Towards Removing Undef Values from LLVM IR

PEDRO LOBO, INESC-ID, Portugal and Instituto Superior Técnico - University of Lisbon, Portugal
JOHN MCIVER, Virginia Tech, USA
GEORGE MITENKOV, Aptos Labs, United Kingdom
JUNEYOUNG LEE, AWS, USA
KIRSHANTHAN SUNDARARAJAH, Virginia Tech, USA
NUNO P. LOPES, INESC-ID, Portugal and Instituto Superior Técnico - University of Lisbon, Portugal

LLVM's intermediate representation (IR) has two deferred undefined behavior (UB) values: undef and poison. The existence of these two values has been a persistent source of bugs. Reasoning about the correctness of analyses and optimizations for the regular cases is already tricky; ensuring that these are sound for all UB cases is highly non-trivial.

Undef values, in particular, are one of the most misunderstood concepts of LLVM IR. On paper, the definition is simple: they represent an arbitrary value of the underlying type, and can yield a different value each time they are observed. However, this property makes even simple algebraic rewrites, such as replacing $2 \times y$ with $y + y$, unsound in LLVM.

Because reasoning about undef is hard, and the benefits of having it are limited, we have set a roadmap to eliminate it altogether from LLVM IR. The last remaining use of undef is the value of uninitialized memory, which has implications on the lowering of bitfields, as well as raw data copies and comparisons.

In this paper, we propose an extension to LLVM IR that includes a raw memory value type and a freezing load. We show that these two constructs are sufficient to replace the remaining uses of undef in LLVM. Our implementation shows that these changes have minimal impact on both run-time and compile-time performance. By removing the final hurdle to eliminating undef from LLVM IR, this work paves the way for a simpler semantic model and easier reasoning about the soundness of IR analyses and optimizations.

CCS Concepts: • **Software and its engineering → Compilers**; **Semantics**.

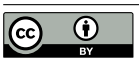
Additional Key Words and Phrases: Undefined Behavior, Compiler Optimizations, LLVM, IR Semantics

ACM Reference Format:
Pedro Lobo, John McIver, George Mitenkov, Juneyoung Lee, Kirshanthan Sundararajah, and Nuno P. Lopes. 2026. Towards Removing Undef Values from LLVM IR. *Proc. ACM Program. Lang.* 10, PLDI, Article 172 (June 2026), 24 pages. https://doi.org/10.1145/3808250

1 Introduction

In the past decade, several research groups have worked on formalizing the LLVM intermediate representation (IR) [2, 9, 21, 30, 32, 36, 56, 58], as well as building tools to automate the verification of analyses and optimizations [12, 31, 37, 38, 48]. Together, these efforts have helped reduce miscompilations, and contributed to a growing body of knowledge on the design of compiler IRs.

Authors' Contact Information: Pedro Lobo, INESC-ID, Portugal and Instituto Superior Técnico - University of Lisbon, Portugal, pedro.lobo@tecnico.ulisboa.pt; John McIver, Virginia Tech, USA, jmciver@vt.edu; George Mitenkov, Aptos Labs, United Kingdom, george@aptoslabs.com; Juneyoung Lee, AWS, USA, lebjuney@amazon.com; Kirshanthan Sundararajah, Virginia Tech, USA, kirshanthans@vt.edu; Nuno P. Lopes, INESC-ID, Portugal and Instituto Superior Técnico - University of Lisbon, Portugal, nuno.lopes@tecnico.ulisboa.pt.



ACM 2475-1421/2026/6-ART172
https://doi.org/10.1145/3808250

The IR is the most important data structure in a compiler, as it represents the program being compiled. Every component interacts with it: language frontends generate IR, analyses prove facts about IR programs, optimizations rewrite IR, and backends transform IR into assembly. Therefore, the design of the IR directly affects compile time, compiler memory usage, the efficiency of generated binaries, the speed of compiler development, the set of supported optimizations, and the range of programming languages and hardware architectures that can be efficiently supported.

For example, the SSA form [8, 11, 45] was adopted by several mainstream compilers (e.g., gcc, LLVM) in the 2000s. The key idea is that in an SSA-based IR, each register is written only once. This essentially caches the result of the analysis that identifies the instruction that last modified each register (e.g., reaching definitions). This seemingly simple concept makes optimizations easier and less error-prone to develop. By enforcing the correctness of this analysis *syntactically*, SSA eliminates an entire class of compiler bugs in an efficient way. Moreover, since most optimizations would otherwise need to perform this analysis, SSA also often improves compile time.

Besides SSA, other high-profile developments that have been adopted recently include MemorySSA [10, 42], which treats memory functionally, just as SSA does for registers (each store takes a memory state as input and produces a new one, thereby caching alias analysis), and SSI [1, 7], which makes control-flow-sensitive analyses sparse and therefore more efficient.

One active research area in IR design concerns undefined behavior (UB). Since compilers usually support multiple source languages and target architectures, the IR must reconcile differing semantics. For example, some programming languages define integer overflow as raising an exception, while others offer wraparound semantics. Similarly, the behavior of shifts by out-of-range amounts differs across hardware architectures (e.g., ARM vs x86). One option is to fix the IR semantics to a particular choice, thereby burdening frontends and backends that require different semantics. Another option is to leave certain cases unspecified, a strategy formalized as undefined behavior (UB).

LLVM has two forms of UB: immediate and deferred. Immediate UB is reserved for operations that trap on common hardware architectures and therefore cannot be executed speculatively (e.g., division by zero or loads from out-of-bounds addresses). Deferred UB covers the remaining cases, including corner cases where different ISAs diverge (e.g., overflowing shifts) or for which different languages offer different semantics (e.g., integer overflows). Operations with only deferred UB can be executed speculatively and are thus much easier for optimizers to reorder than operations that may trigger immediate UB.

LLVM further splits deferred UB into three values: `poison`, `undef`, and fixed non-deterministic (produced by the `freeze` instruction). As you might already imagine, LLVM's value lattice is quite complex. But there is more: `undef` values can yield a different result every time they are observed. For example, $x + x$ is not guaranteed to be even in LLVM. If x is `undef`, the left-hand operand might evaluate to 1 while the right-hand operand evaluates to 0, resulting in 1. This behavior is far from intuitive and often confuses compiler developers, most of whom are not experts in IR semantics.

