Static Analysis of Endian Portability by Abstract Interpretation[⋆]

David Delmas[1,2], Abdelraouf Ouadjaout[2], and Antoine Miné[2,3]

[1] Airbus Operations S.A.S., 316 route de Bayonne, 31060 Toulouse Cedex 9, France
[2] Sorbonne Université, CNRS, LIP6, 75005 Paris, France
[3] Institut universitaire de France, 1 rue Descartes, 75231 Paris Cedex 5, France
`david.delmas@airbus.com`, `antoine.mine@lip6.fr`,
`abdelraouf.ouadjaout@lip6.fr`

Abstract. We present a static analysis of endian portability for C programs. Our analysis can infer that a given program, or two syntactically close versions thereof, compute the same outputs when run with the same inputs on platforms with different byte-orders, *a.k.a.* endiannesses. We target low-level C programs that abuse C pointers and unions, hence rely on implementation-specific behaviors undefined in the C standard.

Our method is based on abstract interpretation, and parametric in the choice of a numerical abstract domain. We first present a novel concrete collecting semantics, relating the behaviors of two versions of a program, running on platforms with different endiannesses. We propose a joint memory abstraction, able to infer equivalence relations between little- and big-endian memories. We introduce a novel symbolic predicate domain to infer relations between individual bytes of the variables in the two programs, which has near-linear cost, and the right amount of relationality to express (bitwise) arithmetic properties relevant to endian portability. We implemented a prototype static analyzer, able to scale to large real-world industrial software, with zero false alarms.

Keywords: Formal Methods · Abstract Interpretation · Abstract Domains · Static Analysis · C Programming Language · Portability · Endianness · Industrial Application

[⋆] This work is performed as part of a collaborative partnership between Sorbonnne Université / CNRS (LIP6) and Airbus. This work is partially supported by the European Research Council under the Consolidator Grant Agreement 681393 – MOPSA.

1 Introduction

There is no consensus on the representation of a multi-byte scalar value in computer memory [10]. Some systems store the least-significant byte at the lowest address, while others do the opposite. The former are called little-endian, the latter big-endian. Such systems include processor architectures, network protocols and data storage formats. For instance, Intel processors are little-endian, while internet protocols and some legacy processors, such as SPARC, are big-endian. As a consequence, programs relying on assumptions on the encoding of scalar types may exhibit different behaviors when run on platforms with different byte-orders, *a.k.a.* endiannesses. The case occurs typically with low-level C software, such as device drivers or embedded software. Indeed, the C standard [20] leaves the encoding of scalar types partly unspecified. The precise representation of types is standardized in implementation-specific *Application Binary Interfaces* (ABI), such as [3], to ensure the interoperability of compiled programs, libraries, and operating systems. Although it is possible to write fully portable, ABI-neutral C code, the vast majority of C programs rely on assumptions on the ABI of the platform, such as endianness. Therefore, the typical approach used, when porting a low-level C program to a new platform with opposite endianness, is to eliminate most of the byte-order-dependent code, and to wrap the remainder, if any, in conditional inclusion directives, which results in two syntactically close endian-specific variants of the same program. A desirable property, which we call endian portability, is that a program computes the same outputs when run with the same inputs on the little- and big-endian platforms. By extension, we also say that a program is endian portable if two endian-specific variants thereof compute the same outputs when run with the same inputs on their respective platforms. In this paper, we describe a static analysis which aims at inferring the endian portability of large real-world low-level C programs.

Motivating example. For instance, Example 1 features a snippet of code for reading network input. The sequence of bytes read from the network is first stored into integer variable x. Assume variable y has the same type. x is then either copied, or byte-swapped into y, depending on the endianness of the platform. Our analysis is able to infer that Example 1 is endian portable, *i.e.* both endian-specific variants compute the same value for y, whatever the values of the bytes read from the network. This property is expressed by the assertion at line 8.

Example 1. Reading input in network byte-order.

```
1    read_from_network((uint8_t *)&x, sizeof(x));
2  # if __BYTE_ORDER == __LITTLE_ENDIAN
3      uint8_t *px = (uint8_t *)&x, *py = (uint8_t *)&y;
4      for (int i=0; i<sizeof(x); i++) py[i] = px[sizeof(x)-i-1];
5  # else
6      y = x;
7  # endif
8    assert_sync(y);
```

Example 1 abuses pointers to bypass the C type system, a common practice in low-level programming known as *type punning*. Alternatively, some implementations rely on bitwise arithmetics. E.g., if `x` and `y` have type `uint32_t`, the little-endian case may be rewritten as `((x & 0xff000000) >> 24) | ((x & 0xff0000) >> 8) | ((x & 0xff00) << 8) | ((x & 0xff) << 24)`. Other implementations rely on compiler built-in functions, or assembly code, possibly using dedicated processor instructions. Examples can be seen in the Linux implementations of the POSIX `htons` and `htonl` functions, converting values between host and network byte-order. Our analysis is able to analyze all the above C implementations successfully, as well as alternative implementations (with stubs for assembly code). In the following of the paper, unless otherwise stated, we will implicitly refer to a version of Example 1 where variables have type `uint16_t`.

Approach. Low-level programs exhibit different semantics when run on platforms with different endiannesses. We thus model them as so-called double programs. The little-endian program is called the first (or left, or little-endian) version of the double program, while the big-endian program is called the second (or right, or big-endian) version. Both versions may share the same source code, or present syntactic differences (if conditional inclusion is used). Our approach to endian portability is to devise a joint, whole-program static analysis of a double program able to infer equivalences between the input-output relations of its versions. To this aim, we define a memory model able to represent a joint abstraction of their memories. We first parameterize a standard memory domain for low-level C programs with an explicit endianness parameter. Then, we lift it to double programs, and tailor it to infer, and represent symbolically, relevant equalities between little- and big-endian memories. We rely on a dedicated numerical domain based on symbolic predicates, to infer complementarity relations between individual bytes of program variables, such as those established by bitwise arithmetic operations. We validate our approach by analyzing large industrial low-level embedded C programs designed to be endian portable.

Related work. Several approaches to endian portability are developed in the literature. [31] relies on a source-to-source translation, which is only sound with respect to annotations provided by the programmer, whereas we require no annotations. [6] extends a compiler to generate code that executes with the opposite byte order semantics as the underlying architecture, at the cost of a performance penalty. Annotations are also required for soundness in some cases. [22] relies on dynamic analysis, which can find portability errors, but cannot prove endian portability formally, unlike our method. The Sparse [7] static analysis tool used by Linux kernel developers relies on pervasive type annotations to detect endiannesses issues, but comes with no formal guarantee.

To our knowledge, no prior work uses sound static analysis to infer endian portability. Yet, our approach leverages prior work. We build on a memory abstract domain [25],[28, Sect. 5.2] developed for run-time error analysis of low-level C programs able to expose endian-dependent behaviors, and on double program semantics developed for patch analysis [15,16]. Our symbolic predicate domain is based on previous work on predicate domains [27], and symbolic constant propagation [26]. Our domain is also reminiscent of the Slice domain introduced in [9,8] for another purpose, and implemented differently.

Contributions. The main contributions of this work are:

– We present a novel concrete collecting semantics, relating the behaviors of two versions of a program, running on platforms with different endiannesses.
– We propose a joint memory abstraction able to infer equivalence relations between little- and big-endian memories.
– We introduce a novel symbolic predicate domain to infer relations between individual bytes of the variables in the two programs, which has near-linear cost, and the right amount of relationality to express (bitwise) arithmetic properties relevant to endian portability.
– We implemented our analysis on the Mopsa [30,21] platform. Our prototype is able to scale to large real-world industrial software, with zero false alarms.

The paper is organised as follows. Section 2 formalizes the concrete collecting semantics, Sect. 3 describes the memory abstraction, Sect. 4 describes the numerical abstraction and introduces a novel numeric domain, Sect. 5 presents experimental results with a prototype implementation. Section 6 concludes.

2 Syntax and concrete semantics

Following the standard approach to abstract interpretation [11], we develop a concrete collecting semantics for a C-like language for double programs. The $\|$ operator may occur anywhere in the parse tree to denote syntactic differences between the left (little-endian) and right (big-endian) versions of a double program. However, $\|$ operators cannot be nested: a double program only describes a pair of programs. Given double program P with variables in $\mathcal{V}$, we call its left (resp. right) version $P_1 = \pi_1(P)$ (resp. $P_2 = \pi_2(P)$), where π_1 (resp. π_2) is a version extraction operator, defined by induction on the syntax, keeping only the left (resp. right) side of $\|$ symbols. For instance, $\pi_1(x \leftarrow 1 \parallel y \leftarrow 0) = x \leftarrow 1$, and $\pi_2(x \leftarrow 1 \parallel y \leftarrow 0) = y \leftarrow 0$, while $\pi_1(z \leftarrow 0) = z \leftarrow 0 = \pi_2(z \leftarrow 0)$. Recall that syntactic differences between P_1 and P_2 may be distinct from semantic differences. Syntactically different statements may exhibit the same semantics in P_1 and P_2, like in Example 1, while syntactically equal statements may exhibit different semantics, like with the C statement `*((`**`char*`**`)&x)=1`, when integer variable `x` is such that $\mathit{sizeof}(x) > 1$.

