be overwritten anyhow), and the remaining two arguments (of the RefinedC type `pte_t`) are logically represented by the records `pa` and `attr` (of the Coq type `Pte`), then at the end of the function, `p` has been overwritten by the C integer (of type `pte_t`) which logically represents the Coq record `svl_pte pa attr`. Here, `svl_pte` is a Coq function of type `Pte -> Pte -> Pte` that describes the entire logic of the function `set_valid_leaf_pte`, *i.e.*, how the bitfields of the output are (logically) computed from those of the two inputs; it is defined on lines 17-23.

Again, the specification works entirely at the level of logical bitfields encoded in the the Coq record type `Pte`. These logical bitfields do not refer to the concrete layouts as those layouts are hidden in the semantic intepretation of the refined type `pte @ pte_t`, freeing the programmer from having to worry about them.

2.3 Verifying BFF Specifications

Next, we describe how BFF verifies functions against their specifications automatically. RefinedC, the framework that BFF extends, relies on an automated type system embedded in Iris. BFF extends RefinedC with new rules for typing applications of bitwise operators (`&`, `|`, etc.) to SBVs, *i.e.*, C expressions with the refined `bitfield<R>` types from §2.2. However, as we explain below, working

with the `bitfield<R>` type is technically inconvenient, so we define a new family of refined types that carry the same information but have refinements with a simpler structure. In the following, we first motivate why we need this second family of refined types and then illustrate our typing rules. A detailed description of typing rules is presented in §3.

The structure of BFF typing rules. At a high level, BFF provides one typing rule for every possible pattern of use of a bitwise operator for bitfield operations. We saw two such patterns in §2.1: the `&` operator applied to an SBV and a bitfield mask to extract a bitfield, and the `|` operator applied to two SBVs with disjoint bitfields already set to get a new SBV with the union of the bitfields set.

Just to illustrate this approach, what might a typing rule for the second pattern be? In RefinedC, the typing judgment "e has type ty" is written $e \triangleright_e ty$, and the rule for the second pattern should have the form

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

which says that if e_1, e_2 are C expressions that evaluate to the concrete representations of the bitfield records r_1, r_2 (both of the Coq type R), then the C expression $e_1|e_2$ will evaluate to the concrete representation of the bitfield record $r_1 \cup r_2$ (also of type R). Here, r_1 and r_2 must have *disjoint* fields set (premise $r_1 \mathbin{\#\#} r_2$), and $r_1 \cup r_2$ is the "merge" of these records, which copies each bitfield's value from whichever of r_1 and r_2 that bitfield is set in (the bitfield's value is 0 if it is 0 in both r_1, r_2).

The need for a new refinement type. Although the above typing rule is intuitively correct, defining disjointness $r_1 \mathbin{\#\#} r_2$ and the merge operator $r_1 \cup r_2$ *parametrically* in the record type R is impossible in Coq. The best we could do is to have our frontend automatically generate *separate* definitions of disjointness and the merge operator for every bitfield record type R, but then the frontend would also have to automatically generate separate type-rule soundness proofs for every record type R. This is a formidable exercise.

Our solution to this problem is to introduce a second RefinedC type that carries the same information as `r @ bitfield<R>`, but whose refinements are of a *single* Coq type, independent of `R`. We call the new RefinedC type bf_term and the single Coq type `sbvd` (abbrv. for SBV Descriptor). The frontend automatically generates a function to *elaborate* the type `r @ bitfield<R>` into the new bf_term type when it generates `R` from the corresponding `rc::bitfields` declaration. Our typing rules work only with bf_term, so we have to define disjointness and the merge operator only once on `sbvd`, which is feasible and easy. This eliminates the problem of the previous paragraph.

Note that programmers do not see bf_term or `sbvd` in the ordinary course of things as these are only used *internally* by BFF for type-checking. This allows us to make `sbvd` *low-level*: its elements refer to the concrete layouts of bitfields, which the type `r @ bitfield<R>` (that programmers interact with) hides purposefully.

***The new RefinedC type* bf_term *and the new Coq type* `sbvd`.** Our new RefinedC type for representing SBVs logically has the form $\mathsf{bf_term}\langle\sigma, \alpha\rangle$. Here, α is the width of the SBV and σ is a *signature*—a list of bitfields, specified in terms of the precise offset and the width of each bitfield. For example, for the SBV type `pte_t`, α would be `u64` (64 bits), and the signature would be

$$\sigma_{\mathrm{PTE}} = [\mathsf{atom}(\langle 0, 1\rangle), \mathsf{atom}(\langle 1, 1\rangle), \mathsf{atom}(\langle 2, 10\rangle), \mathsf{atom}(\langle 12, 36\rangle), \mathsf{atom}(\langle 51, 13\rangle)]$$

meaning that the SBV of type `pte_t` has a bitfield of length 1 bit at offset 0, a bitfield of length 1 at offset 1, a bitfield of length 10 at offset 2, and so on. These correspond exactly to the five bitfields declared in the `rc::bitfields` declaration on line 1 of Fig. 3.[2]

The type $\text{bf_term}\langle\sigma, \alpha\rangle$ is refined by terms of a new Coq type `sbvd`, whose inhabitants, called *terms*, are denoted t. A term t : `sbvd` is a list of values of different bitfields. The bitfields are (again) referenced by their offsets and lengths, written $\langle a, k\rangle$ where a is the offset and k is the length. For example, a term corresponding to the type `pte_t` would have the form

$$t_{\text{pte}} = \langle 0, 1\rangle \mapsto \text{data}(a) :: \langle 1, 1\rangle \mapsto \text{data}(b) :: \langle 2, 10\rangle \mapsto \text{data}(c) \quad (1)$$

$$:: \langle 12, 36\rangle \mapsto \text{data}(d) :: \langle 51, 13\rangle \mapsto \text{data}(e) :: \text{nil} \quad (2)$$

where a–e are Coq values representing the logical contents of the five bitfields. If a bitfield's value is zero, the corresponding bitfield can be omitted altogether. Thus, if $a = e = 0$, then the above term would be equivalent (at the type `sbvd`) to

$$t'_{\text{pte}} = \langle 1, 1\rangle \mapsto \text{data}(b) :: \langle 2, 10\rangle \mapsto \text{data}(c) :: \langle 12, 36\rangle \mapsto \text{data}(d) :: \text{nil} \quad (3)$$

The refined type $t@\text{bf_term}\langle\sigma, \alpha\rangle$ is the singleton set of C values that concretely represent the logical bit vector t.

Note that it does not make sense to refine $\text{bf_term}\langle\sigma, \alpha\rangle$ with a term t if the offsets and lengths in t and σ do not match. For this reason, terms t can be *sorted* by signatures σ, and the refined type $t@\text{bf_term}\langle\sigma, \alpha\rangle$ is well-formed only if t has sort σ. We explain this sorting discipine in §4.

Operations on terms. Disjointness and merging can be defined very easily on terms of type `sbvd`. Two terms t_1, t_2 are disjoint ($t_1 \# \# t_2$) if they do not simultaneously have non-zero data in respective fields at the same offset. The merging operator $t_1 \cup t_2$, defined only for disjoint terms, copies every bitfield from whichever term has a non-zero value for that bitfield. If neither term has a non-zero value, the bitfield is 0. A third operator, which we have not discussed yet, is *extraction*, $t_1 \searrow t_2$. This operator is defined only when t_2 is a bitfield mask—a term where every bitfield is either all 0s or all 1s. The operator implements the common masking or extraction operation logically: It copies from t_1 all bitfields that are set to 1s in t_2 and sets the rest to 0.

Typing rules, revisited. BFF's typing rules rely on the new RefinedC type $t@\text{bf_term}\langle\sigma, \alpha\rangle$. For example, the typing rule for merging two SBVs using `|` shown earlier is actually

$$\textsc{Ty-merge}\quad \frac{e_1 \triangleright_e t_1@\text{bf_term}\langle\sigma, \alpha\rangle \qquad e_2 \triangleright_e t_2@\text{bf_term}\langle\sigma, \alpha\rangle \qquad t_1 \# \# t_2}{(e_1 \mid e_2) \triangleright_e (t_1 \cup t_2)@\text{bf_term}\langle\sigma, \alpha\rangle}$$

Similarly, the typing rule for the use of `&` to extract/mask bitfields using a bitfield mask is

$$\textsc{Ty-mask}\quad \frac{e_1 \triangleright_e t_1@\text{bf_term}\langle\sigma, \alpha\rangle \qquad e_2 \triangleright_e t_2@\text{bf_term}\langle\sigma, \alpha\rangle \qquad \text{is_mask}(t_2)}{(e_1 \,\&\, e_2) \triangleright_e (t_1 \searrow t_2)@\text{bf_term}\langle\sigma, \alpha\rangle}$$

Here, the third premise checks that t_2 is indeed a bit mask and $t_1 \searrow t_2$ defined in the previous paragraph performs the bitfield extraction operation at the logical level (*i.e.*, on the terms t_1, t_2).

Typing rules for other common uses of bitwise operations on bitfields are shown in §3.

[2]The purpose of the constructor atom(...) will become clear in §2.4, where we introduce a second constructor to support nested bitfields.