Because of the non-intuitive semantics of `undef`, which makes many innocent-looking optimizations unsound, `undef` has been a persistent source of bugs in LLVM [28, 47, 55, 61].[1] To address this problem, the community has been systematically replacing uses of `undef` values with `poison` values and `freeze` instructions where possible. From LLVM 10 to 21 (spanning 5 years), 302 (58%) of the uses of `undef` in LLVM's code base have been eliminated, as well as 51 (66%) of the cases in Clang.

For example, `undef` was often used as a placeholder for values that would be removed later, and for the initialization of vectors (which are usually initialized element by element, starting from an

[1]At the time of writing, Alive2 finds 136 miscompilations when run on LLVM's test suite. If `undef` was removed, 25 of these (18%) would automatically become correct (i.e., these optimizations are unsound just because `undef` exists; they do not manipulate `undef` explicitly).

arbitrary vector). Both of these uses were trivially replaced with `poison` values without any semantic changes. However, we have now reached a plateau: most of the remaining `undef` values arise from loads of uninitialized memory, which cannot be eliminated without further extending LLVM IR.

In this paper, we introduce two extensions to LLVM IR that allow us to change the value of uninitialized memory to `poison`, thus eliminating the final obstacle to removing `undef` entirely. Furthermore, our proposal remains backward compatible with older IR.

The first extension is a new type to represent raw memory data in IR registers, which we call byte type (inspired by C++'s `std::byte` type). Even with uninitialized memory being `undef`, LLVM IR is already not sufficiently expressive to represent functions like `memcpy`. These functions are currently lowered into memory operations over integers, which is unsound because if the memory contains a pointer, it loses its provenance. Changing the value of uninitialized memory to `poison` complicates this further because we need to be able to copy `poison` at a bit level, which is currently not possible. With our extension, a call to `memcpy` can be safely lowered into operations over bytes:

```
memcpy(dst, src, 8);
```

```
%v = load b64, ptr %src
store b64 %v, ptr %dst
```

The second extension is a freezing load: a load that freezes the loaded raw data before converting it into the requested type. The freeze instruction converts `undef` and `poison` bits into fixed, non-deterministic bits. If we load, for example, a 32-bit integer and then freeze it, we lose the bits that were not `poison`. This is because `poison` is a value-wise property at the IR register level, but a bit-wise property at the raw memory level.[2] Freezing raw bits directly allows us to preserve the non-poison bits while replacing the poison ones. The following example illustrates the difference:

```
%w = load i32, ptr %p
%v = freeze i32 %w
```

```
%w = load b32, ptr %p
%w2 = freeze b32 %w
%v  = bytecast b32 %w2 to i32
```

```
%v = load i32, ptr %p, !freeze
```

On the left, we first load a 32-bit integer and then freeze it. If any of the bits in memory is `poison`, the entire `%w` becomes `poison`, and `%v` becomes a fully non-deterministic value. In the middle, we load a 32-bit raw value and freeze it. Because the byte type represents `poison` bit-wise, only the `poison` bits are affected by the freeze instruction. Finally, the raw value is converted into an integer. On the right, we show our new freezing load, which is syntactic sugar for the code in the middle.

To summarize, the contributions of this paper are as follows:

(1) An extension to LLVM IR that enables replacing the value of uninitialized memory from `undef` to `poison`, eliminating the last major source of `undef`. We also show that older IR files can be automatically upgraded to be compatible with a refinement of the old semantics.
(2) An implementation of this extension in LLVM and Clang, showing that it can be implemented with reasonable engineering effort and without requiring a complete rewrite of the compiler.
(3) An implementation of the extension in the Alive2 translation validation tool, showing that optimization verification time is not adversely affected.
(4) An evaluation of the proposed extension, showing that it fixes known miscompilations while remaining mostly neutral with respect to the generated code in terms of size, performance, and compilation time.

[2]IR registers are either fully poison or not poison. The main reason for this design is that it is substantially easier to reason about than a per-bit model. For example, defining how poison bits propagate through arithmetic and bitwise operations in a consistent and developer-friendly way would be highly non-trivial. At the memory level, however, per-bit poison granularity is acceptable, as it applies to data at rest rather than to values under computation.

2 Background on LLVM IR

In this section, we give a brief overview of LLVM IR, focusing on the relevant parts for this paper.

LLVM IR is an SSA-based representation that is reasonably low-level, close to an abstract assembly. It supports all the basic arithmetic operations, memory load/store, function calls, stack allocation, and branching (goto). Registers can have one of the following types, each of a particular bitwidth: integer, floating-point, pointer, array, vector, or structure.

Below we show an example C function on the left and the corresponding LLVM IR on the right:

```c
int f(int a, float *p) {
  int c;
  if (a > 0) {
    c = a + 2;
  } else {
    c = 3;
    *p = *p + 2.0;
  }
  return c;
}
```

```
define i32 @f(i32 %a, ptr %p) {
  %cmp = icmp sgt i32 %a, 0              ; signed int comparison
  br i1 %cmp, label %then, label %else   ; conditional branch
then:                                    ; basic block name
  %c1 = add nsw i32 %a, 2                ; integer addition
  br label %end
else:
  %v = load float, ptr %p                ; load a float
  %add = fadd float %v, 2.0              ; floating-point addition
  store float %add, ptr %p               ; store a float
  br label %end
end:
  %c = phi i32 [ %c1, %then ], [ 3, %else ]
  ret i32 %c
}
```

Focusing on the function in LLVM IR, it begins by checking if the first argument is greater than zero. If so, we jump to the `then` basic block (BB). The addition in this BB carries the **`nsw`** (no signed wrap) attribute, reflecting the semantics of C, which specifies that signed integer overflow is undefined behavior. Next, on the `else` basic block, we load a float from memory, add 2.0, and store back the result. Note that the statement `c = 3;` does not appear in this basic block. Finally, in the `end` BB, we encounter a **`phi`** instruction, which is the SSA construct used to merge assignments to the same register across different basic blocks (working around the restriction of a single assignment per register). Each entry corresponds to a predecessor and can be either a register or a constant, which explains why assignments of constants to local variables may "disappear" into a **`phi`**.