2.1 Syntax

Simple programs P_1 and P_2 enjoy a standard, C-like syntax presented in Fig. 1. Statements *stat* are built on top of expressions *expr* and Boolean conditions *cond*. The syntax of double statements *dstat* includes specific **assume_sync** and **assert_sync** statements, used for specifications. The former is used to express

$$\begin{array}{lcl}
\textit{scalar-type} & ::= & \textit{int-sign}\ \textit{int-type} \mid \mathbf{ptr} \\
\textit{type} & ::= & \textit{scalar-type} \mid \ldots \\
\\
\circ & ::= & \texttt{-} \mid \texttt{~} \mid (\textit{scalar-type}) \\
\diamond & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{\&} \mid \texttt{|} \mid \texttt{\^{}} \mid \texttt{>>} \mid \texttt{<<} \\
\\
\textit{stat} & ::= & *_{\textit{scalar-type}}\ \textit{expr} \leftarrow \textit{expr} \\
 & \mid & \mathbf{if}\ \textit{cond}\ \mathbf{then}\ \textit{stat}\ \mathbf{else}\ \textit{stat} \\
 & \mid & \ldots \\
\\
\textit{dstat} & ::= & \textit{stat} \\
 & \mid & \textit{stat} \parallel \textit{stat} \\
 & \mid & \textit{spec} \\
 & \mid & \mathbf{if}\ \textit{dcond}\ \mathbf{then}\ \textit{dstat}\ \mathbf{else}\ \textit{dstat} \\
 & \mid & \ldots
\end{array}$$

$$\begin{array}{lcll}
\textit{int-sign} & ::= & \mathbf{signed} \mid \mathbf{unsigned} & \\
\textit{int-type} & ::= & \mathbf{char} \mid \mathbf{short} \mid \mathbf{int} \mid \mathbf{long} \mid \mathbf{long\ long} & \\
\\
\textit{expr} & ::= & *_{\textit{scalar-type}}\ \textit{expr} & \\
 & \mid & \&V & \\
 & \mid & [c_1, c_2] & c_1, c_2 \in \mathbb{Z} \\
 & \mid & \circ\ \textit{expr} & \\
 & \mid & \textit{expr} \diamond \textit{expr} & \\
\\
\textit{cond} & ::= & \textit{expr} \bowtie 0 & \bowtie \in \{\leq, \geq, =, \neq, <, >\} \\
\textit{spec} & ::= & \mathbf{assume\_sync}(\textit{expr}) & \\
 & \mid & \mathbf{assert\_sync}(\textit{expr}) & \\
\textit{dcond} & ::= & \textit{cond} & \\
 & \mid & \textit{cond} \parallel \textit{cond} &
\end{array}$$

Fig. 1. Syntax of simple and double programs.

assumptions on program inputs, while the latter is used to express assertions on program outputs: **assume_sync**(e) introduces the assumption that expression e evaluates to the same value in double program versions P_1 and P_2, while **assert_sync**(e) checks that the value of e is identical in both versions, and fails otherwise. Expression $[c_1, c_2]$ chooses a value non-deterministically between constants c_1 and c_2. The double statement $x \leftarrow [c_1, c_2]$ may assign different values to variable x in the two program versions. In contrast, the sequence $x \leftarrow [c_1, c_2]$; **assume_sync**(x) ensures that x holds the same non-deterministic value in both versions.

Expressions rely on a C-like type-system. Integer and pointer types are collectively referred to as scalar types. Expressions support pointer arithmetic, expressed as byte-level offset arithmetic. All left-values are assumed to be preprocessed to dereferences $*_\tau\ e$ (*i.e.* `*((`τ`*)e)` in C) where τ is a scalar type, and e is a pointer expression. Note that dereferences are limited to scalar types, and the dereferenced type is explicit in the syntax.

2.2 Semantics of low-level simple C programs

The semantics of simple programs is parameterized by an ABI. In this paper, we assume program versions have the same ABIs, but for endianness. Let $\mathcal{A} \triangleq \{\, \mathcal{L}, \mathcal{B} \,\}$ denote the possible endiannesses (little- and big-endian). The sizes of types, in contrast, are the same for both program versions. We thus assume a unique function $\textit{sizeof} \in \textit{type} \to \mathbb{N}$ given, which provides these sizes (in bytes).

Pointer values are modeled as (semi-)symbolic addresses of the form $\langle V, i \rangle \in \mathcal{A}\textit{ddr} \triangleq \mathcal{V} \times \mathbb{Z}$, which indicate an offset of i bytes from the first byte of V. Special pointer values are defined for C's `NULL` and dangling pointers: $\mathcal{P}\textit{tr} \triangleq \mathcal{A}\textit{ddr} \cup \{\mathbf{NULL}, \mathbf{invalid}\}$.

Let $\mathbb{B} \triangleq [0, 255] \cup (\mathcal{P}\textit{tr} \times \mathbb{N})$ describe the possible numeric byte values and symbolic pointer bytes. We keep pointer values symbolic as their precise numeric values depend on memory allocation strategies outside the scope of the analysis. $\langle p, i \rangle \in \mathbb{B}$ denotes the $i-$th byte in the memory representation of the pointer value p. Expressions manipulate scalar values, which may be numeric (machine

$$\begin{aligned}
&\mathbb{E}[\![*_\tau e]\!]_\alpha \rho \triangleq \{ v \mid \langle V, o\rangle \in \mathbb{E}[\![e]\!]_\alpha \rho \wedge 0 \le o \le \mathit{sizeof}(V) - \mathit{sizeof}(\tau) \\
&\qquad\qquad \wedge v \in \mathit{bdec}_{\tau,\alpha}(\rho\langle V, o\rangle, \ldots, \rho\langle V, o + \mathit{sizeof}(\tau) - 1\rangle) \} \\
&\mathbb{S}[\![*_\tau e_1 \leftarrow e_2]\!]_\alpha R \triangleq \\
&\quad \textstyle\bigcup_{\rho\in R} \{ \rho[\forall i < \mathit{sizeof}(\tau) : \langle V, o + i\rangle \mapsto b_i] \mid \langle V, o\rangle \in \mathbb{E}[\![e_1]\!]_\alpha \rho \\
&\qquad\qquad \wedge 0 \le o \le \mathit{sizeof}(V) - \mathit{sizeof}(\tau) \wedge (b_0, \ldots, b_{\mathit{sizeof}(\tau)-1}) \in \mathit{benc}_{\tau,\alpha}(\mathbb{E}[\![e_2]\!]_\alpha \rho) \}
\end{aligned}$$

Fig. 2. Concrete semantics of memory reads and writes.

integers) or pointer values. We denote the set of values as $\mathbb{V} \triangleq \mathbb{Z} \cup \mathcal{P}tr$. The definition of the most concrete semantics requires a family of representation functions $\mathit{benc}_{\tau,\alpha} \in \mathbb{V} \to \mathcal{P}(\mathbb{B}^*)$, that convert a scalar value of given type $\tau \in \mathit{scalar\text{-}type}$ and endianness $\alpha \in \mathcal{A}$ into a sequence of $\mathit{sizeof}(t)$ byte values. We denote as $\mathit{bdec}_{\tau,\alpha} \in \mathbb{B}^* \to \mathcal{P}(\mathbb{V})$ the converse operation. For instance, on a 32-bit platform, $\mathit{benc}_{\textbf{unsigned int},\mathcal{L}}(1) = \{ (1, 0, 0, 0) \}$, $\mathit{bdec}_{\textbf{unsigned short},\mathcal{B}}(0, 1) = \{ 1 \}$, and $\mathit{benc}_{\textbf{ptr},\mathcal{L}}(p) = \{ (\langle p, 0\rangle, \langle p, 1\rangle, \langle p, 2\rangle, \langle p, 3\rangle) \}$. This seemingly trivial encoding allows modeling copying pointer values byte per byte, as done *e.g.* by `memcpy`. Note that the $\mathit{benc}_{\tau,\alpha}$ and $\mathit{bdec}_{\tau,\alpha}$ functions return a set of possible values. For instance, reinterpreting a pointer value as an integer, as in $\mathit{bdec}_{\textbf{int},\mathcal{L}} \circ \mathit{benc}_{\textbf{ptr},\mathcal{L}}(p)$, returns the full range of type **int**. We do not detail the definitions of these functions here, for the sake of conciseness. An example may be found in [28, Sec. 5.2].

Environments are elements of $\mathcal{E} \triangleq \mathcal{A}\mathit{ddr} \rightharpoonup \mathbb{B}$. The semantics $\mathbb{E}[\![\mathit{expr}]\!] \in \mathcal{A} \to \mathcal{E} \to \mathcal{P}(\mathbb{V})$ and $\mathbb{S}[\![\mathit{stat}]\!] \in \mathcal{A} \to \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$ for simple expressions and statements is defined by standard induction on the syntax. We therefore only show, on Fig. 2, the semantics $\mathbb{E}[\![*_\tau e]\!]_\alpha$ and $\mathbb{S}[\![*_\tau e_1 \leftarrow e_2]\!]_\alpha$ for memory reads and writes, given endianness $\alpha \in \mathcal{A}$. Bytes are fetched and decoded with $\mathit{bdec}_{\tau,\alpha}$ when reading from memory in expression $*_\tau e$, while values computed by expression e_2 are encoded into bytes with $\mathit{benc}_{\tau,\alpha}$ when writing to memory in assignment $*_\tau e_1 \leftarrow e_2$. Note that illegal memory accesses are silently omitted to simplify the presentation.

2.3 Semantics of double programs

We now lift simple program semantics $\mathbb{S}$ to double program semantics $\mathbb{D}$. As both simple program versions $P_k = \pi_k(P)$ have concrete states in $\mathcal{E}$, the double program P has concrete states in $\mathcal{D} \triangleq \mathcal{E} \times \mathcal{E}$. The semantics of P_k is parameterized by its endianness $\alpha_k \in \mathcal{A}$. We assume, without loss of generality, that P_1 is the little-endian version, and P_2 the big-endian one.

$\mathbb{D}[\![s]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$ describes the relation between input and output states of s, which are pairs of states of simple programs. The definition for $\mathbb{D}[\![s]\!]$ is shown on Fig. 3. $\mathbb{D}$ leverages previous work on patch analysis [15,16]. It is defined by induction on the syntax, so as to allow for a modular definition and joint analyses of double programs. Note that $\mathbb{D}$ is parametric in $\mathbb{S}$.