Verification workflow. Before we describe BFF's verification in detail on an example, we give an overview of the overall verification workflow:[3]

(1) The programmer adds RefinedC annotations to the C source files. These annotations include the declaration of SBV types and specifications for bitfield-manipulating functions using the r@bitfield$\langle R\rangle$ type, as explained in §2.2.
(2) The frontend processes the C files and generates (a) Coq representations of the C functions in RefinedC's formalization of C (Caesium), (b) Coq declarations for the type of each function from the annotations, (c) lemmas that show that each function from (a) has the expected type from (b), and (d) other auxiliary definitions such as the signatures of SBVs from `rc::bitfields` annotations.
(3) Coq checks the generated definitions and lemmas. Each lemma comes with a standard proof script that invokes RefinedC's type checking procedure, which is implemented as a Coq tactic. This procedure uses the BFF typing rules along with automatic solvers to discharge as many side conditions as possible.
(4) For the remaining side conditions, the user determines (*e.g.*, by stepping into the Coq proofs) if (a) they are caused by a bug in the program or (b) by an incompleteness of the solvers (*e.g.*, because the side condition goes beyond the supported fragment). In case of (a), the user fixes the bug in the C code. In case of (b), the user interactively solves the side conditions with a snippet of tactics, and then copies that snippet in the source code using the RefinedC's `rc::tactics` annotation. If a few lines of tactics are not adequate, the user can instead prove a lemma in a separate Coq file, and instruct RefinedC to apply the lemma using the `rc::lemmas` annotation. These annotations instruct the RefinedC frontend to include the manually added tactics and lemmas into the generated proof script. After that, the user invokes step (2) again, and iterates until Coq accepts all the generated proofs.

Verification example. We illustrate BFF's verification by walking through the verification of `set_valid_pte_leaf`. First, the user-level refined types `old @ pte_t`, `pa @ pte_t`, and `attr @ pte_t` in the specification of `set_valid_pte_leaf` (Fig. 3) are elaborated to the types t_{old}@bf_term$\langle\sigma_{\text{PTE}}, \texttt{u64}\rangle$, t_{pa}@bf_term$\langle\sigma_{\text{PTE}}, \texttt{u64}\rangle$, and t_{attr}@bf_term$\langle\sigma_{\text{PTE}}, \texttt{u64}\rangle$. The terms t_{old}, t_{pa} and t_{attr} are defined in terms of the record parameters `old`, `pa` and `attr`, respectively using a Coq function injected by our frontend. For example, the term t_{pa} is

$$\begin{aligned} t_{\text{pa}} = {} & \langle 0,1\rangle \mapsto \mathsf{data}(\texttt{pa.pte_valid}) :: \langle 1,1\rangle \mapsto \mathsf{data}(\texttt{pa.pte_type}) \\ & :: \langle 2,10\rangle \mapsto \mathsf{data}(\texttt{pa.pte_leaf_attr_lo}) :: \langle 12,36\rangle \mapsto \mathsf{data}(\texttt{pa.pte_addr}) \\ & :: \langle 51,13\rangle \mapsto \mathsf{data}(\texttt{pa.pte_leaf_attr_hi}) :: \mathsf{nil} \end{aligned}$$

The terms t_{old} and t_{attr} are similar (replace `pa` in the above with `old` and `attr`, respectively).

Next, BFF's typing rules are applied to type check all the bitwise operations in the function's body from Fig. 1. The expression `pa & PTE_ADDR_MASK` on line 11 is type-checked using the typing rule for `&` above: `pa` has the refined type t_{pa}@bf_term$\langle\sigma_{\text{PTE}}, \texttt{u64}\rangle$ from the elaboration, while `PTE_ADDR_MASK` has the type $t_{\text{addr_mask}}$@bf_term$\langle\sigma_{\text{PTE}}, \texttt{u64}\rangle$, where $t_{\text{addr_mask}} = \langle 12,36\rangle \mapsto \mathsf{data}(2^{36}-1) :: \mathsf{nil}$, so, by the typing rule for `&`, `(pa & PTE_ADDR_MASK)` has the refined type t^1_{new}@bf_term$\langle\sigma_{\text{PTE}}, \texttt{u64}\rangle$ where $t^1_{\text{new}} = t_{\text{pa}} \searrow t_{\text{addr_mask}} = \langle 12,36\rangle \mapsto \mathsf{data}(\texttt{pa.pte_addr}) :: \mathsf{nil}$.

On line 12, the expression `PTE_LEAF_ATTR_LO | PTE_LEAF_ATTR_HI` is typed using the rule for `|` above. Here, `PTE_LEAF_ATTR_LO` and `PTE_LEAF_ATTR_HI` have the refined types t_{LO}@bf_term$\langle\sigma_{\text{PTE}}, \texttt{u64}\rangle$ and t_{HI}@bf_term$\langle\sigma_{\text{PTE}}, \texttt{u64}\rangle$, where

$$t_{\text{LO}} = \langle 2,10\rangle \mapsto \mathsf{data}(2^{10}-1) :: \mathsf{nil} \qquad t_{\text{HI}} = \langle 51,13\rangle \mapsto \mathsf{data}(2^{13}-1) :: \mathsf{nil}$$

[3] Since BFF builds on RefinedC, the high-level workflow is the same as the one shown in Sammler et al. [2021, Figure 2].

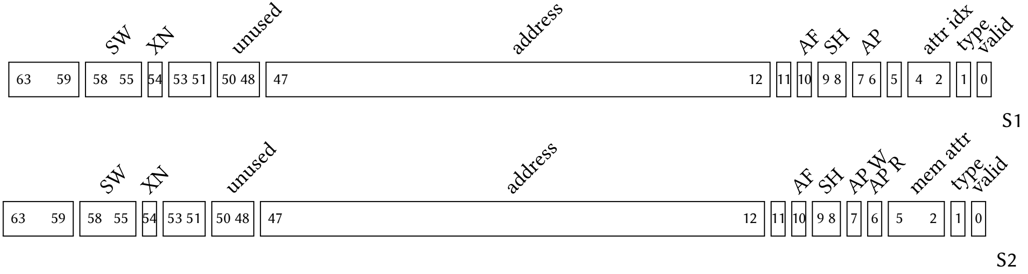

Fig. 4. Bitfields of stage 1 and stage 2 page table entries.

By the rule for typing `|`, (`PTE_LEAF_ATTR_LO | PTE_LEAF_ATTR_HI`) has the type t_{LOHI}@bf_term$\langle\sigma_{\text{PTE}},$ `u64`$\rangle$, where $t_{\text{LOHI}} = \langle 2, 10\rangle \mapsto \text{data}(2^{10} - 1) :: \langle 51, 13\rangle \mapsto \text{data}(2^{13} - 1) :: \text{nil}$, which is the bit mask for both the bitfields `pte_leaf_attr_lo` and `pte_leaf_attr_hi` together—exactly what the operation `PTE_LEAF_ATTR_LO | PTE_LEAF_ATTR_HI` creates logically.

The reader should notice a simple pattern emerging here: The typing rules symbolically compute *logical terms* of type `sbvd` corresponding to the *concrete SBVs* that C's bitwise operations in the program compute. Applying this pattern, after line 13, the type of `pte` is $t_{\text{pte_last}}$@bf_term$\langle\sigma_{\text{PTE}},$ `u64`$\rangle$, where

$$t_{\text{pte_last}} = \langle 0, 1\rangle \mapsto \text{data}(1) :: \langle 2, 10\rangle \mapsto \text{data}(\texttt{attr.pte_leaf_attr_lo}) \\ :: \langle 12, 36\rangle \mapsto \text{data}(\texttt{pa.pte_addr}) :: \langle 51, 13\rangle \mapsto \text{data}(\texttt{attr.pte_leaf_attr_hi}) :: \text{nil}$$

On the last line of the function, the output pointer `ptep` is overwritten by the value `pte`. This pointer write is type-checked using RefinedC's existing rules, and results in the assertion that `ptep` points to a C value of the type $t_{\text{pte_last}}$@bf_term$\langle\sigma_{\text{PTE}},$ `u64`$\rangle$. All that remains to show is that this type is equivalent to the type `{svl_pte pa attr} @ pte_t`, which the specification of the function requires `ptep` to point to at the end (clause `rc::ensures` on line 28 of Fig. 3). The elaboration of `{svl_pte pa attr} @ pte_t` is t_{expected}@bf_term$\langle\sigma_{\text{PTE}},$ `u64`$\rangle$ where

$$t_{\text{expected}} = \langle 0, 1\rangle \mapsto \text{data}(1) :: \langle 1, 1\rangle \mapsto \text{data}(0) :: \langle 2, 10\rangle \mapsto \text{data}(\texttt{attr.pte_leaf_attr_lo}) \\ :: \langle 12, 36\rangle \mapsto \text{data}(\texttt{pa.pte_addr}) :: \langle 51, 13\rangle \mapsto \text{data}(\texttt{attr.pte_leaf_attr_hi}) :: \text{nil}$$

The verification is completed by proving $t_{\text{pte_last}}$ and t_{expected} are equal at the type `sbvd`. Intuitively, this is true because the only difference between the two terms is that t_{expected} maps the bit range $\langle 1, 1\rangle$ to zero, while $t_{\text{pte_last}}$ omits this bit range. Since the default value of a bitfield is anyway zero, this difference is irrelevant. Formally, this proof relies on a simple decision procedure for such semantic equality of `sbvd`-typed terms implemented in Coq. We describe this procedure and its metaproperties in §4.