2.1 Undefined Behavior

LLVM supports multiple programming languages and hardware architectures. To accommodate the semantic differences between these, LLVM makes heavy use of undefined behavior (UB) to leave certain behaviors unspecified rather than enforcing a particular one.

Although UB in programming languages has earned a bad reputation because of security hazards and program crashes [14, 44, 51–53], UB in IRs is a different story. In IRs, UB serves three purposes: (1) it allows IR designers to offer only the least common denominator semantics across the languages and hardware of interest, (2) it allows frontends to efficiently convey assumptions to optimizers, and (3) it supports the efficient caching of facts inferred by analyses. Notably, the safer a programming language is, the more assumptions optimizers can make about the code (e.g., in a memory-safe language, all memory accesses are guaranteed to be in bounds by some mechanism).

LLVM distinguishes between two kinds of UB: immediate, which is reserved for operations that can trap the CPU (e.g., division by zero, dereference a null pointer), and deferred, which covers the remaining cases. Deferred UB is subdivided into **`undef`**, **`poison`**, and fixed non-deterministic values.

Fig. 1 shows LLVM's value lattice. Values are pairs $(\mathcal{S}, \tau)$ where $\mathcal{S} \in \mathcal{P}(\{0,1\}^w) \setminus \{\emptyset\}$ and $\tau \in \{S, U\}$ (stable/unstable). The refinement order combines two dimensions: subset inclusion on

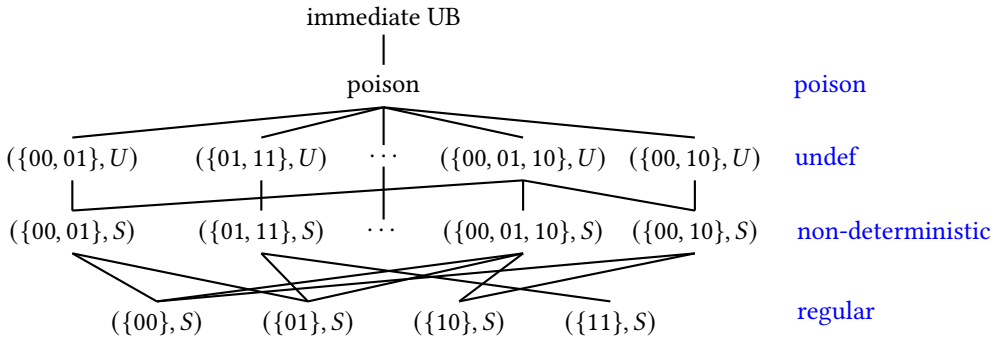


Fig. 1. Lattice of LLVM values for a 2-bit integer type. Values are pairs $(\mathcal{S},\tau)$ where $\mathcal{S}\in\mathcal{P}(\{0,1\}^w)\setminus\{\emptyset\}$ is a non-empty set of possible bit-vector values and $\tau\in\{S,U\}$ is a stability tag (stable/unstable). Immediate UB is a computation-level property projected onto the value domain.

$\mathcal{S}$, and the tag ordering $S <_\tau U$, since per-use observation (U) is strictly more non-deterministic than consistent observation (S). The stability tag captures how non-determinism is resolved across uses: a *stable* value commits to one element of $\mathcal{S}$ observed consistently at every use, corresponding to `freeze` and concrete values; an *unstable* value independently samples a fresh element of $\mathcal{S}$ at each use, corresponding to `undef`. Singleton stable values $(\{v\},S)$ are the concrete regular values at the bottom of the lattice. Immediate UB sits at the top as the image of machine-level UB projected onto the value domain, and refines to any value.

We will now focus on deferred UB, which is what is relevant for this paper. The code snippet below exemplifies how poison works. The first instruction produces a poison value due to an overflow. Poison then propagates throughout the expression DAG, similar to floating-point NaNs. The `freeze` instruction is one of the few that can stop the propagation of poison by converting it into a fixed non-deterministic value.

```
%add = add nuw i16 1, 65535   ; unsigned overflow -> poison
%sub = sub i16 %add, 1        ; still poison
%nd  = freeze i16 %sub        ; a fully non-deterministic number (nnnn.nnnn.nnnn.nnnn)
%nd2 = add i16 %nd, %nd       ; a non-deterministic even number  (nnnn.nnnn.nnnn.nnn0)
```

Going further down the lattice, we encounter undef values. Nowadays, they mostly arise when reading uninitialized memory. In the example below, both `%a` and `%b` are undef. This is because an undef value can yield a different value each time it is observed. Since adding two arbitrary numbers yields an arbitrary number, adding two undefs also produces undef. The last two instructions show that undef and non-deterministic values are not sticky like poison: individual bits can be fixed to particular values while others remain undef or non-deterministic (e.g., $0 \le \%d \le 3$).

```
%p = alloca i16         ; allocate a 2-byte value on the stack
%a = load i16, ptr %p   ; load uninitialized memory: undef
%b = add i16 %a, %a     ; still undef (even or odd number)
%c = and i16 %b, 3      ; a partial undef             (0000.0000.0000.00uu)
%d = freeze i16 %c      ; a partial non-deterministic (0000.0000.0000.00nn)
```

A consequence of the semantics of undef is that it is unsound to replace $y \times 2$ with $y + y$. It is sound, however, to rewrite both expressions as a left shift ($y \ll 1$). Eliminating `undef` from LLVM IR would make these three expressions equivalent, thereby making the semantics more intuitive.