The semantics for the empty program is the identity function. The semantics $\mathbb{D}[\![s_1 \parallel s_2]\!]$ for the composition of two syntactically different statements reverts

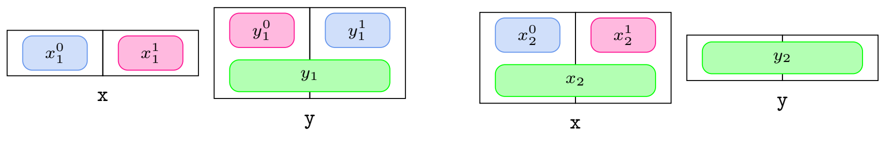
$$
\begin{array}{lcl}
\multicolumn{3}{l}{\underline{\mathbb{D}[\![\, \mathit{dstat} \,]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})}} \\
\mathbb{D}[\![\, \textbf{skip} \,]\!] & \triangleq & \lambda X.\, X \\
\mathbb{D}[\![\, s_1 \parallel s_2 \,]\!]X & \triangleq & \bigcup_{(\rho_1,\rho_2)\in X} \prod_{k\in\{1,2\}} \mathbb{S}[\![\, s_k \,]\!]_{\alpha_k} \{\, \rho_k \,\} \\
\mathbb{D}[\![\, l \leftarrow e \,]\!] & \triangleq & \mathbb{D}[\![\, l \leftarrow e \parallel l \leftarrow e \,]\!] \\
\mathbb{D}[\![\, \textbf{assume\_sync}(e) \,]\!]X & \triangleq & \{\, (\rho_1,\rho_2) \in X \mid \exists v \in \mathbb{V} : \forall k \in \{\,1,2\,\} : \mathbb{E}[\![\, e \,]\!]_{\alpha_k} \rho_k = \{\, v \,\} \,\} \\
\mathbb{D}[\![\, s \,;\, t \,]\!] & \triangleq & \mathbb{D}[\![\, t \,]\!] \circ \mathbb{D}[\![\, s \,]\!] \\
\mathbb{D}[\![\, \textbf{if } e_1 \bowtie 0 \parallel e_2 \bowtie 0 \textbf{ then } s \textbf{ else } t \,]\!] & \triangleq & \mathbb{D}[\![\, s \,]\!] \circ \mathbb{F}[\![\, e_1 \bowtie 0 \parallel e_2 \bowtie 0 \,]\!] \\
 & \dot\cup & \mathbb{D}[\![\, \pi_1(s) \parallel \pi_2(t) \,]\!] \circ \mathbb{F}[\![\, e_1 \bowtie 0 \parallel e_2 \not\bowtie 0 \,]\!] \\
 & \dot\cup & \mathbb{D}[\![\, \pi_1(t) \parallel \pi_2(s) \,]\!] \circ \mathbb{F}[\![\, e_1 \not\bowtie 0 \parallel e_2 \bowtie 0 \,]\!] \\
 & \dot\cup & \mathbb{D}[\![\, t \,]\!] \circ \mathbb{F}[\![\, e_1 \not\bowtie 0 \parallel e_2 \not\bowtie 0 \,]\!] \\
\mathbb{D}[\![\, \textbf{if } c \textbf{ then } s \textbf{ else } t \,]\!] & \triangleq & \mathbb{D}[\![\, \textbf{if } c \parallel c \textbf{ then } s \textbf{ else } t \,]\!] \\
\mathbb{D}[\![\, \textbf{while } e_1 \bowtie 0 \parallel e_2 \bowtie 0 \textbf{ do } s \,]\!]X & \triangleq & \mathbb{F}[\![\, e_1 \not\bowtie 0 \parallel e_2 \not\bowtie 0 \,]\!](\mathrm{lfp}\ H) \\
\mathbb{D}[\![\, \textbf{while } c \textbf{ do } s \,]\!] & \triangleq & \mathbb{D}[\![\, \textbf{while } c \parallel c \textbf{ do } s \,]\!] \\
\text{where} \quad \mathbb{F}[\![\, e_1 \bowtie 0 \parallel e_2 \bowtie 0 \,]\!]X & \triangleq & \{\, (\rho_1,\rho_2) \in X \mid \forall k \in \{\,1,2\,\} : \exists v_k \in \mathbb{E}[\![\, e_k \,]\!]_{\alpha_k} \rho_k : v_k \bowtie 0 \,\} \\
\text{and} \quad H(I) & \triangleq & X \\
 & \dot\cup & \mathbb{D}[\![\, s \,]\!] \circ \mathbb{F}[\![\, e_1 \bowtie 0 \parallel e_2 \bowtie 0 \,]\!]I \\
 & \dot\cup & \mathbb{D}[\![\, \pi_1(s) \parallel \textbf{skip} \,]\!] \circ \mathbb{F}[\![\, e_1 \bowtie 0 \parallel e_2 \not\bowtie 0 \,]\!]I \\
 & \dot\cup & \mathbb{D}[\![\, \textbf{skip} \parallel \pi_2(s) \,]\!] \circ \mathbb{F}[\![\, e_1 \not\bowtie 0 \parallel e_2 \bowtie 0 \,]\!]I
\end{array}
$$

Fig. 3. Denotational concrete semantics of double programs.

to the pairing of the simple program semantics of individual simple statements s_1 and s_2. The semantics for assignments is defined with this construct. The semantics of **assume_sync** and **assert_sync** statements filters away environments where the left and right versions of a double program may disagree on the value of expression e. In addition, **assert_sync** raises an alarm if e may evaluate to different values in P_1 and P_2. We omit alarms from the semantics for conciseness. The semantics for the sequential composition of statements boils down to the composition of the semantics of individual statements. The semantics for selection statements relies on the filter $\mathbb{F}[\![\, e_1 \bowtie 0 \parallel e_2 \bowtie 0 \,]\!]$ to distinguish between cases where both versions agree on the value of the controlling expression, and cases where they do not (*a.k.a.* unstable tests). There are two stable and two unstable test cases, according to the evaluations of the two conditions. The semantics for stable test cases is standard. The semantics for unstable test cases is defined by the composition of left version of the **then** branch, filtered by the condition, and of the right version of the **else** branch, filtered by the negation of the condition (and the dual case). The semantics for (possibly unbounded) iteration statements is defined using the least fixpoint of a function defined similarly.

2.4 Properties of Interest

We wish to prove the functional equivalence between the left and right versions of a given double program $P \in \mathit{dstat}$, restricted to a set of distinguished outputs, specified with the **assert_sync** primitive. Let $x_0 \in \mathcal{D}$ be an initial double-program state. The set of states reachable by P is $\mathbb{D}[\![\, P \,]\!]\{\, x_0 \,\}$. Let Ω be a set of output left-values of program P. The property of interest is that $\pi_1(P)$ and



(a) Program 1 (b) Program 2

Fig. 4. Memory cells of Example 1: 🟦 $= b_0$, 🟪 $= b_1$, 🟩 $= b_0 \times 2^8 + b_1$.

$\pi_2(P)$ compute equal values for all outputs:

$$\forall l \in \Omega : \forall \langle \rho_1, \rho_2 \rangle \in \mathbb{D}[\![\, P \,]\!] \{\, x_0 \,\} : \exists v \in \mathbb{V} : \forall k \in \{\, 1, 2 \,\} : \mathbb{E}[\![\, l \,]\!]_{\alpha_k} \rho_k = \{\, v \,\} \quad .$$

For instance, let S denote the set of reachable states of Example 1, before line 8: $S = \{\, \langle [x_1^o \mapsto b_o, y_1^o \mapsto b_{1-o}], [x_2^o \mapsto b_o, y_2^o \mapsto b_o] \rangle \mid o \in \{0,1\},\ (b_0, b_1) \in [0, 255]^2 \,\}$, where b_0 and b_1 denote the values of bytes read from the network, and we write x_i^o and y_i^o for $\langle x, o \rangle$ and $\langle y, o \rangle$ in program version i. The portability property expressed at line 8 is $y_1^0 + 2^8 y_1^1 = 2^8 y_2^0 + y_2^1$, which can be proved from S.

Our concrete collecting semantics $\mathbb{D}$ is not computable in general. We will thus rely on computable abstractions, to infer this property by static analysis. Note that the use of **assume_sync** and **assert_sync** in specifications allows for both whole-program analysis, and separate analyses of program parts.

3 Memory abstraction

Though we aim at designing a computable abstract semantics in Sect. 4, we first tailor a (non computable) abstraction of our memory model. We rely on the Cells memory abstraction of simple programs [25],[28, Sect. 5.2]. In order to handle C programs computing with machine integers of multiple sizes, with byte-level access to their encoding through type-punning, this domain represents the memory as a dynamic collection of scalar variables, termed cells, holding values for the scalar memory dereferences discovered during the analysis. It maintains a consistent abstract state despite the introduction of overlapping cells by type-punning. We lift this memory abstraction to double programs, and we extend it for representing equalities between cells symbolically.

3.1 Cells

We first consider the finite universe $\mathcal{C}ell \triangleq \mathcal{V} \times \mathbb{N} \times$ *scalar-type* $\times \mathcal{A}$ of cells of one program. A cell $\langle V, o, \tau, \alpha \rangle \in \mathcal{C}ell$ is denoted as a variable V, an offset o, and information specifying the encoding of values: a scalar type τ and endianness α. To account for both programs, we introduce *projected cells* as $\widetilde{\mathcal{C}ell} \triangleq \mathcal{C}ell \times \{\, 1, 2 \,\}$, where 1 (resp. 2) denotes a cell in the memory of P_1 (resp. P_2).

For instance, consider the program in Example 1. We show in Fig. 4 the cells synthesized at the end of the program. Let $x_k \triangleq \langle x, 0, \mathbf{u16}, \alpha_k, k \rangle$ denote 2-byte cells for `x` in Program $k \in \{1, 2\}$, where $\alpha_1 = \mathcal{L}$ and $\alpha_2 = \mathcal{B}$. 1-byte cells are denoted as $x_k^o \triangleq \langle x, o, \mathbf{u8}, \alpha_k, k\rangle$ where $o \in \{0, 1\}$. The cells for `y` are defined in a similar way. Both program versions first call function `read_from_network`, which reads a stream of bytes from an external source, and writes it into a buffer. The same stream is read by both program versions. A stub for `read_from_network` is shown in Fig. 5. After completion of the call, we have $x_1^0 = b_0 = x_2^0$ and

```
void read_from_network(u8 buf[], u32 size) {
  for (int i=0; i<size; i++) {
    buf[i] = [0,255];
    assume_sync( buf[i] );
  }
}
```

Fig. 5. Stub for `read_from_network` function.