2.4 Bitfields with Nested Structures

In some situations, parts of an SBV type may have different bitfield layouts depending on the context. This is actually the case for the ARMv8 PTEs (type `pte_t`) that we have been using in our examples. The two fields `pte_leaf_attr_lo` (bit range [2 .. 11]) and `pte_leaf_attr_hi` (bit range [51 .. 63]), which we have been treating atomically so far, are actually not atomic: they each have *nested subfields* in *two possible layouts*, depending on whether the PTE is for stage 1 (S1) of ARMv8's

```
1  //@rc::bitfields Pte_staged (s : stage) as u64
2  //@  pte_valid        : bool[0]
3  //@  pte_type         : int[1]
4  //@  pte_leaf_attr_lo : nested[2..11]
5  //@    match s with S1 => Pte_S1_lo | S2 => Pte_S2_lo end
6  //@  pte_addr         : int[12..47]
7  //@  pte_leaf_attr_hi : nested[51..63]
8  //@    match s with S1 => Pte_S1_hi | S2 => Pte_S2_hi end
9  //@rc::end
10
11 //@rc::bitfields Pte_S1_lo 10 bits
12 //@  attr_lo_s1_attridx : int[0..2]
13 //@  attr_lo_s1_ap      : int[4..5]
14 //@  attr_lo_s1_sh      : int[6..7]
15 //@  attr_lo_s1_af      : bool[8]
16 //@rc::end
```

Fig. 5. Definition of `Pte_staged`.

address translation or stage 2 (S2). Fig. 4 shows the two possible layouts of a PTE, with the subfields fully exposed.

SBV specifications with variant nested subfields. A naive approach to dealing with these two variant layouts of PTEs would be to treat the two layouts as completely different types, *i.e.*, write two independent `rc::bitfields` specifications for S1 and S2 page table entries. However, this approach would also require us to write two different specifications and two different verifications even for functions like `set_valid_leaf_pte`, which actually do not examine the subfields where the two layouts really differ and, hence, are truly *parametric* in the PTE stage. This is not just pointlessly cumbersome for the programmer of `set_valid_leaf_pte`—it forces them to unnecessarily think about the two variants when they don't really need to.

Consequently, BFF takes a different approach: We support SBV specifications where some fields have explicitly nested subfields, and these nested subfields can have multiple layouts depending on an additional parameter (in our example, the parameter codes the stage S1 or S2). As an example, Fig. 5 shows the revised SBV specification for PTEs based on this idea.

This revised specification, called `Pte_staged` (line 1), is parameterized by `s`, which is of a new Coq type `stage`—an enumeration type containing two values `S1` and `S2` (we elide the straightfoward definition). The definition of `Pte_staged` defines the bitfield `pte_leaf_attr_lo` as *nested*: Depending on whether `s` is `S1` or `S2`, `pte_leaf_attr_lo` either has subfields adhering to the bitfield specification `Pte_S1_lo` or to the bitfield specification `Pte_S2_lo`. The bitfield specification `Pte_S1_lo` is shown on line 11; `Pte_S2_lo` is similarly defined and elided here. In a similar way, `Pte_staged` defines the bitfield `pte_leaf_attr_hi` as nested.

The frontend-generated record type for this `Pte_staged` specification uses Coq sum types to represent the possible variants.

```
Record Pte_staged :=
  { pte_valid : bool; pte_type : Z; pte_leaf_attr_lo : Pte_S1_lo + Pte_S2_lo;
    pte_addr : Z; pte_leaf_attr_hi : Pte_S1_hi + Pte_S2_hi }.
```

The corresponding refinement type, $r@\text{bitfield}\langle R, s\rangle$, also takes the additional parameter s, which specifies the stage and, hence, whether the record r (of type R) should have values tagged `inl` or

inr in the fields pte_leaf_attr_lo and pte_leaf_attr_hi. The semantic definition of $r@\text{bitfield}\langle R, s\rangle$ is empty (False in Coq) if the two inl/inr annotations in r do not match what s mandates.[4]

Function specifications. With this, we can specify our two example functions pte_valid and set_valid_leaf_pte parametrically in the stage s.

```
1  [[rc::parameters("s : stage", "pte : Pte_staged")]]
2  [[rc::args("pte @ bitfield<Pte_staged, s>")]]
3  [[rc::returns("{pte.(pte_valid)} @ builtin_boolean")]]
4  static bool pte_valid(pte_t pte) { /* code unchanged */ }
5
6  [[rc::parameters("p: loc", "s : stage", "old, pa, attr: Pte_staged")]]
7  [[rc::args("p @ &own<old @ bitfield<Pte_staged, s>>", "pa @ bitfield<Pte_staged, s>",
8            "attr @ bitfield<Pte_staged, s>")]]
9  [[rc::ensures("own p: {svl_pte pa attr} @ bitfield<Pte_staged, s>")]]
10 static void set_valid_leaf_pte(pte_t *ptep, u64 pa, pte_t attr) { /* code unchanged */ }
```

Note how the specifications of both functions are parameterized by the stage s. A client of these functions can therefore safely call these functions for PTEs of either stage.

The specifications of other functions that manipulate PTEs of a specific stage would use the concrete stage S1 or S2 in place of the parameter s.

Internal types and typing rules. Recall from §2.3 that BFF elaborates the bitfield types, which are refined by bitfield records, into lower-level $\text{bf_term}\langle\sigma, \alpha\rangle$ types, which are refined by terms of the sbvd type, because it is much easier to define operators like $\cup$ and $\searrow$ on sbvd as compared to the variously-typed records.

To handle nested subfields, we modify signatures σ and the terms of sbvd. Signatures σ, which were earlier lists of elements of the form $\text{atom}(\langle a, k\rangle)$ (a bitfield of length k at offset a), may now also contain a new kind of element, $\text{nested}(\langle a, k\rangle, \sigma')$, which denotes a nested bitfield of length k at offset a, whose nested subfields have the signature σ'.

Similarly, terms t of type sbvd, which were earlier lists of elements of the form $\langle a, k\rangle \mapsto \text{data}(v)$, may now also contain elements of the form $\langle a, k\rangle \mapsto \text{nested}(t)$, wherein t is a term logically representing a nested bitfield.

As an example, the term t_{pa}, which logically represents the value of the parameter pa at the beginning of the function set_valid_leaf_pte, would now be revised to

$$\begin{aligned} t_{\text{pa}} = \langle 0, 1\rangle &\mapsto \text{data}(\texttt{pa.pte_valid}) :: \langle 1, 1\rangle \mapsto \text{data}(\texttt{pa.pte_type}) :: \langle 2, 10\rangle \mapsto \text{nested}(t_{\text{lo}}) \\ &:: \langle 12, 36\rangle \mapsto \text{data}(\texttt{pa.pte_addr}) :: \langle 51, 13\rangle \mapsto \text{nested}(t_{\text{hi}}) :: \text{nil} \end{aligned}$$

where t_{lo} (resp. t_{hi}) is a term defined using the stage s and the value pa.pte_leaf_attr_lo (resp. pa.pte_leaf_attr_lo).

Verification. The verification of functions such as pte_valid and set_valid_leaf_pte, which are parametric in the variant selector (s in our examples), is not really affected in any significant way by the changes above. Of course, some fields in the logical terms, which were earlier of the form $\text{data}(v)$, now change to the form $\text{nested}(t)$, but because the functions do not really examine the nested subfields, this difference is only cosmetic.

For typechecking functions that actually examine nested subfields, the only change is in the definitions of the sbvd term operators like $\cup$ and $\searrow$, which perform logical bitfield transformations on terms. These operators must now merge or extract nested subfields as well. We provide the definitions of these operators (with nested subfields) in §4.1.

[4] An alternative approach would be to define the record type Pte_staged dependent on s. However, we found it very difficult to work with this dependent type in Coq.

op	ty_1	ty_2	P	ty'	rule name
		t_1@bf_term$\langle\sigma,\alpha\rangle$	t_2@bf_term$\langle\sigma,\alpha\rangle$	t_1 ## t_2	$(t_1 \cup t_2)$@bf_term$\langle\sigma,\alpha\rangle$	Ty-merge
|	t_1@bf_term$\langle\sigma,\alpha\rangle$	t_2@bf_term$\langle\sigma,\alpha\rangle$	is_mask(t_2)	$(t_1 \cup t_2)$@bf_term$\langle\sigma,\alpha\rangle$	Ty-set
	t_1@bf_term$\langle\sigma,\alpha\rangle$	t_2@bf_term$\langle\sigma,\alpha\rangle$	is_mask(t_i)	$(t_1 \cup t_2)$@bf_term$\langle\sigma,\alpha\rangle$	Ty-concat
&	t_1@bf_term$\langle\sigma,\alpha\rangle$	t_2@bf_term$\langle\sigma,\alpha\rangle$	is_mask(t_2)	$(t_1 \searrow t_2)$@bf_term$\langle\sigma,\alpha\rangle$	Ty-mask
~	t@bf_term$\langle\sigma,\alpha\rangle$	-	is_mask(t)	t@bfneg$\langle\sigma,\alpha\rangle$	Ty-not
&	t_1@bf_term$\langle\sigma,\alpha\rangle$	t_2@bfneg$\langle\sigma,\alpha\rangle$	-	$(t_1 \searrow_{\sim} t_2)$@bf_term$\langle\sigma,\alpha\rangle$	Ty-clear
$\gg$	$(r \mapsto$ data(n) :: nil) @bf_term$\langle\sigma,\alpha\rangle$	k@int$\langle\alpha\rangle$	$k = r$.offset	n@int$\langle\alpha\rangle$	Ty-read
$\ll$	n@int$\langle\alpha\rangle$	k@int$\langle\alpha\rangle$	$k = r$.offset	$(r \mapsto$ data(n) :: nil) @bf_term$\langle\sigma,\alpha\rangle$	Ty-load

Fig. 6. Typing rules for bitfield manipulations by bitwise operations.