The motivation for introducing `undef` in LLVM IR was to represent the value of uninitialized memory. When generating SSA, it is common to encounter `phi` nodes whose operands correspond to values that may be uninitialized. Assigning such values a fixed constant (e.g., zero) would penalize

performance and binary size, so LLVM used `undef` instead, enabling simplifications such as rewriting `phi(x, undef)` to `x`. Correct programs, such as the one below, can produce such `phi` instructions.

```
int x;
if (y > 0) x = z;
// x uninitialized if y <= 0
foo();
if (y > 0) bar(x);
```

```
    %x = phi i32 [%z, %then], [undef, %entry] ; optimized to %z
    call void @foo()
    br i1 %cmp, label %bb, label %exit ; jump to bb if y > 0

bb: call void @bar(i32 %x)   ; optimized to @bar(i32 %z)
```

However, after the introduction of `poison`, this reasoning became invalid. Because `poison` is stronger (UB-wise) than `undef`, replacing `undef` with an arbitrary value `x` is unsound if `x` might be `poison`. Other optimizations, such as removing stores of `undef`, are similarly unsound. LLVM continues to apply these optimizations for performance reasons, at the cost of potential miscompilations. Changing the value of uninitialized memory from `undef` to `poison` would resolve these issues.

2.2 Bitfields

Bitfields are a C/C++ construct that allow multiple structure fields to be packed into the same byte, avoiding the usual padding between fields. For example, the program below packs three fields into a 32-bit integer. Because the field widths are not aligned to byte boundaries, and LLVM IR supports memory accesses only at byte granularity, Clang compiles bitfields using bitwise operations.

```
struct S {
  int a:3;
  int b:5;
  int c:24;
};

int f() {
  S s;
  s.a = 3;
  return s.a + 1;
}
```

```
define i32 @f() {
  %s = alloca i32
  %bf.load = load i32, ptr %s       ; undef
  %bf.clear = and i32 %bf.load, -8  ; clear last 3 bits (uu..uu000)
  %bf.set = or i32 %bf.clear, 3     ; uu..uu011
  store i32 %bf.set, ptr %s         ; store s

  %bf.load2 = load i32, ptr %s      ; load s (uu..uu011)
  %bf.shl = shl i32 %bf.load2, 29
  %bf.shr = ashr i32 %bf.shl, 29    ; s.a  (00..00011)

  %add = add nsw i32 %bf.shr, 1
  ret i32 %add
}
```

Loading a bitfield involves first loading the entire word and then masking out the other fields using a left shift (`shl`) followed by an arithmetic right shift (`ashr`). Writing a bitfield requires loading the entire word, clearing the bits corresponding to the target field (`%bf.clear`), setting the relevant bits with a bitwise OR, and finally storing the updated word back to memory.

The main justification for the correctness of this encoding is that loads from uninitialized memory produce an `undef` value. When writing a bitfield for the first time, we first load its corresponding word, which is uninitialized. Because undef bits are not sticky (unlike poison), we can safely set the relevant bits and store the updated word back to memory.

Clang supports an alternative bitfield encoding (`-ffine-grained-bitfield-accesses`), which reduces the bitwidth of the memory operations whenever possible. For our example, this produces the following IR, where memory accesses operate on a single byte instead of four:

```
%bf.load = load i8, ptr %s        ; load s.{a,b}
%bf.clear = and i8 %bf.load, -8   ; clear s.a, keep s.b
%bf.set = or i8 %bf.clear, 3      ; set s.a
store i8 %bf.set, ptr %s          ; store s.{a,b}
```

These encodings are ABI-compatible, meaning it is safe to link two files compiled with different encodings that access the same bitfields. In a model where uninitialized memory is **undef**, the encodings are effectively equivalent. Although the default encoding may write more bytes, it only modifies the bits corresponding to the changed bitfield, while the remaining bits are stored back unmodified.

Changing uninitialized memory to become **poison** requires modifying first the encoding of bitfields because poison is sticky and thus the trick to mask out bits would no longer work. One option is to freeze the loaded value, turning poison into a non-deterministic value, and then do the usual masking. The caveat of freezing the loaded value is that the two encodings become incompatible. To illustrate the problem, consider a program that first executes the assignment `s.b = 2;` compiled with the fine-grained encoding (shown on the left), and then executes the assignment `s.a = 3;` compiled with the default encoding (shown on the right):

```
%bf.load = load i8, ptr %s      ; poison
%bf.fr = freeze i8 %bf.load     ; nnnn.nnnn
%bf.clear = and i8 %bf.fr, 7    ; 0000.0nnn
%bf.set = or i8 %bf.clear, 16   ; 0001.0nnn
store i8 %bf.set, ptr %s        ; store s.{a,b}
; %p is now: 00010nnn . poison . poison . poison
```

```
%bf.load = load i32, ptr %s     ; poison
%bf.fr = freeze i32 %bf.load    ; nn..nn
%bf.clear = and i32 %bf.fr, -8  ; nn..nn000
%bf.set = or i32 %bf.clear, 3   ; nn..nn011
store i32 %bf.set, ptr %s       ; store s
; %p is now: nnnn.n011 . nn...nn
```

Assume these assignments are the first writes to any of the structure's bitfields. Because the first assignment is compiled using a single-byte store, three bytes remain poison (corresponding to bitfield `c`), while bitfield `a` is initialized to a non-deterministic value. The second assignment is compiled using the default encoding, which first loads all three bitfields. Since at least one of the bits is poison, the entire loaded value becomes poison. The subsequent freeze comes in too late; it can only replace poison with a fully non-deterministic value. At this point, the value of bitfield `b` is lost, and after the store, both `b` and `c` hold non-deterministic values.

2.3 Pointer Provenance

A key aspect of an IR's memory model is how pointers are represented. LLVM adopts a logical model, in which a pointer is represented as a pair of an object identifier and an offset within that object. Each allocation creates a new object. Pointer arithmetic operations can only modify the offset, not the identifier, and thus they are said to propagate *provenance*. This model contrasts with the typical Von Neumann hardware model, where memory is a single flat address space and pointers are plain integers that can point anywhere within it.