$x_1^1 = b_1 = x_2^1$, where b_0 and b_1 are the first and second bytes read from the network, respectively. Then, Program 1 swaps the bytes of `x` into those of `y`: $x_1^0 = y_1^1$ and $x_1^1 = y_1^0$. Program 2, in contrast, assigns `x` to `y`. `x` is thus read as a 2-byte cell, while only 1-byte cells are present. Therefore, the Cells domain synthesizes x_2 by adding the constraint $x_2 = 2^8 x_2^0 + x_2^1$, following big-endian byte-order, before performing the assignment $y_2 \leftarrow x_2$. To sum up, we obtain the following constraints:

$$x_1^0 = x_2^0 = y_1^1 \qquad x_1^1 = x_2^1 = y_1^0 \qquad y_2 = x_2$$

In addition to the cell constraints on `x` and `y`:

$$x_1 = x_1^0 + 2^8 x_1^1 \qquad y_1 = y_1^0 + 2^8 y_1^1 \qquad x_2 = 2^8 x_2^0 + x_2^1 \qquad y_2 = 2^8 y_2^0 + y_2^1$$

Our goal is to prove that $y_1 = y_2$ given such constraints. To do so, we want to leverage numerical domains to abstract the values of cells. However, such constraints require an expressive domain, such as polyhedra or linear equalities, that can hamper the scalability of the analysis. In addition, we note that we need to infer many equalities, most of them between the left and right versions of the same cells. This is no surprise as we expect most variables to hold equal values in the little- and big-endian memories most of the time, with only local differences. Rather than relying completely on the expressiveness of the underlying numeric domain, we first optimize our memory model for this common case, introducing the concept of shared bi-cells, which act as a symbolic representation of cells equality.

3.2 Shared bi-cells

We denote as $\mathcal{B}icell \triangleq \widetilde{\mathcal{C}ell} \cup (\widetilde{\mathcal{C}ell} \times \widetilde{\mathcal{C}ell})$ the set of bi-cells. A bi-cell is either a projected cell in $\widetilde{\mathcal{C}ell}$, or a pair of such cells in $\widetilde{\mathcal{C}ell} \times \widetilde{\mathcal{C}ell}$ assumed to hold equal value, called a *shared bi-cell*. Bi-cell sharing allows a single representation, in the memory environment, for two projected cells from different program versions at the same memory location and holding equal values. Abstract memory states of double programs are modeled as a choice of a set of bi-cells $C \subseteq \mathcal{B}icell$, and a set of scalar environments on C. Let $\mathcal{D}^\flat \triangleq \bigcup_{C \subseteq \mathcal{B}icell} \{\, \langle C, R \rangle \mid R \in \mathcal{P}(C \to \mathbb{V}) \,\}$ be the associated abstract domain. An abstract state represents a set of concrete byte-level memories in $\mathcal{D} = \mathcal{E} \times \mathcal{E}$. The values of the bytes of these memories must satisfy all the numeric constraints on bi-cells implied by the environments:

$$\begin{aligned}\gamma_{\mathcal{B}icell}\langle C, R\rangle \triangleq \{\, (\mu_1, \mu_2) \in \mathcal{D} \mid \exists \rho \in R :\ & \forall c_k = \langle V, o, \tau, \alpha, k\rangle \in \widetilde{\mathcal{C}ell} : \\ & \forall c \in occ(c_k, C) : \exists (b_0, \ldots, b_{sizeof(\tau)-1}) \in benc_{\tau,\alpha}(\rho(c)) : \\ & \forall 0 \le i < sizeof(\tau) : \mu_k\langle V, o+i\rangle = b_i \,\}\end{aligned}$$

where $occ \in \widetilde{\mathcal{C}ell} \times \mathcal{P}(\mathcal{B}icell) \to \mathcal{P}(\mathcal{B}icell)$ records occurrences of a projected cell among bi-cells: $occ(c, C) \triangleq \{\, c' \in C \mid c' = c \vee \exists c'' :\ c' = \langle c, c''\rangle \vee c' = \langle c'', c\rangle \,\}$.

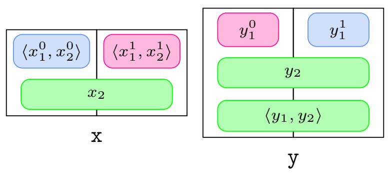

$\langle x_1^0, x_2^0\rangle$
$\langle x_1^1, x_2^1\rangle$
x_2
x
y_1^0
y_1^1
y_2
$\langle y_1, y_2\rangle$
y

Fig. 6. Shared bi-cells of Example 1.

In Fig. 6, we depict the bi-cells obtained after analyzing the program shown in Example 1. For variable `x`, since `read_from_network` writes the same value to x_1^0 and x_2^0, we can synthesize the shared bi-cell $\langle x_1^0, x_2^0\rangle$ to represent the equality $x_1^0 = x_2^0$. In a similar way, we synthesize the shared bi-cell $\langle x_1^1, x_2^1\rangle$. Therefore, as opposed to the separate representation of the memories of Programs 1 and 2 in Fig. 4, the joint representation induced by bi-cell sharing allows reducing the burden on numeric domains. In the following, we describe more involved cell synthesis operations that allow us to realize $\langle y_1, y_2\rangle$, and thus to infer that $y_1 = y_2$.

3.3 Cell synthesis

A cornerstone of our memory model is bi-cell synthesis. In order to read or write a scalar value to a given location of memory, we must create a suitable bi-cell, or retrieve an existing one from the environment. To guarantee the soundness of the analysis when adding a new bi-cell, it is necessary to ensure that values assigned to it are consistent with those of existing overlapping bi-cells. Our memory domain first attempts to synthesize shared bi-cells if an equality can be inferred from the environment, by pattern-matching. In case of failure, it safely defaults to a pair of projected bi-cells, the values of which are set according to those of existing overlapping bi-cells.

We have already used shared bi-cell synthesis implicitly on Fig. 6. When reading variable `y` at the end of Example 1, the memory domain attempts to synthesize $\langle y_1, y_2\rangle$, as a proof of $y_1 = y_2$. To this aim, it searches, among possible patterns, for an existing cell, equal to both y_1 and y_2. x_2 is a candidate, assuming equality $x_2 = y_2$ is recorded in (an abstraction of) the environment. Then the domain looks for 1-byte bi-cells for y_1 and x_2, and finds the four blue and red

$$\phi^\flat\langle V,o,\tau\rangle\langle C,R\rangle \triangleq \begin{cases} \langle c_1,c_2\rangle & \textbf{if } \mathit{equal}(c_1,c_2,\alpha_1,\alpha_2)\langle C,R\rangle \\ \langle c_1^*,c_2\rangle & \textbf{else if } \mathit{equal}(c_1,c_2,\alpha_2,\alpha_2)\langle C,R\rangle \\ \langle c_1,c_2^*\rangle & \textbf{else if } \mathit{equal}(c_1,c_2,\alpha_1,\alpha_1)\langle C,R\rangle \\ \top & \textbf{otherwise} \end{cases} \qquad \begin{aligned} \text{where } c_i &= \langle V,o,\tau,\alpha_i,i\rangle, \\ c_i^* &= \langle V,o,\tau,\alpha_{3-i},i\rangle, \\ \alpha_1 &= \mathcal{L}, \text{ and} \\ \alpha_2 &= \mathcal{B}. \end{aligned}$$

Fig. 7. Shared bi-cell synthesize function.

cells from Fig. 6. As y_1 and x_2 have opposite endian encodings, it queries the environment for equalities $y_1^0 = \langle x_1^1, x_2^1\rangle$ and $y_1^1 = \langle x_1^0, x_2^0\rangle$. The success of the synthesis relies on pattern-matching, and three equalities which may be inferred by a numerical domain implementing simple symbolic propagation.

Shared bi-cell synthesis. More generally, function $\phi^\flat$ formalizes the patterns matched attempting to synthesize a shared bi-cell for a given dereference $c \in \mathcal{C}ell_0 \triangleq \mathcal{V} \times \mathbb{N} \times \mathit{scalar\text{-}type}$. An implementation is proposed in Fig. 7. Firstly, it returns $\langle c_1, c_2\rangle$ if $c_1 = c_2$ may be inferred from the environment, where $c_i = \langle c, \alpha_i, i\rangle$ are projected versions of c, with the native endiannesses of their respective platforms. Otherwise, it returns $\langle c_1^*, c_2\rangle$, where c_1^* is a big-endian projected bi-cell of Program 1, if $c_1^* = c_2$ holds. For instance, $\langle x_1^*, x_2\rangle$ will be synthesized if variable `x` is read after the end of Example 1. Otherwise, it returns $\langle c_1, c_2^*\rangle$, where c_2^* is a little-endian projected bi-cell of Program 2, if $c_1 = c_2^*$ holds. Finally, if all fails, it returns an error $\top$. $\phi^\flat \in \mathcal{C}ell_0 \to \mathcal{D}^\flat \to \widetilde{\mathcal{C}ell}^2 \cup \{\top\}$ relies on predicate *equal* to compare two projected bi-cells of the same type, with specified endianness encodings. An implementation is shown on Fig. 8. *equal* returns *true* when compared cells are part of a shared bi-cell, or when equality is ensured by the environment. Otherwise, it compares individual 1-byte bi-cells of the same weights 2^{8w}, at endianness-dependent offsets: $\mathit{offset}(w,s,\alpha) \triangleq w$ if $\alpha = \mathcal{L}$, and $s-w-1$ otherwise. Otherwise, *equal* searches for candidate projected bi-cells in the environment, equal to both c and c'. In the formula, we denote the set of projected bi-cells in the environment as $\mathit{flatten}(C) \triangleq \{\, c \in \widetilde{\mathcal{C}ell} \mid c \in C \vee \exists c' \in C : \langle c, c'\rangle \in C \vee \langle c', c\rangle \in C \,\}$. *equal* returns *true* in case of success, *false* otherwise.

Projected bi-cell synthesis. If all attempts to synthesize a shared bi-cell $\langle c_1, c_2\rangle$, $\langle c_1^*, c_2\rangle$, or $\langle c_1, c_2^*\rangle$ fail, our memory domain creates the pair of projected bi-cells c_1 and c_2 instead. To set their values soundly, it calls $\phi_1(c_1)(C)$ and $\phi_2(c_2)(C)$, where $\phi_i(c_i)(C)$ returns a syntactic expression denoting (an abstraction of) the value of c_i as a function of cells existing in C. For instance, $\phi_1(\langle y, 0, \mathbf{u16}, \mathcal{B}, 1\rangle)(C) = 2^8 y_1^0 + y_1^1$ at the end of Example 1 (see Fig. 6).