3 TYPING BITFIELD MANIPULATIONS

In this section, we describe common patterns of use of C's bitwise operators for bitfield manipulation, and the BFF typing rules that allow reasoning about these patterns. Fig. 6 shows all the typing rules of BFF in a compact form. Each line in this table should be read as the definition of a typing rule of the following form (the rule for the unary "~" operator omits e_2 and ty_2):[5]

$$\frac{e_1 \rhd_e ty_1 \qquad e_2 \rhd_e ty_2 \qquad P}{(e_1\ op\ e_2) \rhd_e ty'}$$

In what follows below, we go over the different patterns of bitfield manipulation and explain how they are captured by the BFF typing rules. (Note that we briefly introduced the rules Ty-merge and Ty-mask in §2.3.)

Merging and setting bitfields via bitwise OR. The bitwise OR operator, |, is commonly used in three different ways for bitfield manipulation, all of which occur in the `set_valid_leaf_pte` function (Fig. 1).

The first use, illustrated by the expression `pte | (attr & (PTE_LEAF_ATTR_LO | PTE_LEAF_ATTR_HI))` on line 12, is to *merge* (take the union of) the bitfields of two SBVs. In the example, the bitfields `pte_leaf_attr_lo` and `pte_leaf_attr_hi` from `attr` are merged with the `pte_addr` bitfield from `pte`. This use of `|` is type-checked using the typing rule Ty-merge, which relies on the `sbvd` merging operator $t_1 \cup t_2$ introduced in §2.3. The operator $t_1 \cup t_2$ requires the bitfields of t_1 and t_2 to be disjoint (encoded by the side condition t_1 ## t_2). While we could, in principle, get rid of this disjointness condition, we prefer to keep it because the condition excludes bitwise-or of values in the *same* bitfield, which is rarely used in SBVs manipulation, and is almost always indicative of a bug. If we were to get rid of the disjointness condition in the typing rule, the typing rule would still apply in this likely-buggy scenario, and verification would later fail deep inside RefinedC's existing automation (which cannot handle field-level bitwise-or operations), resulting in a very hard-to-decode error message for the programmer. In contrast, with our design, verification fails very early—right at the typing rule—which results in a much more comprehensible error message for the programmer.[6]

[5]To prevent exponential blowup for overlapping rules, the actual encoding of the typing rules in RefinedC is more involved. Concretely, RefinedC first infers the types ty_1 and ty_2 and only then selects which rule to apply.

[6]The rare cases where the programmer really does intend a bitwise-or of the values in the same bitfield can be verified in RefinedC via manual proof.

The second use—which is actually an exception to this disjointness check—is to apply `|` to an SBV and a *mask* to *set* bitfields in the SBV. For example, the expression `pte | PTE_VALID` (line 13) sets the `pte_valid` bitfield of `pte`. This use case is type-checked using the rule Ty-set that applies when the second operand of `|` is a mask. (We elide a symmetric rule which applies when the first operand is a mask.)

The third use, a special instance of the second one, is to apply the `|` operator to two masks to create a mask spanning the union of the bitfields of the two masks. An example is the expression `PTE_LEAF_ATTR_LO | PTE_LEAF_ATTR_HI` on line 12. This third use-case is typed by the rule Ty-concat.

Extracting/Masking bitfields via bitwise AND. There are two common ways of using the bitwise AND operator, &, for bitfield manipulation: to *extract or mask* fields in an SBV and, in conjunction with bitwise negation ~, to *clear or unset* bitfields. We describe the extract use-case here and the clear use-case below.

To *extract* a bitfield from an SBV, the & operator is applied to the SBV and a mask for the bitfield. An example of this is the expression `pte & PTE_VALID` in the function `pte_valid` on line 8 of Fig. 1, which results in an SBV that only contains the `pte_valid` bitfield (all other bitfields are set to zero). This use of the & operator is type-checked using the typing rule Ty-mask, which relies on the extraction operation $t_1 \searrow t_2$ on `sbvd` introduced in §2.3. The rule applies only when t_2 is a mask (condition $P = \text{is_mask}(t_2)$) as, otherwise, the & operation ends up taking the bitwise AND of the value in the same bitfield, which is rarely what the programmer intends when manipulating SBVs. (We elide a symmetric rule which applies when t_1 is a mask.)

Clearing bitfields via bitwise NOT and AND. A second use of & is to *clear* bitfields. For this, the SBV is &-ed with the bitwise negation ~ of the mask spanning the bitfields to be cleared. An example is the expression `pte & ~PTE_VALID`, which clears the `pte_valid` bitfield of the SBV `pte`.

This combined use of & and ~ is type-checked in two steps. First, the *negation of the mask* is type-checked using the rule Ty-not, which says that if e is a mask SBV of type $t@\text{bf_term}\langle\sigma, \alpha\rangle$, then $\sim e$ is an SBV of a new BFF type $t@\text{bfneg}\langle\sigma, \alpha\rangle$, which is the singleton type of the *negation* of t. (The well-formedness of this type also requires that t describe a mask.)

Second, the application of & to the SBV and the now negated mask is type-checked using the rule Ty-clear. This rule uses the term operator $t_1 \searrow_{\sim} t_2$ (called the clearing operator), which bitwise ANDs t_1 with the bitwise negation of t_2 (when t_2 is a mask over some fields, this has the effect of clearing out those fields selectively from t_1 while retaining the rest).

Reading and loading bitfield values via shifts. So far, we have described how to type-check programming patterns that combine or convert SBVs to other SBVs. However, there are two common programming patterns that convert between SBVs and *integers*.

The first of these patterns, which we call *bitfield read*, reads the data value of a specific bitfield in an SBV as an integer. This pattern is implemented using the bit-right-shift operator, $\gg$. Specifically, if e_1 is an SBV containing a *single* non-zero bitfield at offset e_2, then $e_1 \gg e_2$ is an integer containing just the value of the bitfield. This use of the $\gg$ operator to read a bitfield's value as an integer is typed using the rule Ty-read. The rule applies when $e_1 : (r \mapsto \text{data}(n) :: \text{nil})@\text{bf_term}\langle\sigma, \alpha\rangle$, *i.e.*, e_1 is the concrete representation of an `sbvd` with only one bitfield set, and $e_2 : r.\text{offset}@\text{int}\langle\alpha\rangle$, *i.e.*, e_2 is exactly the offset of that one bitfield. The rule's conclusion says that, under these conditions, $e_1 \gg e_2$ has the type $n@\text{int}\langle\alpha\rangle$, *i.e.*, $e_1 \gg e_2$ will evaluate to the integer contained in the bitfield.

An example of this use of $\gg$ to read a bitfield is the following standard Linux macro for accessing the bitfield of the SBV `reg` specified by the mask `mask`. Here, `__bf_shf(mask)` computes the index of the lowest non-zero bit of `mask`, *i.e.*, the offset of the bitfield represented by `mask`.

```
#define FIELD_GET(mask, reg) (((reg) & (mask)) >> __bf_shf(mask))
```

Term $t \in$ `sbvd`	::=	nil	(empty list)
	|	$r \mapsto v :: t$	(nonempty list)
Data value $v \in$ `sbvd_val`	::=	data(n)	(integer value)
	|	mask	(mask value)
	|	nested(t)	(nested value)
Range r	:=	$\langle a, k \rangle$	(offset, width)

Fig. 7. Syntax of SBVD terms.

The second pattern, which we call a *bitfield load*, does the opposite of a bitfield read: It takes an integer value e_1 and the offset e_2 of some bitfield, and uses the bit-left-shift operator $e_1 \ll e_2$ to yield an SBV that contains e_1 in the bitfield and 0s in all other bitfields. This use of the $\ll$ operator is typed using the using the rule Ty-load, which should be self-explanatory.

An example of this use of $\ll$ to load a bitfield is the following expression that appears in a pKVM function: `pte | (type << __bf_shf(PTE_TYPE_MASK))`. The expression first uses $\ll$ to load the integer value `type` to the `pte_type` bitfield, and then uses the `|` operator to merge this resulting SBV with the SBV `pte`.

Summary and limitations. BFF's typing rules (Fig. 6) cover all common bitfield manipulations on SBVs: merging the bitfields in two SBVs or two masks, setting bitfields in SBVs using masks, clearing bitfields in SBVs using negated masks, masking out specific bitfields from SBVs (extraction), reading the value in a specific bitfield as an integer, and loading an integer value to specific a bitfield.