The main goal of logical memory models is to simplify alias analysis. For example, in the code below, we can assume that `pi` and `qj` do not alias, irrespective of the values of `i` and `j`. Even if, at run time, they might point to the same memory location, the memory model disallows a pointer from one object accessing another object. Such cases are therefore defined as UB.

```
char *p = malloc(4);  // (obj1, 0)
char *q = malloc(4);  // (obj2, 0)
char *pi = p + i;     // (obj1, i)
char *qj = q + j;     // (obj2, j)
*pi = 1;              // UB if (intptr_t)pi == (intptr_t)qj
*qj = 2;              // thus we can assume they write to different locations
```

A consequence of this model is that pointers are not merely integers representing memory addresses. This, in turn, requires the compiler to avoid accidental conversions of pointers into integers. Although pointers can be safely converted to integers and back, this must be done explicitly using the appropriate instructions (**ptrtoint** and **inttoptr**). Implicit casts (e.g., storing an integer and later loading it as a pointer) produce pointers without provenance, which cannot be dereferenced.

2.4 Raw Memory Copies

Both C[3] and C++[4] define types that can hold values of any other type: (1) `char`, which is an integer type (typically 8 bits) that also serves as a universal holder, and (2) `std::byte`, introduced in C++17 to improve safety by providing a type that is purely a value holder, for which arithmetic operations are disallowed (only copies, comparisons, and casts are permitted).

These types enable the implementation of memory-related functions, such as `memcpy`, directly in C/C++, allowing them to operate on raw data regardless of the underlying stored type. An important consequence is that `char*` pointers can alias with any other pointer type, which limits the effectiveness of type-based alias analysis.

In LLVM IR, `memcpy` is represented either as an intrinsic call or as a sequence of load/store instructions when the number of bytes to copy is small:

```
; copies 4 bytes
call void @llvm.memcpy(ptr %dst, ptr %src, i64 4, i1 false)
```

```
%v = load i32, %src
store i32 %v, ptr %dst
```

The advantage of using an intrinsic is that its semantics can be defined precisely; in this case, that `memcpy` performs a raw memory copy. However, intrinsics are often treated as black boxes by optimizations. Exposing the copy as explicit load/store instructions in the IR enables optimizations to propagate the stores or even eliminate them when possible.

The code on the right shows the lowered form of `memcpy` using a 32-bit integer to hold the value being copied. This lowering is unsound because, as previously discussed, pointers are not merely integers: they carry provenance information, which is lost when a pointer is represented as an integer. Moreover, because poison is a value-wise property, a single poison bit would contaminate all four bytes, whereas the semantics of `memcpy` require that only the affected bit be poison. LLVM currently transforms the IR on the left into the form on the right, but this is unsound.

3 Extending LLVM IR to Reduce Reliance on Undef Values

In this section, we propose two extensions to LLVM IR that lay the groundwork for the eventual removal of undef values. In particular, we change the value of uninitialized memory from undef to poison. Our extensions address the two core challenges identified earlier that must be resolved before this transition: preserving the correctness of bitfield encodings, and ensuring that memory operations such as `memcpy` remain sound, maintaining pointer provenance across copies and propagating poison on a per-bit basis.

While the current lowering of `memcpy` is already unsound with uninitialized memory being undef, changing it to poison without addressing the underlying issue would exacerbate the problem, as the proportion of poison bytes in memory would increase. We anticipated that miscompilations in practice would rise if the issue was not addressed, and in this work, we provide a correct solution.

3.1 Preliminaries

Without loss of generality, in this paper we adopt a simplified abstract machine model of LLVM IR, derived from the full model of Lee et al. [30]. We assume there are only two types: integers and 64-bit pointers, and we already assume there are no undef values, only poison. These simplifications allow us to focus on the semantics most relevant to our proposal.

[3] *Values stored in non-bit-field objects of any other object type consist of n x* `CHAR_BIT` *bits, where* `n` *is the size of an object of that type, in bytes. The value may be copied into an object of type* `unsigned char [n]` *(e.g., by* `memcpy`*); the resulting set of bytes is called the object representation of the value.* (C99 Standard, 6.2.6.1.4)

[4] *The underlying bytes making up the object can be copied into an array of* `char`, `unsigned char`, *or* `std::byte`. *If the content of that array is copied back into the object, the object shall subsequently hold its original value.* (C++23 Standard, 6.8.1.2)

$$
\begin{array}{lll}
\text{Num}(w) & ::= & \{\, i \mid 0 \le i < 2^w \,\} \\
\text{BlockID} & ::= & \mathbb{N} \\
\text{Mem} & ::= & \text{BlockID} \nrightarrow \text{Block} \\
\text{Block} & ::= & \{\, (n, c) \mid n \in \mathbb{N} \wedge c \in [\![\mathbf{b}8]\!] \,\} \\
\text{Pointer} & ::= & \{\, (id, o) \mid id \in \text{BlockID} \wedge o \in \text{Num}(64) \,\} \\
[\![\mathbf{i}w]\!] & ::= & \text{Num}(w) \uplus \{\, \textbf{poison} \,\} \\
[\![\mathbf{ptr}]\!] & ::= & \text{Pointer} \uplus \{\, \textbf{poison} \,\} \\
\text{Bit} & ::= & \{\, \text{Int}(n) \mid n \in \{0, 1\} \,\} \uplus \{\, \text{Ptr}(p, i) \mid p \in \text{Pointer} \wedge (0 \le i < 64) \,\} \uplus \{\, \textbf{poison} \,\} \\
[\![\mathbf{b}w]\!] & ::= & \text{Bit}^w \\
\text{Name} & ::= & \{\, \%\text{x}, \%\text{y}, \ldots \,\} \\
\text{Reg} & ::= & \text{Name} \to \{\, (ty, v) \mid v \in [\![ty]\!] \,\}
\end{array}
$$

Fig. 2. Definitions. The set of all possible values of a type ty is given by $[\![ty]\!]$.

Fig. 2 shows the definitions of our model. Let memory (Mem) be a partial function from block identifiers to blocks, and let block be a pair consisting of its size and contents (a sequence of bytes). Each allocation, including calls to allocation functions or the creation of stack or global variables, generates a fresh block initialized to poison. A pointer is represented as a pair consisting of a block identifier and an offset within that block. Pointer arithmetic operations modify only the offset.

The set of all possible values of an integer of bitwidth w is given by $[\![\mathbf{i}w]\!]$, which also includes the poison value. A bit is either a 1-bit integer or a pointer bit (represented as a pair of a pointer and an offset within the pointer's address). For example, when a pointer p is stored to memory, the abstract model does 64 stores of $(p, 0), \ldots, (p, 63)$, one for each bit of the pointer's address.