To define the synthesize functions ϕ_1 and $\phi_2 \in \widetilde{\mathcal{C}ell} \to \mathcal{P}(\mathcal{B}icell) \to \mathit{expr}$ for projected bi-cells, we first need to define a generic cell synthesize function $\phi \in \mathcal{C}ell \to \mathcal{P}(\mathcal{C}ell) \to \mathit{expr}$, such that $\phi(c)(C)$ returns a syntactic expression

$$\begin{array}{l}
\mathit{equal}(\langle V, o, \tau, -, k\rangle, \langle V', o', \tau, -, k'\rangle, \alpha, \alpha')\langle C, R\rangle \triangleq \\
\quad \text{let } c = \langle V, o, \tau, \alpha, k\rangle \text{ and } c' = \langle V', o', \tau, \alpha', k'\rangle \text{ and } s = \mathit{sizeof}(\tau) \text{ in} \\
\quad \langle c, c'\rangle \in C \vee \langle c', c\rangle \in C \ \vee \\
\quad (\exists (x, x') \in \mathit{occ}(c, C) \times \mathit{occ}(c', C) : \forall \rho \in R : \ \rho(x) = \rho(x')) \ \vee \\
\quad \left(\forall 0 \le w < s : \ \mathit{equal}(c^{\mathit{offset}(w,s,\alpha)}, c'^{\,\mathit{offset}(w,s,\alpha')}, \alpha, \alpha')\langle C, R\rangle\right) \ \vee \\
\quad (\exists x \in \mathit{flatten}(C) \setminus \{ c, c' \} : \ \mathit{equal}(c, x, \alpha, \alpha_x)\langle C, R\rangle \wedge \mathit{equal}(c', x, \alpha', \alpha_x)\langle C, R\rangle)
\end{array}$$

where c^p denotes the 1-byte bi-cell $\langle V, o + p, \mathbf{u8}, \alpha, k\rangle$ (and respectively for c'^p), and α_x denotes the endianness encoding of x.

Fig. 8. Equality test between projected bi-cells.

$$\phi\langle V, o, t, e\rangle(C) \triangleq \begin{cases}
\langle V, o, t, e\rangle \ \textbf{if} \ \langle V, o, t, e\rangle \in C \\
\mathit{wrap}(\langle V, o, t', e\rangle, \mathit{range}(t)) \ \textbf{else if} \ \langle V, o, t', e\rangle \in C \wedge t, t' \in \textit{int-type} \wedge \mathit{sizeof}(t) = \mathit{sizeof}(t') \\
\mathit{byte}(\langle V, o - b, t', e'\rangle, w(e', b, \mathit{sizeof}(t'))) \\
\qquad\qquad \textbf{else if} \ \langle V, o - b, t', e'\rangle \in C \wedge \ t = \mathbf{u8} \wedge t' \in \textit{int-type} \wedge b < \mathit{sizeof}(t') \\
\mathit{wrap}(\sum_{i=0}^{\mathit{sizeof}(t)-1} 2^{8 \times w(e, i, \mathit{sizeof}(t))} \times \langle V, o + i, \mathbf{u8}, e_i\rangle, \mathit{range}(t)) \\
\qquad\qquad \textbf{else if} \ \forall 0 \le i < \mathit{sizeof}(t) : \ \langle V, o + i, \mathbf{u8}, e_i\rangle \in C \wedge t \in \textit{int-type} \\
\mathit{range}(t) \qquad \textbf{else if} \ t \in \textit{scalar-type} \\
\textbf{invalid} \qquad \textbf{else if} \ t = \mathbf{ptr}
\end{cases}$$

Fig. 9. Generic cell synthesize function.

denoting (an abstraction of) the value of the cell c as a function of cells in C. ϕ is designed as an extension to multiple endianness encodings of the cell synthesize function originally proposed in [28, sec. 5.2].

An example implementation is proposed in Fig. 9. Firstly, if the cell already exists ($c \in C$), it is directly returned by ϕ. Otherwise, ϕ looks for integer cells of the same size and different signedness, and converts them using function *wrap* to model wrap-around, and function *range* for the range of the type: $\mathit{wrap}(v, [l, h]) \triangleq \min \{ v' \mid v' \ge l \wedge \exists k \in \mathbb{Z} : \ v = v' + k(h - l + 1) \}$, and $\mathit{range}(t) \triangleq [0, 2^{8s} - 1]$ if t is unsigned, and $[-2^{8s-1}, 2^{8s-1} - 1]$ if t is signed, where $s = \mathit{sizeof}(t)$. Thirdly, ϕ extracts unsigned bytes from integers. Fourthly, ϕ aggregates unsigned bytes into integers. Function $w \in \mathcal{A} \times \mathbb{N}^2 \to \mathbb{N}$ is used to model the endianness-dependent weight of bytes in integers: $w(\mathcal{L}, b, s) \triangleq b$ and $w(\mathcal{B}, b, s) \triangleq s - b - 1$. The value of the byte of weight 2^{8w} in integer x is: $\mathit{byte}(x, w) = \lfloor x / 2^{8w} \rfloor \mod 2^8$. When all fails, ϕ returns the full range of the type (or **invalid**, for a pointer). Many definitions are possible for ϕ, e.g. adding cases to support floats, or to synthesize integer cells from cells of opposite endianness.

To define ϕ_1 and ϕ_2, we project bi-cells of the appropriate side onto cells, apply ϕ, and lift the resulting cell expression back to a bi-cell expression. More precisely, to compute $\phi_1\langle c, 1\rangle(C)$, we first project the bi-cell set C to the cells of program version 1 : $C_1 \triangleq \{ x \mid \langle x, 1\rangle \in C \ \vee \ \exists y : \ \langle\langle x, 1\rangle, \langle y, 2\rangle\rangle \in C \}$.

Then, we retrieve the constraints on cell c by applying the generic cell synthesize function: $e_1 \triangleq \phi(c)(C_1)$. Finally, $\phi_1\langle c, 1\rangle(C)$ is obtained by substituting every cell x occurring in e_1 with an element of $occ(\langle x, 1\rangle, C)$. Note that e_1 is a syntactic expression over cells in C_1, and $occ(\langle x, 1\rangle, C) \neq \emptyset$ for all $x \in C_1$. The definition of $\phi_2\langle c, 2\rangle(C)$ is analogue.

Cell addition. Cell addition, $\textit{add-cell}^\flat \in \mathcal{C}ell_0 \to \mathcal{D}^\flat \to \mathcal{D}^\flat$, then simply adds the cell(s) and initializes their value(s).

```
$\textit{add-cell}^\flat(c)\langle C, R\rangle \triangleq$
    if $\phi^\flat(c)\langle C, R\rangle = \langle x_1, x_2\rangle$ then
        $\langle C \cup \{\, \langle x_1, x_2\rangle \,\}, \{\, \rho[\langle x_1, x_2\rangle \mapsto v] \mid \rho \in R,\ v \in \mathbb{E}[\![\, \phi_1(x_1)(C) \,]\!]_{\alpha_1}\rho \,\}\rangle$
    else
        $\langle C \cup \{c_1, c_2\}, \{\, \rho[\forall i:\ c_i \mapsto v_i] \mid \rho \in R,\ \forall i:\ v_i \in \mathbb{E}[\![\, \phi_i(c_i)(C) \,]\!]_{\alpha_i}\rho \,\}\rangle$
```

where $c_1 = \langle c, \mathcal{L}, 1\rangle$ and $c_2 = \langle c, \mathcal{B}, 2\rangle$.

3.4 Abstract join

The abstract join must merge environment sets defined on heterogeneous bi-cell sets. We therefore define a unification function $\textit{unify} \in (\mathcal{D}^\flat)^2 \to (\mathcal{D}^\flat)^2$. $\textit{unify}(\langle C_1, R_1\rangle, \langle C_2, R_2\rangle)$ adds, with $\textit{add-cell}^\flat$, any missing cells to $\langle C_1, R_1\rangle$ and $\langle C_2, R_2\rangle$: respectively $C_2 \setminus C_1$ and $C_1 \setminus C_2$. Let $\langle C_1', R_1'\rangle$ and $\langle C_2', R_2'\rangle$ be the resulting abstract states. C_1' and C_2' may include both projected and shared bi-cells. A shared bi-cell that does not occur in both C_1' and C_2' cannot be soundly included in the unified state, as it conveys equality information that holds for one abstract state only. All such cells are thus removed before unification. Formally, $\textit{unify}(\langle C_1, R_1\rangle, \langle C_2, R_2\rangle) = (\langle C_{12}, R_1''\rangle, \langle C_{12}, R_2''\rangle)$, where $C_{12} = (C_1' \cup C_2') \setminus (((C_1' \cup C_2') \setminus \widetilde{\mathcal{C}ell}) \setminus (C_1' \cap C_2'))$, and $R_k'' = \{\, \rho_{|C_{12}} \mid \rho \in R_k' \,\}$. The abstract join may now be defined as: $\langle C_1, R_1\rangle \sqcup \langle C_2, R_2\rangle \triangleq \langle C_{12}, R_1'' \cup R_2''\rangle$.

3.5 Semantics of simple statements

Before defining the semantics for double statements in this domain, we first define the semantics $\mathbb{E}_k^\flat[\![\, *_t\, e \,]\!] \in \mathcal{D}^\flat \to \mathcal{P}(\mathbb{V})$ and $\mathbb{S}_k^\flat[\![\, *_t\, e_1 \leftarrow e_2 \,]\!] \in \mathcal{P}(\mathcal{D}^\flat) \to \mathcal{P}(\mathcal{D}^\flat)$ for simple memory reads and writes, in program version $k \in \{\, 1, 2 \,\}$.