Given BFF's focus on bitfield manipulation, verification of arbitrary bitwise operations (that go beyond bitfield manipulation) is not in the scope of BFF, and thus our typing rules cannot handle other usages of bitwise operators such as optimized integer multiplication and division via bit shifting. Among bitfield operations, the main limitations of BFF that we are aware of are missing support for (a) dynamically determined layouts where the layout is determined by the values of other fields, and (b) the rare situation where a logical value is stored non-contiguously by splitting it across two or more bitfields (*e.g.*, a 32-bit integer is stored by placing bits 0-17 at offset 0, and bits 18-31 at offset 20).

4 THE TYPE OF SBV DESCRIPTORS, ITS TERMS, AND ITS OPERATIONS

In §2.3, we introduced the Coq type `sbvd`, whose inhabitants represent SBVs logically and refine the BFF/RefinedC type bf_term$\langle \sigma, \alpha \rangle$. We also introduced the binary operators $\cup$, $\searrow$ and $\searrow_{\sim}$ on `sbvd` in §2.3 and §3. In this section, we formally define the syntax of the terms of `sbvd`, their *sorting* to signatures σ, and the binary operators. We also define semantic equality on `sbvd`, and a simple sound and complete algorithm for checking this equality.

Terms of **`sbvd`*.*** Technically, `sbvd` is a Coq datatype, defined mutually inductively with another datatype, `sbvd_val`, which describes the *contents* of individual bitfields. The two datatypes are mutually inductive because the content of a nested bitfield is itself an `sbvd`. Elements of `sbvd`, called *terms* (denoted t), are defined mututally inductively with the elements of `sbvd_val`, called *data values* (denoted v). The syntax of both is shown in Fig. 7. A term t is essentially a list of mappings from ranges r to data values v. A range $r = \langle a, k \rangle$ represents a bitfield of length k at offset a. Values v represent contents of bitfields. A data value v is one of the following: (1) data(n), which is the

$$
\begin{aligned}
[\![\mathsf{nil}]\!] &:= 0 \\
[\![r \mapsto v :: t]\!] &:= ([\![v]\!]_r \ll r.\text{offset}) \mid [\![t]\!]
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathsf{data}(n)]\!]_r &:= n \\
[\![\mathsf{mask}]\!]_r &:= 2^{r.\text{width}} - 1 \\
[\![\mathsf{nested}(t)]\!]_r &:= [\![t]\!]
\end{aligned}
$$

Fig. 8. Denotational semantics.

$$
\textsc{Sort-nil}\quad \vdash \mathsf{nil} : \sigma
\qquad
\textsc{Sort-cons}\quad \frac{\vdash t : \sigma \qquad \vdash_{\text{val}} v : \sigma, r \qquad r \prec_{\text{hd}} \mathsf{ranges}(t)}{\vdash r \mapsto v :: t : \sigma}
$$

$$
\textsc{Sort-val}\quad \frac{\mathsf{atom}(r) \in \sigma \qquad 0 \leq n < 2^{r.\text{width}}}{\vdash_{\text{val}} \mathsf{data}(n) : \sigma, r}
\qquad
\textsc{Sort-mask}\quad \frac{r \in \sigma}{\vdash_{\text{val}} \mathsf{mask} : \sigma, r}
\qquad
\textsc{Sort-nested}\quad \frac{\mathsf{nested}(r, \sigma_r) \in \sigma \qquad \vdash t_r : \sigma_r}{\vdash_{\text{val}} \mathsf{nested}(t_r) : \sigma, r}
$$

Fig. 9. Sorting of terms and data values.

integer n in binary, (2) mask which stands for all 1s, and (3) $\mathsf{nested}(t)$, which are nested subfields described by t.

We define a *denotation* of terms and data values into (Coq) integers, written $[\![t]\!]$ and $[\![v]\!]_r$, respectively, and shown in Fig. 8. Intuitively, $[\![t]\!]$ is the integer obtained by reading the SBV represented by t as an integer coded in binary, and $[\![v]\!]_r$ is the integer obtained by reading the contents represented by v in a bitfield of width r.

Sorting. Not all terms of `sbvd` meaningfully represent SBVs. For example, the term $\langle 0, 2\rangle \mapsto \mathsf{data}(4) :: \mathsf{nil}$ is not meaningful because it claims that a bitfield of width 2 contains the integer 4, but 4's binary representation (100) requires 3 bits. Another example is any term whose list has overlapping bit ranges, *e.g.*, $\langle 0, 2\rangle \mapsto v_1 :: \langle 1, 2\rangle \mapsto v_2 :: \mathsf{nil}$, whose two bitfields overlap each other.

To eliminate such meaningless terms, we defining a *sorting* relation, which assigns a signature (σ from §2.3) to a term. Recall that a signature σ is a layout specification. It is a list of bit range declarations of the forms $\mathsf{atom}(\langle a, k\rangle)$ and $\mathsf{nested}(\langle a, k\rangle, \sigma')$ meaning, respectively, that there is an *atomic* bitfield of width k at offset a and that there is a *nested* bitfield of width k at offset a whose subfields have signature σ. We define sorting so that any term sorted by σ cannot have the problems mentioned in the previous paragraph and, additionally, such a term can mention only those ranges that occur in σ.

The sorting relation for terms, written $\vdash t : \sigma$, and that for data values, written $\vdash_{\text{val}} v : \sigma, r$, are defined by mutual induction in Fig. 9. The condition $r \prec_{\text{hd}} \mathsf{ranges}(t)$ in rule Sort-cons means that the bit positions in range r are strictly less than bit positions in any range occurring in t. This check enforces that: (a) The bitfields in a well-sorted term do not overlap, and (b) The bitfields in a well-sorted term are sorted ascending by their offsets. We exploit the second property in defining binary operators on `sbvd` and in our algorithm for semantic equality checking of terms.

In the three rules for $\vdash_{\text{val}} v : \sigma, r$, the checks $\text{atom}(r) \in \sigma$, $\text{nested}(r, \sigma_r) \in \sigma$, and $r \in \sigma$ (shorthand for the disjunction of the first two checks) ensure that a term does not mention any bit ranges that do not appear in the signature. They also ensure that if a signature says that a bit range is an atomic bitfield, then the ascribed term does not contain a nested data value at that bit range (and vice-versa). Finally, the condition $0 \leq n < 2^{r.\text{width}}$ in the rule Sort-val ensures that a data value in an atomic bitfield can actually be represented in binary within the bitfield's width.

Note that $\vdash t : \sigma$ does not imply that every range mentioned in σ also appears in t. Any ranges omitted from t are automatically treated as mapped to 0 by the denotational semantics of t.

4.1 Operators and Predicates on Terms of sbvd

The typing rules of §3 use operations ($t_1 \cup t_2$, $t_1 \searrow t_2$, $t_1 \searrow_{\sim} t_2$) and predicates ($\text{is_mask}(t)$, $t_1 \mathbin{\#\#} t_2$) on terms of sbvd. We describe how these operations and predicates are defined. In the interest of not repeating similar-looking definitions, we omit some definitions. These can be found in our technical appendix[7].

Predicates. The predicate $\text{is_mask}(t)$ checks that t is a mask, *i.e.*, every range in it is mapped to mask. The inductive rules defining this predicate and an identically named predicate on data values are shown below.

$$\frac{}{\text{is_mask}(\text{mask})} \qquad \frac{\text{is_mask}(t)}{\text{is_mask}(\text{nested}(t))} \qquad \frac{}{\text{is_mask}(\text{nil})} \qquad \frac{\text{is_mask}(v) \quad \text{is_mask}(t)}{\text{is_mask}(r \mapsto v :: t)}$$

Our second predicate $t_1 \mathbin{\#\#} t_2$ is defined only when t_1 and t_2 have the same sort. The predicate checks that any ranges mentioned in both t_1 and t_2 are mapped to 0 in at least one of them. This predicate is defined by simultaneous recursion on the structures of t_1 and t_2 in a manner similar to the definition of $t_1 \cup t_2$ below, so we elide the details.

Operators. We describe the definition and properties of the operator $t_1 \cup t_2$ here. The other two operators—$\searrow$ and $\searrow_{\sim}$—have similar definitions, so we defer these to our technical appendix.

The operator $t_1 \cup t_2$ is defined only when t_1 and t_2 are sorted to the same signature and either $t_1 \mathbin{\#\#} t_2$ or at least one of t_1 and t_2 is a mask (as determined by the predicate $\text{is_mask}(t)$). This condition holds whenever the operator is used in the typing rules (§3, Fig. 6). $t_1 \cup t_2$ merges the terms t_1 and t_2, to logically simulate a bitwise `|` operator on the concrete representations of t_1 and t_2. The definition, shown below, relies on the fact that well-sortedness implies that the bitfields (ranges) in t_1 and t_2 are sorted ascending by offset. Like the "merge" operation of the classic merge sort algorithm, $t_1 \cup t_2$ is simultaneously recursive in the list structures of t_1 and t_2, and analyzes the heads of t_1 and t_2 together.

$$\text{nil} \cup t = t \tag{4}$$

$$t \cup \text{nil} = t \tag{5}$$

$$\begin{aligned} r_1 \mapsto v_1 :: t_1 \cup r_2 \mapsto v_2 :: t_2 = {} & \textbf{if } r_1 = r_2 \textbf{ then } r \mapsto (v_1 \cup^{\text{v}} v_2) :: (t_1 \cup t_2) \\ & \textbf{else if } r_1 \prec r_2 \textbf{ then } r_1 \mapsto v_1 :: (t_1 \cup r_2 \mapsto v_2 :: t_2) \\ & \textbf{else } r_2 \mapsto v_2 :: (r_1 \mapsto v_1 :: t_1 \cup t_2) \end{aligned} \tag{6}$$

$$v_1 \cup^{\text{v}} \text{mask} = \text{mask} \tag{7}$$

$$\text{mask} \cup^{\text{v}} v_2 = \text{mask} \tag{8}$$

$$\text{nested}(t_r) \cup^{\text{v}} \text{nested}(m_r) = \text{nested}(t_r \cup m_r) \tag{9}$$

[7] https://plv.mpi-sws.org/refinedc/bff/bff-appendix.pdf

The interesting clause of the definition is (6). If a range is mentioned only in t_1, then we copy that range's data value from t_1 (and similarly for t_2). Otherwise, if both t_1 and t_2 mention a range r, then the data values for that range are merged using the auxiliary operation $v_1 \cup^{\mathsf{v}} v_2$ on data values. Note that $v_1 \cup^{\mathsf{v}} v_2$ does not need to handle the case of overlapping values as this case is ruled out by the precondition of $t_1 \cup t_2$. This makes it easy to see that $t_1 \cup t_2$ never introduces bitwise operations.