The register map (Reg) is a function from register names to their type and value. All registers are initialized to poison. The machine state is represented as a pair (Reg, Mem).

3.2 Byte Type: Representing Raw Memory Data in Registers

The first extension we propose is the addition of a raw byte type to the IR, allowing registers to hold raw memory data. This type is written as $\mathbf{b}w$, where w is the bitwidth (not limited to 8 bits!). We also introduce a new instruction, `bytecast`, to convert between bytes and other types, and extend selected existing instructions to support the byte type.

Before introducing the new IR semantics, we first define a few auxiliary functions in Fig. 3. In (a), we define functions for converting values into bytes. (b) and (c) define conversions from a byte into an integer and a pointer, respectively. If any bit does not match the target type, the result is poison. Freezing a bit is a no-op for non-poison values and produces a non-deterministic bit otherwise.

We also define two functions that forcefully convert a byte into an integer or a pointer. These differ from $ty\Uparrow b$ in that they yield poison only if any bit is poison; otherwise, they force the conversion to the desired type. Pointers are converted to integers by extracting the corresponding bits from their address. CastToPtr is more involved: if any two bits belong to different pointers, or any bit is an integer, the type-punning conversion produces a pointer without provenance, represented here with block id 0 (the null block, zero-sized and not dereferenceable).

Fig. 4 shows selected rules of the operational semantics for instructions that manipulate the byte type. Each rule updates the machine state. In this case, only the register map is updated as none of the rules affect memory.

Bitcast is an existing instruction that reinterprets a value as a different type of the same bitwidth, without modifying the underlying bits. We extend it to allow casts *to* the byte type (but not *from* the byte type), and we also extend certain bitwise instructions (truncation and shifts) to support bytes. The freeze instruction is also extended to support bytes, with freezing applied on a per-bit basis. Semantically, bytes are bit sequences indexed from the least significant bit.

$$\begin{array}{lcl} \mathbf{i}w\Downarrow v & = & \lambda i.\,\mathsf{Int}(\mathsf{getbit}(v,i)) \\ \mathbf{ptr}\Downarrow p & = & \lambda i.\,\mathsf{Ptr}(p,i) \end{array}$$

(a) Converting a value to a byte.

$$\mathbf{i}w\Uparrow b = \begin{cases} n & \text{if } \forall_{0\le i<w}.\ \mathsf{Int}(\mathsf{getbit}(n,i)) = b[i] \\ \mathbf{poison} & \text{otherwise} \end{cases}$$

(b) Converting a byte to an integer.

$$\mathbf{ptr}\Uparrow b = \begin{cases} p & \text{if } \forall_{0\le i<64}.\ \mathsf{Ptr}(p,i) = b[i] \\ \mathbf{poison} & \text{otherwise} \end{cases}$$

(c) Converting a byte to a pointer.

$$\mathsf{freeze}(b) = \begin{cases} b & \text{if } b = \mathsf{Ptr}(\_,\_) \vee b = \mathsf{Int}(\_) \\ b' & \text{otherwise, such that } b' \\ & \text{is a non-det non-poison bit} \end{cases}$$

$$\mathsf{CastToInt}(b) = \begin{cases} n & \text{if } \forall_{0\le i<\mathsf{len}(b)}\ \exists_{n'}.\ \big(\mathsf{Int}(n') = b[i] \vee \big(\exists_{j,p}.\ \mathsf{Ptr}(p,j) = b[i] \wedge n' = \mathsf{addr}(p,j)\big)\big) \wedge \\ & \qquad\qquad \mathsf{getbit}(n,i) = n' \\ \mathbf{poison} & \text{otherwise} \end{cases}$$

$$\mathsf{CastToPtr}(b) = \begin{cases} p & \text{if } \forall_{0\le i<64}.\ \ \mathsf{Ptr}(p,i) = b[i] \\ \mathsf{Ptr}(0,\mathsf{n}) & \text{if } \forall_{0\le i<64}\ \exists_{n'}.\ \big(\mathsf{Int}(n') = b[i] \vee \big(\exists_{j,p}.\ \mathsf{Ptr}(p,j) = b[i] \wedge n' = \mathsf{addr}(p,j)\big)\big) \wedge \\ & \qquad\qquad \mathsf{getbit}(n,i) = n' \\ \mathbf{poison} & \text{otherwise} \end{cases}$$

Fig. 3. Auxiliary functions used to define the semantics of the byte type in LLVM IR. Let $\mathsf{len}(b)$ denote the number of bits in a byte b, $b[i]$ the i-th bit of b, $\mathsf{getbit}(n,i)$ the i-th bit of the number n in its binary representation, and $\mathsf{addr}(p,i)$ the i-th bit of the address in binary of pointer p.

$$(\iota = \text{“}r = \textbf{bitcast } ty\ v \textbf{ to b}w\text{”})\quad \textsc{bitcast}\ \frac{v' = ty\Downarrow R[v]}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto v'], M}$$

$$(\iota = \text{“}r = \textbf{bytecast b}w\ v \textbf{ to i}w\text{”})\quad \textsc{bytecast-int}\ \frac{v' = \mathsf{CastToInt}(R[v])}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto v'], M}$$

$$(\iota = \text{“}r = \textbf{bytecast b64 } v \textbf{ to ptr}\text{”})\quad \textsc{bytecast-ptr}\ \frac{v' = \mathsf{CastToPtr}(R[v])}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto v'], M}$$

$$(\iota = \text{“}r = \textbf{bytecast exact b}w\ v \textbf{ to } ty\text{”})\quad \textsc{bytecast-exact}\ \frac{v' = ty\Uparrow R[v]}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto v'], M}$$

$$(\iota = \text{“}r = \textbf{trunc b}w\ v \textbf{ to b}w_2\text{”})\quad \textsc{trunc}\ \frac{v' = (R[v][0], \ldots, R[v][w_2 - 1])}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto v'], M}$$

$$(\iota = \text{“}r = \textbf{lshr b}w\ a,\ b\text{”})\quad \textsc{shift-right}\ \frac{b' = \mathbf{i}w\Uparrow R[b] \qquad b' \neq \mathbf{poison} \qquad 0 \le b' < w \qquad v' = (R[v][b'], \ldots, R[v][w-1], \mathsf{Int}(0), \ldots, \mathsf{Int}(0))}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto v'], M}$$

$$(\iota = \text{“}r = \textbf{freeze b}w\ v\text{”})\quad \textsc{freeze}\ \frac{v' = (\mathsf{freeze}(R[v][0]), \ldots, \mathsf{freeze}(R[v][w-1]))}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto v'], M}$$

Fig. 4. Selected rules of the operational semantics for instructions that manipulate the byte type.