Evaluations. To compute $\mathbb{E}_k^\flat[\![\, *_t\, e \,]\!]\,\langle C, R\rangle$, we first resolve $*_t\, e$ into a set L of projected bi-cells on side k, by evaluating e into a set of pointer values, and gathering projected bi-cells corresponding to valid pointers:

$$L \triangleq \{\, \langle V, o, t, \alpha_k, k\rangle \mid \langle V, o\rangle \in \mathbb{E}[\![\, e \,]\!]_{\alpha_k}\rho,\ \rho \in R,\ 0 \le o \le \textit{sizeof}(V) - \textit{sizeof}(t) \,\}$$

Then, we call $\textit{add-cell}^\flat$ to ensure that all the target cells in L are in the abstract environment, which updates $\langle C, R\rangle$ to $\langle C_0, R_0\rangle$. Finally: $\mathbb{E}_k^\flat[\![\, *_t\, e \,]\!]\,\langle C, R\rangle = \{\, \rho(c) \mid \rho \in R_0,\ c \in L \,\}$.

Assignments. The semantics of assignments $\mathbb{S}^\flat_k[\![*_t e_1 \leftarrow e_2]\!] \langle C, R\rangle$ involves more steps. Like for evaluations, we start with resolving $*_t e_1$ into a set L of projected bi-cells on side k. Then, we realize the cells in L using *add-cell*$^\flat$: let $\langle C_0, R_0\rangle$ be the updated environment. Some of the projected bi-cells in L may have been realized into shared bi-cells. Let $S \triangleq (C_0 \setminus C) \cap \widetilde{\mathcal{C}ell}^2$ be the set of such shared bi-cells. Elements of S represent equalities between bi-cells projected on side k, and on side opposite to k. Such equalities may no longer hold, after assignment on side k. Therefore, we split shared bi-cells of S into their left and right projections, in a copy-on-write strategy. The updated environment is:

$$\langle C_0', R_0'\rangle = \langle C_0 \cup \bigcup_{\langle c,c'\rangle \in S} \{ c, c' \}, \{ c \mapsto \begin{cases} \rho(x) \text{ if } \exists x \in occ(c, S) \neq \emptyset \\ \rho(c) \text{ otherwise} \end{cases} \mid \rho \in R_0 \}\rangle$$

Finally, we update the environment for the projected bi-cells written (elements of L), with the possible values of e_2. However, this is not sufficient: it is also necessary to update the environment for any overlapping bi-cells, including shared bi-cells that have been split into pairs of projected cells. A sound and efficient (though possibly coarse) solution is to simply remove them. Indeed, removing any bi-cell is always sound in our memory model: it amounts to losing information, as we loose constraints on the byte-representation of the memory. Let $\Omega \subseteq \mathcal{C}_0' \setminus L$ be the set of such bi-cells: elements of Ω are shared bi-cells and projected bi-cells on side k, with offsets and sizes such that they overlap some element of L. The updated environment is:

$$\begin{aligned} \mathbb{S}^\flat_k[\![*_t e_1 \leftarrow e_2]\!] \langle C, R\rangle = \langle & C_0' \setminus \Omega, \\ & \{ \rho_{|C_0' \setminus \Omega}[\forall c \in L : c \mapsto v] \mid \rho \in R_0',\, v \in \mathbb{E}^\flat_k[\![e_2]\!] \langle C_0', R_0'\rangle \}\rangle \end{aligned}$$

3.6 Semantics of double statements

We are now ready to define the semantics $\mathbb{D}^\flat[\![dstat]\!] \in \mathcal{D}^\flat \to \mathcal{D}^\flat$ of double statements in this domain. Like $\mathbb{D}$, $\mathbb{D}^\flat$ is defined by induction on the syntax. We focus on base cases, as inductive cases are unchanged.

The semantics $\mathbb{D}^\flat[\![s_1 \parallel s_2]\!]$ for two syntactically different statements composes simple programs semantics: $\mathbb{D}^\flat[\![s_1 \parallel s_2]\!] \triangleq \mathbb{S}^\flat_2[\![s_2]\!] \circ \mathbb{S}^\flat_1[\![s_1]\!]$. The semantics for **assume_sync**, **assert_sync**, and $\mathbb{F}^\flat[\![e_1 \bowtie 0 \parallel e_2 \bowtie 0]\!]$ are mostly unchanged, but for symbolic simplifications taking advantage of symbolic representations of equalities in our domain, for improved efficiency and precision. In particular, when e is a deterministic expression containing a single dereference, then $\mathbb{D}^\flat[\![\textbf{assume\_sync}(e)]\!]$ adds a shared bi-cell for this dereference to the abstract environment. Consistently, $\mathbb{D}^\flat[\![\textbf{assert\_sync}(e)]\!]$ first tests whether e is deterministic, and its dereferences evaluate to shared bi-cells. In this case, $\mathbb{D}^\flat[\![\textbf{assert\_sync}(e)]\!]$ raises no alarm. Otherwise, the semantics uses environment functions ρ to test equalities of bi-cell values, like for $\mathbb{D}$. A similar symbolic simplification is used for the $\mathbb{F}^\flat[\![\cdot]\!]$ filter: $\mathbb{F}^\flat[\![e \bowtie 0 \parallel e \not\bowtie 0]\!]\langle C, R\rangle = \emptyset$ (hence the test is stable) when e is deterministic and all dereferences evaluate to shared bi-cells, which is the common case. For instance, when evaluating $\mathbb{D}[\![\textbf{if}\,(x < y)\,\textbf{then}\, s\, \textbf{else}\, t]\!]$, if the dereferences for variable x and y evaluate to shared bi-cells, the two unstable tests cases are $\bot$.

Assignments. In an assignment $\mathbb{D}^\flat[\![*_t e_1 \leftarrow e_2]\!] \langle C, R\rangle$, although both programs execute the same syntactic assignment, their semantics are different, as are their endiannesses. In addition, available bi-cells may be different. By default, double assignments are straightforward extensions of simple assignments: $\mathbb{D}^\flat[\![*_t e_1 \leftarrow e_2]\!] = \mathbb{S}_2^\flat[\![*_t e_1 \leftarrow e_2]\!] \circ \mathbb{S}_1^\flat[\![*_t e_1 \leftarrow e_2]\!]$. We introduce two precision optimizations, taking advantage of implicit equalities represented by shared bi-cells. We first transform $*_t e_1$ and the dereferences in e_2 into sets of bi-cells L and R, respectively. R may be empty, as e_2 may be a constant expression. Then, we realize the cells in L and R, using *add-cell*$^\flat$. Let $\langle C_0, R_0\rangle$ be the updated environment. Two optimizations are possible, depending on e_1, e_2, L, and R.

Optimization 1: Assignment of shared bi-cells. If e_1 and e_2 are deterministic expressions, and if they evaluate to bi-cells that are all shared ($L \cup R \subseteq \widetilde{\mathcal{C}ell}^2$), then Programs 1 and 2 write the same value to the same destination. We thus update shared destination bi-cells (in L), and remove any overlapping bi-cells. Formally:

$$\mathbb{D}^\flat[\![*_t e_1 \leftarrow e_2]\!] \langle C, R\rangle = \langle C_0 \setminus \Omega, \\ \{ \rho_{|C_0\setminus\Omega}[\forall c \in L : c \mapsto v] \mid \rho \in R_0, v \in \mathbb{E}_1^\flat[\![e_2]\!] \langle C_0, R_0\rangle \}\rangle,$$

where $\Omega \subseteq \mathcal{C}_0 \setminus L$ is the set of (shared or projected) bi-cells overlapping elements of L. The choice of evaluating $\mathbb{E}_1^\flat[\![e_2]\!]$ (rather than $\mathbb{E}_2^\flat[\![e_2]\!]$) is arbitrary, as they are equal. Indeed, endianness $\alpha_1 = \mathcal{L}$ is not used by $\mathbb{E}_1^\flat[\![e_2]\!]$, as all the necessary cells are materialized before evaluating expression e_2.

Optimization 2: Copy assignment. If the conditions for optimization *1* are satisfied, and if, in addition, $e_2 = *_t e_2'$, and both $*_t e_1$ and $*_t e_2'$ evaluate to single bi-cells ($|L| = |R| = 1$), then we are dealing with a copy assignment. We may thus soundly copy a memory information from the source $\{l\} = L$ to the destination $\{r\} = R$, so as to further improve precision. We therefore create a copy of r, and any smaller bi-cell for the same bytes, to a corresponding bi-cell for the bytes of l. Newly created destination bi-cells have the sides and endiannesses of their sources. The environment is updated accordingly, to reflect equalities between sources and destinations.

4 Value abstraction

Connecting to numerical domains. We now rely on numeric abstractions to abstract further $\mathbb{D}^\flat$ into a computable abstract semantics $\mathbb{D}^\sharp$, resulting in an effective static analysis. Like [28, Sec. 5.2], our memory domain translates memory reads and writes into purely numerical operations on synthetic bi-cells, that are oblivious to the double semantics of double programs: each bi-cell is viewed as an independent numeric variable, and each numeric operation is carried out on a single bi-cell store, as if emanated from a single program. In particular, we notice that the transfer function for simple assignments $\mathbb{S}_k^\flat[\![*_t e_1 \leftarrow e_2]\!]$ described in Sect. 3.5 has the form of that of an assignment in a purely numeric language, where bi-cells play the roles of the numeric variables. This property is a key motivation for the Cell domain and the extension presented in this paper. Bi-cells may thus be fed, as variables, to a numerical abstract domain for environment abstraction. Any standard numerical domain, such as polyhedra [12], may be used. Yet, as we aim at scaling to large programs, we restrict ourselves to combinations of efficient non-relational domains, intervals and congruences [18], together with a dedicated symbolic predicate domain.

We thus assume an abstract domain $\mathcal{D}^\sharp_C$ given, with concretization γ_C, for each bi-cell set $C \subseteq \mathcal{B}icell$. It abstracts $\mathcal{P}(C \to \mathbb{Z}) \simeq \mathcal{P}(\mathbb{Z}^{|C|})$, i.e., sets of points in a $|C|-$dimensional vector space. A cell of integer type naturally corresponds to a dimension in an abstract element. We also associate a distinct dimension to each cell with pointer type; it corresponds to the offset o of a symbolic pointer $\langle V, o\rangle \in \mathcal{P}tr$. In order to abstract fully pointer values, we enrich the abstract numeric environment with a map P associating to each pointer cell the set of variables it may point to. Hence, the abstract domain becomes: $\mathcal{D}^\sharp \triangleq \{\, \langle C, R^\sharp, P\rangle \mid C \subseteq \mathcal{C}ell,\ R^\sharp \in \mathcal{D}^\sharp_C,\ P \in P_C \to \mathcal{P}(\mathcal{V} \cup \{\, \mathbf{NULL}, \mathbf{invalid}\,\})\,\}$, where $P_C \subseteq C$ is the subset of bi-cells of pointer type. We refer to [28, Sec. 5.2] for a formal presentation of the concretization and the abstract operators.