The following two theorems state that on well-sorted terms that satisfy the preconditions of $\cup$, the operation preserves sorts, and that, *semantically*, $\cup$ computes the bitwise | operation.

Theorem 4.1 (Sort preservation by $\cup$). *Suppose* $\vdash t_1 : \sigma$ *and* $\vdash t_2 : \sigma$. *If either* $\mathsf{is_mask}(t_2)$ *or* $t_1 \mathbin{\#\#} t_2$, *then* $\vdash t_1 \cup t_2 : \sigma$.

Theorem 4.2 (Semantic correctness of $\cup$). *Suppose* $\vdash t_1 : \sigma$ *and* $\vdash t_2 : \sigma$. *If either* $\mathsf{is_mask}(t_2)$ *or* $t_1 \mathbin{\#\#} t_2$, *then* $[\![t_1 \cup t_2]\!] = [\![t_1]\!] \mid [\![t_2]\!]$, *where* | *is the bitwise-or operator on Coq integers.*

The definitions and properties of the operators $\searrow$ and $\searrow_{\sim}$ follow the same pattern. The preconditions for applying these operators are different but are also satisfied wherever they are used in the typing rules. The definitions of the operators use the same recursive structure as the definition of $\cup$ above. Both operators also preserve sorts and satisfy appropriate semantic correctness criteria. Details are in the appendix.

4.2 Checking Semantic Equality of Terms

Recall from §2.3 the verification of the function `set_valid_leaf_pte`, which required proving that two terms, $t_{\text{pte_last}}$ and t_{expected}, are semantically equal. We now describe a sound and complete decision procedure that BFF uses to check such semantic equality.

To start, two terms t_1, t_2 are semantically equal if $[\![t_1]\!] = [\![t_2]\!]$. Semantic equality testing is meaningful only on terms of the same sort. The following function $\Phi_{\text{eq}}(t_1, t_2)$ checks semantic equality of terms t_1 and t_2 (of the same sort) algorithmically.

$$\Phi_{\text{eq}}(\mathsf{nil}, t) := \Phi_{\text{zero}}(t) \tag{10}$$

$$\Phi_{\text{eq}}(t, \mathsf{nil}) := \Phi_{\text{zero}}(t) \tag{11}$$

$$\begin{aligned}\Phi_{\text{eq}}(r_1 \mapsto v_1 :: t_1, r_2 \mapsto v_2 :: t_2) := \;& \textbf{if } r_1 = r_2 \textbf{ then } \Phi^{\mathsf{v}}_{\text{eq}}(v_1, v_2, r_1) \wedge \Phi_{\text{eq}}(t_1, t_2) \\ & \textbf{else if } r_1 \prec r_2 \textbf{ then } \Phi^{\mathsf{v}}_{\text{zero}}(v_1, r_1) \wedge \Phi_{\text{eq}}(t_1, r_2 \mapsto v_2 :: t_2) \\ & \textbf{else } \Phi^{\mathsf{v}}_{\text{zero}}(v_2, r_2) \wedge \Phi_{\text{eq}}(r_1 \mapsto v_1 :: t_1, t_2)\end{aligned} \tag{12}$$

$$\tag{13}$$

$$\begin{aligned}\Phi^{\mathsf{v}}_{\text{eq}}(v_1, v_2, r) := \;& \textbf{if } v_1 = \mathsf{nested}(t_1) \text{ and } v_2 = \mathsf{nested}(t_2) \textbf{ then } \Phi_{\text{eq}}(t_1, t_2) \\ & \textbf{else if } v_1 = \mathsf{mask} \text{ and } v_2 = \mathsf{mask} \textbf{ then } \text{True} \\ & \textbf{else } [\![v_1]\!]_r = [\![v_2]\!]_r\end{aligned} \tag{14}$$

$$\tag{15}$$

$$\Phi_{\text{zero}}(\mathsf{nil}) := \text{True} \tag{16}$$

$$\Phi_{\text{zero}}(r \mapsto v :: t) := \Phi^{\mathsf{v}}_{\text{zero}}(v, r) \wedge \Phi_{\text{zero}}(t) \tag{17}$$

$$\tag{18}$$

$$\Phi^{\mathsf{v}}_{\text{zero}}(v, r) := \textbf{if } v = \mathsf{nested}(t) \textbf{ then } \Phi_{\text{zero}}(t) \textbf{ else } [\![v]\!]_r = 0 \tag{19}$$

The first two clauses of $\Phi_{\text{eq}}(t_1, t_2)$ (10, 11) say that if one of the two terms is nil, then the other term must be semantically equal to 0. Semantic equality to 0 is checked algorithmically by the auxiliary function $\Phi_{\text{zero}}(t)$, which is also defined above.

The third clause of $\Phi_{eq}(t_1, t_2)$ (12) says that if a range appears in both terms, then the corresponding data values must be equal, but if a range appears in only one of the two terms, then the corresponding data value must be 0 semantically.

The function $\Phi_{eq}(t_1, t_2)$ is sound and complete for semantic equality.

THEOREM 4.3 (SOUNDNESS AND COMPLETENESS OF SEMANTIC EQUALITY CHECKING). *Suppose* $\vdash t_1 : \sigma$ *and* $\vdash t_2 : \sigma$. *Then,* $[\![t_1]\!] = [\![t_2]\!]$ *if and only if* $\Phi_{eq}(t_1, t_2)$ *holds.*

5 SEMANTIC INTERPRETATION

Now we have all the pieces together to present how the bf_term and bitfield types are defined semantically.

An expression of type $t@\text{bf_term}\langle\sigma, \alpha\rangle$ is just an integer $[\![t]\!]@\text{int}\langle\alpha\rangle$ with the additional constraint that t has signature σ (and σ is within the bounds of α, denoted by $\text{compatible}(\sigma, \alpha)$):

$$t@\text{bf_term}\langle\sigma, \alpha\rangle := [\![t]\!]@\text{int}\langle\alpha\rangle \;\&\; \vdash t : \sigma \;\&\; \text{compatible}(\sigma, \alpha)$$

This definition uses the subset type $ty \,\&\, P$ that restricts the type ty with the constraint P.

The semantic definition for type $r@\text{bitfield}\langle R, x\rangle$ relies on information generated automatically from the `rc::bitfields` annotation:[8] (1) a function $\text{term}_R(r)$ that converts r into a term based on the fields in the annotation; (2) a function $\sigma_R(x)$ that provides the signature of the converted term for parameter x; (3) the integer size α_R of the underlying integer that r represents (from the `as`-clause); (4) a predicate $\text{nestedwf}_R(r, x)$ enforcing that the representation of r corresponds to the parameter x. The definition of bitfield uses this information to desugar to bf_term with the additional constraint nestedwf:

$$r@\text{bitfield}\langle R, x\rangle := \text{term}_R(r)@\text{bf_term}\langle\sigma_R(x), \alpha_R\rangle \;\&\; \text{nestedwf}_R(r, x)$$

6 IMPLEMENTATION IN REFINEDC

We implemented BFF using RefinedC, an extensible framework for automatic and foundational verification of C code. RefinedC handles the standard features of C like pointers, structures, arrays, and the many different kinds of control-flow. Thus, the main effort of our present work is to add support for bitfield-manipulating programs. RefinedC supports such extensions by defining new types and typing rules as discussed below. Concretely, our extension to RefinedC consists of three parts: frontend support for bitfield declarations (extending RefinedC's frontend), bitfield types with typing rules (extending RefinedC's type system), and the formalization of the meta-theories on `sbvd` terms.

First, we extended the RefinedC frontend with support for parsing bitfield declarations like the annotations on lines 1-7 in Fig. 3. Based on them, the frontend generates the record type definition, together with the information attached to this type as mentioned in §5.