We introduce a new instruction, **bytecast**, to convert bytes into other types. As with **bitcast**, the source and destination types must have the same bitwidth. Two variants are provided: an exact variant and a type-punning variant. The exact variant returns poison if the byte contains any bit whose underlying type differs from the target type.

The type-punning variant returns poison only if any bit is poison; otherwise, it forcefully converts the byte into the requested type. When converting a pointer bit, its semantics resemble the newly introduced **ptrtoaddr** instruction, which converts a pointer into an integer without capturing the pointer itself. Unlike **ptrtoint**, which escapes the pointer (allowing it to be recovered later through

`inttoptr`), `ptrtoaddr` only observes the address. When converting an integer byte to a pointer, the result is a pointer without provenance. For example, it can be used in comparisons but cannot be dereferenced. Specifically, the returned pointer is $(0, v)$, where v is the integer value stored in the byte, and 0 denotes the null block (of zero size).

We also extend shift, truncation, phi, and select instructions to support bytes in addition to integers. The bitwise extensions can be used for extracting parts of bytes, as shown next.

3.3 Applications of the Byte Type

The byte type has several immediate applications, enabling the resolution of long-standing LLVM bugs. As shown in the introduction, it provides a sound basis for lowering raw-copy functions such as `memcpy` and `memmove`. Here, we focus on two additional cases: the correct compilation of C's `char` variables by Clang, and optimizations involving load merging.

In C, `char` variables can hold values of any type and are converted to integers only when used in arithmetic operations. In the middle, we show the incorrect IR currently generated by Clang, which treats `char` as an integer and loses provenance information if it holds part of a pointer. Note how C's semantics require that operations on smaller integer types be promoted to `int` (32 bits in this case). On the right is the corrected IR from our prototype, where `char` values are represented as `b8`.

```
char f(char c) {
  return c + 1;
}
```

```
define i8 @f(i8 %c) {
  %i32 = sext i8 %c to i32
  %add = add nsw i32 %i32, 1
  %r8  = trunc i32 %add to i8
  ret i8 %r8
}
```

```
define b8 @f(b8 %c) {
  %i8   = bytecast b8 %c to i8
  %i32  = sext i8 %i8 to i32
  %add  = add nsw i32 %i32, 1
  %r8   = trunc i32 %add to i8
  %byte = bitcast i8 %r8 to b8
  ret b8 %byte
}
```

Another application of the byte type is making optimizations that merge multiple loads, such as load widening, safe. On the left is a snippet that loads two consecutive integers from memory, `%a` and `%b`. In the middle, we show the IR that LLVM currently produces when merging the two loads: it widens the load, then extracts the values using shifts and truncations. This transformation is unsound if either value might be poison, as a single poison bit taints the entire loaded value. On the right, the IR using the byte type widens the load without spreading poison bits.

```
%a = load i8, ptr %p, align 2
%q = getelementptr i8, ptr %p, i8 1
%b = load i8, ptr %q
```

```
%v = load i16, ptr %p
%a = trunc i16 %v to i8
%s = lshr i16 %v, 8
%b = trunc i16 %s to i8
```

```
%v = load b16, ptr %p
%c = trunc b16 %v to b8
%a = bytecast b8 %c to i8
%s = lshr b16 %v, 8
%d = trunc b16 %s to b8
%b = bytecast b8 %d to i8
```

Merging loads of pointer and non-pointer types from the same address must be done carefully to preserve pointer provenance. The original code snippet is shown on the left, with the code optimized by GVN in the middle. This transformation is unsound, as the resulting pointer may not have sufficient provenance. On the right, we show a correct implementation of this transformation using the byte type. Although the example is artificial, such patterns do occur in practice, for instance across different branches.

```
%a = load i64, ptr %p
%b = load ptr, ptr %p
```

```
%a = load i64, ptr %p
%b = inttoptr i64 %a to ptr
```

```
%v = load b64, ptr %p
%a = bytecast b64 %v to i64
%b = bytecast b64 %v to ptr
```

3.4 Freezing Loads

We extend the `load` instruction with an optional mode to freeze the loaded byte before converting it to the target type: `load i8, ptr %p, !freeze`. This is equivalent to a byte `load`, followed by `freeze` and a `bytecast` to the target type. The purpose of this extension is to prevent a single poison bit from tainting the entire value, while preserving the non-poison bits.

One application of the freezing load is implementing C's bitfields. As discussed in Section 2.2, the first write to a bitfield involves loading uninitialized memory. Since uninitialized memory is now treated as poison, both encodings can be made correct simply by using freezing loads. Bitfield loads with the default encoding also need freezing to remain compatible with the fine-grained encoding.

3.5 Impact of the Proposed Changes

The proposed changes to LLVM IR semantics affect both LLVM developers and end users (e.g., C/C++ developers using Clang). For LLVM developers, the benefits are clear: the value lattice is simplified (one fewer level) and all algebraic equalities in modular arithmetic now hold in LLVM IR. This reduces surprises and the likelihood of introducing unsound transformations in the compiler.

The alternatives to `undef` are `poison` and `freeze poison`. The former is weaker than `undef` and thus theoretically easier to optimize, while the latter yields a fixed non-deterministic value, making it stronger. The key distinction is that each use of `undef` may produce a different value, whereas all uses of a given `freeze poison` observe the same value. This complicates register allocation: registers holding `undef` can be discarded (no need to spill to the stack), whereas registers holding fixed non-deterministic values must be treated like ordinary registers to maintain consistent values, potentially increasing code size and reducing performance. However, reusing registers for `undef` can pose security risks, as register data may leak depending on allocator behavior. Nevertheless, supporting frontends that want to keep using a non-deterministic value for uninitialized memory instead of `poison` becomes more costly in theory.