Introducing a dedicated symbolic predicate domain. Recall Example 1 from Sect. 1. Various implementations are possible for the byte-swaps enforcing endian portability of software. Though Example 1 shows an implementation relying on type-punning, implementations relying on bitwise arithmetics are also commonplace. In addition, system-level software, such as [32], often rely on combinations of type-punning and bitwise arithmetics. Example 2 is a simplified instance of such programming idioms: as `y` has type `unsigned char`, `y|0xff00` and `(y<<8)|0xff` represent the same 16-bit word in different endiannesses.

Example 2. Byte-wise equal memories in different endiannesses.

```
  u16 x; u8 y = rand_u8(), *p = &x;
  assume_sync(y);
# if __BYTE_ORDER == __LITTLE_ENDIAN
    x = y | 0xff00;
# else
    x = (y << 8) | 0xff;
# endif
  assert_sync(p[0]); assert_sync(p[1]);
```

For a successful analysis of Example 2, the numerical domain must interpret bitwise arithmetic expressions precisely, and infer relations such as: the low-order (respectively high-order) byte of the little-endian (respectively big-endian) version of integer `x` is equal to `y`. Then, the interpretation of dereferences of `p` by the memory domain introduces similar relations between cells, thanks to the bi-cell synthesize function. In this example, it infers that the little-endian version of the low-address (respectively high-address) byte cell in `x` is equal to the low-order (respectively high-order) byte of `x` – and the converse for big-endian.

Predicate abstract domain. We use a domain based on pattern matching of expressions to detect arithmetic manipulations of byte values commonly implemented as bitwise arithmetics. It is not sufficient to match each expression independently, as computations are generally spread across sequences of statements. We need, in addition, to maintain some state that retains and propagates information between statements. We maintain this state in a predicate domain $\mathcal{P}red^\sharp \triangleq C \to \mathcal{B}its$, which maps each bi-cell $c \in C \subseteq \mathcal{B}icell$ to a syntactic expression e in a language $\mathcal{B}its$, as a symbolic representation of predicate $c = e$.

$$\mathcal{B}its ::= \top \mid \mathcal{S}lice$$
$$\mathcal{S}lice ::= n \mid c \mid \overrightarrow{c[i,j)}^{k} \mid (\mathcal{S}lice \,|\, \mathcal{S}lice) \qquad (n \in \mathbb{Z},\ c \in C,\ i, j, k \in \mathbb{N})$$

$\top$ denotes the absence of information. Otherwise, a syntactic predicate expression may be either a bit-slice, or a bitwise OR of bit-slices. A bit-slice may be an integer constant n, a bi-cell c, or a slice expression $\overrightarrow{c[i,j)}^{k}$ denoting the value obtained by shifting the bits of c between i and $j-1$ to position k: $\overrightarrow{c[i,j)}^{k} \triangleq \lfloor (c \bmod 2^j)/2^i \rfloor \times 2^k$. Each term of a bitwise OR of bit-slices represents a interval of bits, *e.g.* $[k, k+j-i)$ for a term $\overrightarrow{c[i,j)}^{k}$. We assume that bit-intervals do not overlap: each bit from the result comes from a single cell or constant. The ordering is flat, based on syntactic predicate equality:

$$X^\sharp \sqsubseteq^\sharp Y^\sharp \iff^{\triangle} \forall c \in C :\ X^\sharp(c) = Y^\sharp(c) \vee Y^\sharp(c) = \top$$

An abstract element $X^\sharp \in \mathcal{P}red^\sharp$ denotes the set of environments that satisfy all the predicates in $X^\sharp$, where predicates are evaluated as expressions:

$$\gamma_{\mathcal{P}red}(X^\sharp) \triangleq \{\, \rho \in C \to \mathbb{V} \mid \forall c \in C :\ X^\sharp(c) = \top \vee \rho(c) \in \mathbb{E}[\![\, X^\sharp(c) \,]\!]\rho \,\}$$

We do not present the abstract operators in this paper. Like that of the related symbolic constant domain [26], they are based on symbolic propagation, and implement simple algebraic simplifications. They exhibit similar, near-linear time cost in our experiments.

Analysis of Example 2. Before line 8, three cells are synthesized by the memory domain: $C_8 = \{\, x_1, x_2, y_{12} \,\}$, where $x_1 = \langle x, 0, \mathbf{u16}, \mathcal{L}, 1\rangle$ is the little-endian projected bi-cell of variable `x`, $x_2 = \langle x, 0, \mathbf{u16}, \mathcal{B}, 2\rangle$ is the big-endian one, and $y_{12} = \langle\langle y, 0, \mathbf{u8}, \mathcal{L}, 1\rangle, \langle y, 0, \mathbf{u8}, \mathcal{B}, 2\rangle\rangle$ is a shared bi-cell.

- y_{12} is created at line 2, and represents the fact that variable `y` has the same value in the little- and big-endian versions.
- The transfer function for assignment of the symbolic predicate domain infers invariants $x_1 = y_{12} \,|\, 65280$ from line 4, and $x_2 = 255 \,|\, \overrightarrow{y_{12}[0,8)}^{8}$ from line 6.
- Then, the dereferences of pointer `p` at line 8 are interpreted by the memory domain. Four more cells $\{\, x_k^o \mid (k, o) \in \{\,1,2\,\} \times \{\,0,1\,\} \,\}$ are added to the abstract environment, to denote the bytes of variable `x` in the little- and big-endian programs. More precisely, $x_1^o = \langle x, o, \mathbf{u8}, \mathcal{L}, 1\rangle$, and $x_2^o = \langle x, o, \mathbf{u8}, \mathcal{B}, 2\rangle$, at offsets $o \in \{\,0,1\,\}$. Following the bi-cell synthesize functions ϕ_1 and ϕ_2, these new bi-cells are added together with assumptions on their values. In practice, these assumptions are four tests, used by the memory domain to filter the abstract environment. These tests are $x_1^o = \mathit{byte}(x_1, o)$, and $x_2^o = \mathit{byte}(x_1, 1-o)$, for offsets $o \in \{\,0,1\,\}$, with $\mathit{byte}(n,k) = \lfloor n/2^{8k} \rfloor \bmod 2^8$.

These tests are then interpreted by the numerical domain, and the symbolic predicate domain in particular, as $x_1^o = \mathit{byte}'(x_1, o)$, and $x_2^o = \mathit{byte}'(x_2, 1-o)$, with $\mathit{byte}'(n,k) = \overrightarrow{n[8k, 8k+8)}^0$.

– Finally, the assertion line 8 is interpreted as tests $x_1^0 = x_2^0$ and $x_1^1 = x_2^1$ by the memory domain. The transfer function for tests in the symbolic predicate domain replaces all bi-cells with the symbolic expressions bound to them, if any. The tests are thus $\overrightarrow{(y_{12} \mid 65280)\,[0,8)}^0 = \overrightarrow{\left(255 \mid \overrightarrow{y_{12}[0,8)}^8\right)[8,16)}^0$ and $\overrightarrow{(y_{12} \mid 65280)\,[8,16)}^0 = \overrightarrow{\left(255 \mid \overrightarrow{y_{12}[0,8)}^8\right)[0,8)}^0$. Both tests evaluate to true, using symbolic simplifications (and integer arithmetic computations) supported by the transfer function.

Hence, the assertions line 8 are proved correct: at the end of the program, the memories for variable `x` are byte-wise equal in the little and big-endian versions.

5 Evaluation

We implemented our analysis into the Mopsa platform [30,21] designed to support modular developments of precise static analyses for multiple languages and multiple properties. Our prototype is composed of 3,000 lines of OCaml: 45% for the memory abstraction, 36% for the symbolic predicate domain, and 19% for double program management and iterators. It leverages 31,000 lines (excluding parsers) of elementary functions of Mopsa: framework and utilities (64%), generic iterators and numeric domains for analyses of all languages (11%), specific iterators and memory domains for the C language (25%). We have experimented our prototype on small idiomatic examples, open source software, and large industrial software. The analyses were run on a 3.4 GHz Intel® Xeon® CPU.

5.1 Idiomatic examples

We first check the precision and robustness of our analysis against a collection of small double C programs (between 20 and 100 LOC), inspired by various implementations of byte-swaps in Linux drivers, POSIX `htonl` functions, and industrial software.

A set of 9 programs illustrate network data processing. These programs are similar to Example 1 of Sect. 1. They receive an integer from the network, increment it, and send over the result. Necessary byte-swaps are implemented for little-endian versions of these programs. Each example program implements a different byte-swapping technique on a 2, 4, or 8-byte integer: type-punning with pointer casts (like in Example 1), unions, or bitwise arithmetics. Refer to Examples 4, 5, and 6 in artifact [17] for the source codes. We also analyze Example 2 from Sect. 4 to demonstrate the efficiency of our symbolic predicate domain.

Our prototype also handles floating-point data, which was omitted in the paper for the sake of conciseness. We developed small floating-point examples representative of industrial use-cases of Sect. 5.3. They include byte-swappings of simple or double precision floating-point numbers sent to or received from the network, on architectures where integers and floats are guaranteed to have the same byte-order. Type-punning is used to reinterpret floats as integers of the same size, which are byte-swapped using bitwise arithmetics. Also, a combination of type-punning and byte-swapping is used to extract exponents from double precision floats. The source codes of these Examples 8 and 9 is available in artifact [17]. All analyses run in less than 200 ms and report no false alarm.

5.2 Open source benchmarks

We then check the soundness, precision, and modularity of our analysis on three benchmarks based on open source software available on GitHub, with multiple commits for bug-fixes related to endianness portability. Refer to Examples 10, 11, and 12 in artifact [17] for relevant source codes excerpts. We analyze slices between 100 and 250 LOC, using primitives **assume_sync** and **assert_sync** for modular specifications of program parts.