Second, we added the types and typing rules presented in §3. Each typing rule is formulated as a lemma where the lemma statement corresponds to the rule and the lemma proof to the soundness proof of the rule. These typing rules are then added to the RefinedC type system via Coq's typeclass mechanism. Most of the typing rules are listed in Fig. 6. There is also a conversion rule that desugars bitfield to bf_term using the translation generated by the frontend. Additionally, there is a rule that allows casting $\text{bf_term}\langle\sigma, \alpha\rangle$ from the underlying integer type α to a different integer type β if both α and β are compatible with σ. Such a cast for example occurs when the `GENMASK` macro, which generates a `u64` integer, is used for a bitfield of shorter width (*e.g.*, `u32`). Also RefinedC has the limitation that it does not allow customizing how expressions on integer constants are type-checked (they always receive the `int` type). Thus, the BFF implementation reconstructs the corresponding

[8]Technically, this information is contained in the `BitfieldDesc` type class mentioned in §2.2.

Codebase	# Func.	All LoC	Func. LoC	# of bitfield manipulations							
				$\vert_1$	$\vert_2$	$\vert_3$	$\&_1$	$\sim$	$\&_2$	$\gg$	$\ll$
#1 `pgtable`	7	144	83	8	6	1	13	1	1	1	6
#2 `x86_pgtable`	30	190	120	9	0	3	10	9	9	0	0
#3 `tcp_input`	6	76	33	2	0	0	4	4	4	0	0
#4 `mt7601u`	4	792	115	13	3	1	9	0	0	7	3
Total	47	1202	351	23	9	5	35	5	5	8	9

Fig. 10. Statistical overview of the functions we considered in four codebases. LoC: lines of code. For bitfield manipulations, subscripts are attached to tell apart the multiple usages of one operator: "$\&_1$" for masking field(s), "$\&_2$" for clearing field(s), "$|_1$" for setting field(s), "$|_2$" for merging fields, and "$|_3$" for concatenating masks.

`sbvd` term from a constant integer expression lazily: the reconstruction happens only when the result of the constant expression is used in one of BFF's typing rules.

Finally, we formalized the definitions, operations, and meta-theory for `sbvd` terms presented in §4 in Coq. The operations were implemented as Gallina functions so that they can be efficiently executed in Coq. In addition to the sorting rules (Fig. 9) we also defined a recursive function that checks if an input term is well-sorted under a given signature. To automate the proofs of lemmas (like Theorem 4.2) related to bitwise operations, we developed a tactic that proves equality of two Coq integer expressions by proving that all their bits are equal (a.k.a. bit-blasting).

7 CASE STUDIES

To measure the expressiveness and automation of the proposed BFF approach, we studied and verified the full functional correctness of 47 functions that use bitwise operators to manipulate bitfields, selected from four codebases in the Linux kernel:

- (#1 `pgtable`) 7 functions that manipulate pKVM page table entries (the example functions listed in Fig. 1 are from this codebase);
- (#2 `x86_pgtable`) 30 short functions that test, set, or clear the access flags of x86 page table entries;
- (#3 `tcp_input`) 6 functions that control the ECN status bits in a TCP socket;
- (#4 `mt7601u`) 4 functions that handle the status of data transmission and rate control for the mt7601u (a Wi-Fi chip) Linux driver.

These functions usually serve as helpers for other functionalities; for example, the 30 functions we studied in `x86_pgtable` are useful in implementing virtual memory management.

Fig. 10 provides a summary of relevant characteristics of the functions we considered. We counted lines of code (LoC)[9] in two criteria: *Func. LoC* for the verified functions only, and *All LoC* for everything including bit mask macros, type aliases, and structs that do not need to be verified. Then, we manually collected the number of bitfield manipulations in the verified functions—including the bitfield manipulations occurring in an inlined function, once for each place where it is inlined—by their usages as listed in Fig. 6. Note that we do not provide specifications for inlined functions: their bodies will be expanded during verification. However, for non-inlined calls, compositional verification applies: if the pre-condition of the callee is met, then the post-condition will be ensured.

[9]Excluding comments and blanks, counted by an open-source Unix tool `tokei` [The Tokei Team 2022].

	Bitfield LoA	Struct LoA	Func. LoA	Other LoA	All LoA	$\frac{\text{Func. LoA}}{\text{Func. LoC}}$	$\frac{\text{All LoA}}{\text{All LoC}}$
#1	35	10	50	24	119	0.60	0.83
#2	17	2	97	6	122	0.81	0.64
#3	13	19	28	0	60	0.85	0.79
#4	69	44	84	12	209	0.73	0.26
Total	134	75	259	42	510	0.74	0.42

Fig. 11. Statistics on lines of annotations (LoA) added for BFF verification.

	# of typing rule applications										Side conditions		Time (s)
	$\vert_1$	$\vert_2$	$\vert_3$	$\&_1$	$\sim$	$\&_2$	$\gg$	$\ll$	cast	convert	manual	total	
#1	60	6	2	33	1	1	1	29	0	17	0	75	85
#2	9	0	3	10	9	9	0	0	0	12	0	17	95
#3	2	0	0	4	6	6	0	0	20	21	0	0	22
#4	42	10	2	39	0	0	10	18	143	56	3	63	209
Total	113	16	7	86	16	16	11	47	163	106	3	155	411

Fig. 12. Statistical metrics of the BFF verification process.

7.1 Formal Specification

Next, we evaluate the number of annotations required for verifying the functions using BFF. We categorize the annotations into four categories:

- Bitfield declarations: `rc::bitfields` blocks that specify the bitfield layout of bit vectors.
- Struct refinement: `rc::refined_by` and `rc::field` annotations that provide refinement types for a C struct and its members. These annotations are necessary for the examples that store bitfields in structures.
- Functional correctness: annotations attached to functions, including `rc::parameters` and `rc::args` for binding arguments, `rc::returns` for specifying the expected return value, and `rc::requires` (resp. `rc::ensures`) for describing pre-conditions (resp. post-conditions).
- Other annotations: extra auxiliary Coq definitions (*e.g.*, constants, a record that represents a C struct, or a helper function/predicate for pre-/post-conditions) used by the annotations mentioned above, typically enclosed in an `rc::inlined_prelude` block.

The lines of annotations (LoA) added to the studied codebases are presented in Fig. 11, according to the above four categories, and *All LoA* gives their sums. We measure the annotation burden by the ratio of LoA to LoC, provided in the last two columns of the table. The ratios are similar to what has been reported for non-bitfield programs in prior work on RefinedC [Sammler et al. 2021; Lepigre et al. 2022].

7.2 Automated Verification

Verification using RefinedC proceeds in two steps: In the first step, the automatic type checker applies typing rules like the rules shown in §3. These typing rules can generate pure side conditions that are discharged in a second step. First, RefinedC tries to automatically solve the side conditions with a builtin solver. The remaining side conditions are given to the user to solve manually.

Fig. 12 presents metrics for the two steps of the verification. The first metric is the number of typing rule applications for bitfield manipulation operations. Note that the number of typing rule applications in Fig. 12 is higher than the number of operators in Fig. 10 since, by default, RefinedC verifies each path through a function separately and thus sometimes type checks the same operator multiple times. In addition to the typing rules in §3, Fig. 12 also lists the statistics for the casting rule (column *cast*) and conversion rule (column *convert*) mentioned in §6.

The second metric is the number of manually solved side conditions and all generated side conditions (last two columns of Fig. 12).[10] Overall, only 3 out of 155 side conditions needed to be solved manually: in other words, all the other 152 side conditions (98.1%) were discharged without manual proofs.

The three side conditions that required manual proof came from a use of the right-shift operator ("≫") in an *arithmetic* context (not a bitfield context), which is *not* the target domain of BFF. Following RefinedC's standard procedure, we first solved these side conditions via interactive verification inside Coq (by rewriting with the `Z.shiftr_div_pow2` lemma from the Coq standard library that converts right shifting to an equivalent arithmetic expression and then invoking the `lia` solver). Then, we added the proof script to the C code via a `rc::tactics` annotation such that it is automatically inserted by the frontend to solve the side conditions. After that, RefinedC successfully verified all the functions we considered.

Lastly, we measured the verification time on an M1-chip MacBook Pro with 16 GB memory. The times for each codebase are listed in the last column of Fig. 12, with an overall time of 411 s. Individual function verification times varied from 1 s to 75 s (median: 4 s, 90th-percentile: 28.6 s). While this may not be fast enough for interactive verification in some cases, we believe it is sufficient in most situations.

8 RELATED WORK

Bitfield manipulation. Jhala and Majumdar [2006] present a constraint solving-based type inference algorithm that tags each integer in a bitfield-manipulating program with a bit vector type which describes the layout (similar to our signatures). Based on the inferred types, they "compile away" low-level bitwise operations, yielding a high-level program that encodes bit vectors as C structs (much like our high-level record representation). Rather than relying on SMT bit vector theories, which are slower than other well-tuned theories (*e.g.*, integer arithmetic), they then rely on software model checkers to verify properties of the translated high-level code.

Jhala and Majumdar [2006]'s approach is more automated than ours in that they can infer bit vector types automatically, whereas we assume the signatures are provided by developers as part of the specification. However, due to the difference in memory representation between the original bitfield representation and their high-level C struct version, their approach does not foundationally verify the original low-level program, whereas BFF does. BFF also supports compositional verification, whereas they do not. Lastly, we handle bitwise negation on bit masks, which is not covered by their approach.