Our first benchmark is an implementation of a tunneling driver [32] based on the Geneve [19] encapsulation network protocol, which uses big-endian integers as tunnel identifiers. The driver was introduced in the Linux kernel, and patched several times for endianness-related issues detected by Sparse [7]. Then, a performance optimization introduced a new endianness portability bug, which Sparse failed to detect. It was fixed a year later. Our analysis soundly reports this bug, as well as previous issues detected by Sparse. It reports no alarm on the fixed code. Our second benchmark is a core library of the mlx5 Linux driver [24] for ethernet and RDMA net devices [23]. We analyze a slice related to a patch, committed to fix an endianness bug introduced 3 years earlier, and undetected by Sparse despite the use of relevant annotations. The fix turned out to be incomplete, and was updated 6 months later. Our analysis soundly reports bugs on the two first versions, and no alarm on the third. Our third benchmark is extracted from a version of Squashfs [35], a compressed read-only filesystem for Linux, included in the LineageOS [34] alternative Android distribution. We analyze a slice related to a patch, committed to fix an endianness bug undetected by Sparse due to a lack of type annotations. Our analysis soundly reports the bug, and no alarm on the fixed version. All the analyses run within 1 second.

5.3 Industrial case study

We analyzed two components of a prototype avionics application, developed at Airbus for a civil aircraft. This application is written in C, and primarily targets an embedded big-endian processor. Nonetheless, it must be portable to little-endian commodity hardware, as its source code is reused as part of a simulator used for functional verification of SCADE [4] models. The supplement to the applicable aeronautical standard [1] related to model-based development [2] mandates, in this case, that "*an analysis should provide compelling evidence that the simulation approach provides equivalent defect detection and removal as testing of the Executable Object Code*". Airbus, known to rely on formal methods for other verification objectives [13,33,14,29,5], is currently considering the use of static analysis to verify this portability property.

Endianness is the main difference between the ABIs of the embedded computer and the simulator. We thus experimented our prototype analyzer on the modules of the application integrated to the simulator, to which we refer as A and S. Modules A and S are data-intensive reactive software, processing thousands of global variables, with very flat call graphs. Module A is in charge of acquiring and emitting data through aircraft buses. It is composed of about 1 million LOC, most of which generated automatically from a description of the avionics network. It handles integers, Booleans, single and double precision floats. The code features bounded loops, memcpys, pointer arithmetics, and type-punning with unions and pointer casts. It also uses bitwise arithmetics, among which several thousand byte-swaps related to endianness portability. Module S is in charge of the main applicative functions. It is composed of about 300,000 LOC, most of which generated automatically from SCADE models. It handles mostly Booleans and double precision floats. It features bounded loops and bitwise arithmetics, but no type-punning. The target application is required to meet its specifications for long missions. Analysis entry points contain loops with several million iterations to emulate this execution context.

Both analyses run in 5 abstract iterations. The analysis of A runs in 20.4 hours and uses 5.5 GB RAM. The analysis of S runs in 9.7 hours and uses 2.7 GB RAM. We worked with the development and simulation teams to analyze early prototypes, and incorporate findings into the development cycle. On current versions of both modules, both analyses report zero alarm related to endianness.

6 Conclusion

We presented a sound static analysis of endian portability for low-level C programs. Our method is based on abstract interpretation, and parametric in the choice of a numerical abstract domain. We first presented a novel concrete collecting semantics, relating the behaviors of two versions of a program, running on platforms with different endiannesses. Then we proposed a joint memory abstraction, able to infer equivalence relations between little- and big-endian memories. We introduced a novel symbolic predicate domain to infer relations between individual bytes of the variables in the two programs, which has near-linear cost. We implemented a prototype static analyzer, able to scale to large real-world industrial software, with zero false alarms.

In future work, we aim at extending our analysis to further ABI-related properties, such as portability between different layouts of C types, or sizes of machine integers. We also anticipate that our bi-cell sharing approach will benefit the analysis of patches [15,16] modifying C data-types, even if the two versions run under the same ABI. Finally, we are considering an industrial deployment of our endian portability analysis, as a means to address avionics certification objectives related to simulation fidelity, as mentioned in Sect. 5.3.

References

1. DO-178C: Software considerations in airborne systems and equipment certification (2011)
2. DO-331: Model-based development and verification supplement to DO-178C and DO-278A (2011)
3. AT & T, The Santa Cruz Operation Inc.: System V application binary interface (1997)
4. Berry, G.: Scade: Synchronous design and validation of embedded control software. In: Ramesh, S., Sampath, P. (eds.) Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems. pp. 19–33. Springer Netherlands, Dordrecht (2007)
5. Brahmi, A., Delmas, D., Essoussi, M.H., Randimbivololona, F., Atki, A., Marie, T.: Formalise to automate: deployment of a safe and cost-efficient process for avionics software. In: 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018). Toulouse, France (Jan 2018), `https://hal.archives-ouvertes.fr/hal-01708332`
6. Brevnov, E., Domeika, M., Loenko, M., Ozhdikhin, P., Tang, X., Willkinson, H.: Bec: Bi-endian compiler technology for porting byte order sensitive applications **16** (2012)
7. Brown, N.: Sparse: a look under the hood (2016), `https://lwn.net/Articles/689907/`
8. Chevalier, M.: Proving the Security of Software-Intensive Embedded Systems by Abstract Interpretation. Ph.D. thesis, Université PSL (Nov 2020)
9. Chevalier, M., Feret, J.: Sharing ghost variables in a collection of abstract domains. In: Beyer, D., Zufferey, D. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 158–179. Lecture Notes in Computer Science, Springer International Publishing (2020)
10. Cohen, D.: On holy wars and a plea for peace. Computer **14**(10), 48–54 (1981). https://doi.org/10.1109/C-M.1981.220208
11. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL'77. pp. 238–252. ACM (Jan 1977)
12. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL'78. pp. 84–97. ACM (1978)
13. Delmas, D., Souyris, J.: Astrée: from research to industry. In: SAS'07, LNCS, vol. 4634, pp. 437–451. Springer (Aug 2007)
14. Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., Védrine, F.: Towards an industrial use of fluctuat on safety-critical avionics software. In: Alpuente, M., Cook, B., Joubert, C. (eds.) FMICS. Lecture Notes in Computer Science, vol. 5825, pp. 53–69. Springer (2009)
15. Delmas, D., Miné, A.: Analysis of Program Differences with Numerical Abstract Interpretation. In: PERR 2019. Prague, Czech Republic (Apr 2019)
16. Delmas, D., Miné, A.: Analysis of Software Patches Using Numerical Abstract Interpretation. In: Chang, B.Y.E. (ed.) Proc. of the 26th International Static Analysis Symposium (SAS'19). Lecture Notes in Computer Science, vol. 11822, pp. 225–246. Bor-Yuh Evan Chang, Springer, Porto, Portugal (Oct 2019)

17. Delmas, D., Ouadjaout, A., Miné, A.: Artifact for Static Analysis of Endian Portability by Abstract Interpretation (2021). https://doi.org/10.5281/zenodo.5206794
18. Granger, P.: Static analysis of arithmetic congruences. Int. Journal of Computer Mathematics **30**, 165–199 (1989)
19. Gross, J., Ganga, I., Sridhar, T.: Geneve: Generic network virtualization encapsulation. RFC 8926, RFC Editor (November 2020)
20. ISO/IEC JTC1/SC22/WG14 working group: C standard. Tech. Rep. 1124, ISO & IEC (2007)
21. Journault, M., Miné, A., Monat, R., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: Proc. of the 11th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE19). Lecture Notes in Computer Science (LNCS), vol. 12031, pp. 1–18. Springer (Jul 2019)
22. Kápl, R., Parízek, P.: Endicheck: Dynamic analysis for detecting endianness bugs. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 254–270. Springer International Publishing, Cham (2020)
23. Mahameed, S.: Mellanox, mlx5 rdma net device support (2017), `https://lwn.net/Articles/720074/`
24. Mellanox Technologies: mlx5 core library (2020), `https://github.com/torvalds/linux/tree/master/drivers/net/ethernet/mellanox/mlx5/core`
25. Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: Proc. of the ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'06). pp. 54–63. ACM (June 2006)
26. Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. In: Proc. of the 7th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'06). LNCS, vol. 3855, pp. 348–363. Springer (Jan 2006)
27. Miné, A.: Abstract domains for bit-level machine integer and floating-point operations. In: Proc. of the 4th Int. Workshop on Invariant Generation (WING'12). p. 16. No. HW-MACS-TR-0097, Computer Science, School of Mathematical and Computer Science, Heriot-Watt University, UK (Jun 2012)
28. Miné, A.: Static analysis by abstract interpretation of concurrent programs. Tech. rep., École normale supérieure (May 2013)
29. Miné, A., Delmas, D.: Towards an industrial use of sound static analysis for the verification of concurrent embedded avionics software. In: Proc. of the 15th International Conference on Embedded Software (EMSOFT'15). pp. 65–74. IEEE CS Press (Oct 2015)
30. Miné, A., Ouadjaout, A., Journault, M.: Design of a Modular Platform for Static Analysis. In: The Ninth Workshop on Tools for Automatic Program Analysis (TAPAS'18). Fribourg-en-Brisgau, Germany (Aug 2018), `https://hal.sorbonne-universite.fr/hal-01870001`
31. Nita, M., Grossman, D.: Automatic transformation of bit-level C code to support multiple equivalent data layouts. In: Hendren, L.J. (ed.) Compiler Construction, 17th International Conference, CC 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. 4959, pp. 85–99. Springer (2008)
32. Red Hat, Inc.: Generic network virtualization encapsulation (2017), `https://github.com/torvalds/linux/blob/master/drivers/net/geneve.c`

33. Souyris, J., Wiels, V., Delmas, D., Delseny, H.: Formal verification of avionics software products. pp. 532–546 (2009)
34. The LineageOS Project: Lineageos (2020), `https://github.com/LineageOS/`
35. The Squashfs Project: Squashfs (2020), `https://github.com/LineageOS/android_kernel_sony_msm8960t/tree/lineage-18.1/fs/squashfs`