Verification of bit-manipulating programs via SMT solvers. For verifying bit-manipulating programs, the most common approach adopted by existing deductive verification tools (such as Dafny [Leino 2010], Viper [Müller et al. 2016], and F* [Swamy et al. 2016]) is to (1) generate verification conditions (VCs) for bit-manipulating code based directly on the semantics of the operations used, and then (2) pass these VCs to an SMT solver instantiated with a bit-vector theory. This approach has the advantage of leveraging the great deal of work that has been put into SMT

[10]These side conditions include conditions from the rules in §3, but also from other typing rules and from proving pre- and postconditions of functions.

solving with bit vectors. In particular, there is an expressive theory for reasoning about bitwise and bit-propagating operators (*e.g.*, extraction, concatenation, shifts) between fixed-size bit vectors defined in SMT-LIB [Barrett et al. 2010]. Examples of state-of-the-art SMT solvers that support this theory are Yices [Dutertre and De Moura 2006], Z3 [de Moura and Bjørner 2008b], Boolector [Brummayer and Biere 2009], MathSAT [Cimatti et al. 2013], and CVC4 [Barrett et al. 2011] (also the latest release CVC5 [Barbosa et al. 2022]).

However, this approach comes at the cost of essentially abandoning a fully foundational proof. First of all, to our knowledge, none of the existing tools establish a formal connection between the VCs they generate and the semantics of the original program (though in principle they could). Second and more importantly, the correctness of SMT solvers has not generally been certified in proof assistants, and as recent studies have revealed, their implementations can be buggy [Winterer et al. 2020b,a; Mansur et al. 2020; Park et al. 2021]. For example, OpFuzz [Winterer et al. 2020a] found 819 confirmed unique bugs (including 184 soundness bugs) in Z3 and CVC4 during one year of extensive testing. Although it is possible to build certified decision procedures, such as the fixed-width quantifier-free bit-vector theory [Shi et al. 2021] and the bit blasting algorithm [Swords and Davis 2011], completely verifying state-of-the-art SMT solvers is still well out of reach.

One way to address this concern is to instrument solvers to emit correctness certificates: if the formula is satisfiable, then one can simply check if the satisfying model/assignment is correct by evaluating the formula under this model; otherwise, the solver emits a proof for the unsatisfiability that is checkable by trusted verifiers (*e.g.*, proof assistants). CVC4 [Barrett et al. 2011] can produce such certificates for bit-vector queries, based on LFSC [Stump et al. 2013], a meta-logic that serves as a unified proof format for SMT solvers. Z3 [de Moura and Bjørner 2008a; Böhme et al. 2011] also supports certificate generation and the proofs can be reconstructed in Isabelle/HOL. SMTCoq [Ekici et al. 2017] can check certificates from veriT [Bouton et al. 2009] and CVC4. One obvious downside of this solution is that the certificates must be verified case-by-case, which is time-consuming; and in the worst case, the verification may fail if the solver implementation is unsound (as reported by Park et al. [2021]). In that situation, the user may fall back to use general bit-vector libraries (*e.g.*, in Coq [The Coqutil Team 2022] or Isabelle [Lochbihler 2018; Dross et al. 2015]) to establish the formal proof themselves.

In contrast to the SMT-based approach, BFF's *structured bit vectors* can be regarded as a restricted fragment of the general bit-vector theory that supports effective automation and is easy to embed foundationally in Coq, yet remains expressive enough for handling standard patterns of bitfield manipulation.

9 CONCLUSION

This paper presents BFF, a foundational and automated approach for verifying bitfield-manipulating programs. The key insight is that typical bitfield manipulation operates on the logical, high-level structure of fields, which are packed into integers. While we have implemented BFF in RefinedC, we believe that this insight is more general and that our approach can be integrated into refinement type-based frameworks other than RefinedC. Programmers typically think in terms of the logical structure of bitfields, so specification can be done using that higher-level structure (§2.2) rather than the lower-level bit layout that most SMT-based approaches work with. Additionally, since only a handful of primitive bitfield operations are used in practice (§3), these can be given specialized typing rules (Fig. 6), producing verification conditions that are easier to check automatically (§4.2).

ACKNOWLEDGMENTS

This research was supported in part by a European Research Council (ERC) Consolidator Grant for the project "RustBelt", funded under the European Union's Horizon 2020 Framework Programme

(grant agreement no. 683289), in part by a Google PhD Fellowship (Sammler), and in part by awards from Android Security's ASPIRE program and from Google Research.

REFERENCES

Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. 2010. Semantic foundations for typed assembly languages. *TOPLAS* 32, 3 (2010), 1–67.

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. CVC5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The SMT-LIB standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, Vol. 13. 14.

Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

Sascha Böhme, Anthony C. J. Fox, Thomas Sewell, and Tjark Weber. 2011. Reconstruction of Z3's Bit-Vector Proofs in HOL4 and Isabelle/HOL. In *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 7086)*, Jean-Pierre Jouannaud and Zhong Shao (Eds.). Springer, 183–198. https://doi.org/10.1007/978-3-642-25379-9_15

Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. 2009. veriT: An Open, Trustable and Efficient SMT-Solver. In *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5663)*, Renate A. Schmidt (Ed.). Springer, 151–156. https://doi.org/10.1007/978-3-642-02959-2_12

Robert Brummayer and Armin Biere. 2009. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5505)*, Stefan Kowalewski and Anna Philippou (Eds.). Springer, 174–177. https://doi.org/10.1007/978-3-642-00768-2_16

Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7795)*, Nir Piterman and Scott A. Smolka (Eds.). Springer, 93–107. https://doi.org/10.1007/978-3-642-36742-7_7

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008a. Proofs and Refutations, and Z3. In *Proceedings of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics, Doha, Qatar, November 22, 2008 (CEUR Workshop Proceedings, Vol. 418)*, Piotr Rudnicki, Geoff Sutcliffe, Boris Konev, Renate A. Schmidt, and Stephan Schulz (Eds.). CEUR-WS.org. http://ceur-ws.org/Vol-418/paper10.pdf

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008b. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Will Deacon. 2020. Virtualization for the Masses: Exposing KVM on Android. https://www.youtube.com/watch?v=wY-u6n75iXc. KVM Forum Talk.

Claire Dross, Clément Fumex, Jens Gerlach, and Claude Marché. 2015. *High-level functional properties of bit-level programs: Formal specifications and automated proofs.* Ph. D. Dissertation. Inria Saclay.

Bruno Dutertre and Leonardo De Moura. 2006. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf* 2, 2 (2006), 1–2.

Jake Edge. 2020. KVM for Android. https://lwn.net/Articles/836693/.

Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. 2017. SMTCoq: A Plug-In for Integrating SMT Solvers into Coq. In *CAV (2) (Lecture Notes in Computer Science, Vol. 10427)*. Springer,

126–133. https://doi.org/10.1007/978-3-319-63390-9_7

Ranjit Jhala and Rupak Majumdar. 2006. Bit level types for high level reasoning. In *SIGSOFT FSE*. ACM, 128–140. https://doi.org/10.1145/1181775.1181791

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6355)*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

Rodolphe Lepigre, Michael Sammler, Kayvan Memarian, Robbert Krebbers, Derek Dreyer, and Peter Sewell. 2022. VIP: Verifying real-world C idioms with integer-pointer casts. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–32. https://doi.org/10.1145/3498681

Andreas Lochbihler. 2018. Fast Machine Words in Isabelle/HOL. In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10895)*, Jeremy Avigad and Assia Mahboubi (Eds.). Springer, 388–410. https://doi.org/10.1007/978-3-319-94821-8_23

Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 701–712. https://doi.org/10.1145/3368089.3409763

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9583)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2021. Generative type-aware mutation for testing SMT solvers. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–19. https://doi.org/10.1145/3485529

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the foundational verification of C code with refined ownership types. In *PLDI*. ACM, 158–174. https://doi.org/10.1145/3453483.3454036

Xiaomu Shi, Yu-Fu Fu, Jiaxiang Liu, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2021. CoqQFBV: A Scalable Certified SMT Quantifier-Free Bit-Vector Solver. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 149–171. https://doi.org/10.1007/978-3-030-81688-9_7

Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean, and Cesare Tinelli. 2013. SMT proof checking using a logical framework. *Formal Methods Syst. Des.* 42, 1 (2013), 91–118. https://doi.org/10.1007/s10703-012-0163-3

Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F. In *POPL*. ACM, 256–270. https://doi.org/10.1145/2837614.2837655

Sol Swords and Jared Davis. 2011. Bit-Blasting ACL2 Theorems. In *Proceedings 10th International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2011, Austin, Texas, USA, November 3-4, 2011 (EPTCS, Vol. 70)*, David S. Hardin and Julien Schmaltz (Eds.). 84–102. https://doi.org/10.4204/EPTCS.70.7

The Coqutil Team. 2022. coqutil. https://github.com/mit-plv/coqutil.

The Tokei Team. 2022. Tokei. https://github.com/XAMPPRocky/tokei.

Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020a. On the Unusual Effectiveness of Type-Aware Operator Mutations for Testing SMT Solvers. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 193 (nov 2020), 25 pages. https://doi.org/10.1145/3428261

Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020b. Validating SMT Solvers via Semantic Fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 718–730. https://doi.org/10.1145/3385412.3385985

Fengmin Zhu, Michael Sammler, Rodolphe Lepigre, Derek Dreyer, and Deepak Garg. 2022. BFF: Foundational and Automated Verification of Bitfield-Manipulating Programs (Artifact). https://doi.org/10.5281/zenodo.7079022 Repository: https://plv.mpi-sws.org/refinedc/bff.