For C/C++ developers, changing the value of uninitialized memory affects only programs that execute UB; otherwise, it is a no-op. For instance, masking an uninitialized value would previously produce a result with certain fixed bits, whereas with our change it yields `poison`. Some corner cases also change, e.g., evaluating `x > INT_MAX` for an uninitialized `x` now produces `poison` instead of false. Other conditions remain, such as `x == x` potentially evaluating to false if `x` is uninitialized. Because branching on both `undef` and `poison` is UB, the practical impact of these changes is minor: although some expressions now evaluate to `poison`, the outcome at branches remains unchanged.

4 Implementation

We implemented our prototype on top of LLVM 21, extending both the LLVM optimizer and code generation passes, as well as the Clang C/C++ frontend, to support the new IR constructs.

The implementation of the byte type required approximately 2.6k lines of C++ changes across LLVM and Clang,[5] while the `!freeze` extension added about 0.5 k lines.[6] Modifying Clang to compile `char` variables to the new byte type affected roughly 1.9k unit tests, resulting in 387k lines being updated (mostly automated replacements of expected output from `i8` to `b8`).

Our code modifications represent only 0.06% of LLVM/Clang's source code, demonstrating that introducing a new type into LLVM IR can be achieved with minimal impact on the existing codebase.

We implemented the following changes in LLVM and Clang:

(1) Added the byte type and the `bytecast` instruction to the IR, and extended the `trunc`, `lshr`, and `freeze` instructions to operate on byte values.

[5]Code available at https://github.com/pedroclobo/llvm-project/tree/byte-type
[6]Code available at https://github.com/jmciver/llvm-project/tree/users/jmciver/u2p-paper-clang-tests

(2) Added support for the optional `!freeze` modifier in the `load` instruction.
(3) Modified the SROA optimization (which converts stack allocations into registers) to handle `!freeze` correctly. SROA must freeze register uses corresponding to loads with `!freeze`.
(4) Modified Clang to compile `char` variables to the new `b8` type.
(5) Modified Clang to compile bitfield writes using freezing loads. Recall that writing to a bitfield requires first loading the corresponding word and then updating the relevant bits via masking. Therefore, the first write must use a freezing load if uninitialized memory yields `poison`. For simplicity, we apply freezing loads to all bitfield writes.
(6) Modified Clang to compile bitfield reads with freezing loads when using the default word-level encoding. This ensures compatibility with the short encoding, allowing bitfields stored with the short encoding to be safely read using the word-level encoding.
(7) Modified Clang so that none of the loads are freezing, except the case above for bitfields.
(8) Fixed the lowering of `memcpy` and `memmove` to do load/stores of bytes rather than integers.
(9) Added new optimizations for the byte type, including constant folding, store forwarding, SLP vectorization support, redundant cast elimination, and extended the cost model to treat `bytecast` as a zero-cost operation (since it does not generate any assembly code).
(10) Using the byte type, fixed long-standing bugs in LLVM optimizers, namely issues in GVN's load merging mechanism and load widening (cf. details in Section 3.3).
(11) Switched the value of uninitialized memory from `undef` to `poison`, including changing several analyses and optimizations that deal with it (e.g., MemoryBuiltins, GVN, InstCombine).

Let us now give examples of the new optimizations. The first eliminates redundant cast round trips involving the `bytecast` instruction. In the snippet below, all uses of `%c` can be replaced with `%a`.

```
%b = bitcast i16 %a to b16
%c = bytecast b16 %b to i16
```

Another optimization folds a `bytecast` into a preceding load. Below, if `%b` has a single user (`%c`), the code on the left is replaced with the one on the right. Under the current semantics, where type punning through a load yields `poison`, this optimization is sound only for the `exact` variant.

```
%b = load b8, ptr %p                    |   %c = load i8, ptr %p
%c = bytecast exact b8 %b to i8
```

Closely related is the store forwarding optimization, where a byte store (below, on the left) can be forwarded to a load of any type via a `bytecast` (on the right). Similarly, a store of a non-byte type can be forwarded to a byte `load` using a `bitcast`.

```
store b32 %x, ptr %p                    |   store b32 %x, ptr %p
%v = load i32, ptr %p                   |   %v = bytecast b32 %x to i32
```

The new `bytecast` instruction also supports vector operands. We extended the SLP optimization (straight-line vectorizer) to vectorize `bytecast` instructions. In the example below, the code on the left loads two bytes and assembles them into a vector, while the optimized version on the right performs a single vector load followed by a vector `bytecast` operation.

```
%p1 = getelementptr inbounds b8, ptr %p0, i64 1
%b0 = load b8, ptr %p0
%b1 = load b8, ptr %p1
%x0 = bytecast b8 %b0 to i8
%x1 = bytecast b8 %b1 to i8
%v0 = insertelement <2 x i8> poison, i8 %x0, i32 0
%r  = insertelement <2 x i8> %v0, i8 %x1, i32 1
```

```
%l = load <2 x b8>, ptr %p0
%r = bytecast <2 x b8> %l to <2 x i8>
```

An alternative compilation strategy for stack-allocated bitfields is to store `freeze poison` immediately after allocation, rather than using freezing loads for writes, since such bitfields would then be guaranteed to be initialized. Likewise, for local bitfields that do not escape, it is guaranteed that a single compilation strategy is used, eliminating the need to emit freezing loads for reads. We did not implement either optimization, as bitfield performance did not appear relevant for the benchmarks we used.

4.1 Backwards Compatibility

LLVM is a large system, so changing the semantics of uninitialized memory and eventually removing `undef` values completely requires careful planning to avoid breaking existing projects. To preserve backward compatibility, updates are needed both in LLVM's source code (so that instruction creation functions used by frontends remain compatible without having to change all frontends at once), and in the IR auto-upgrade mechanism, which automatically upgrades bitcode files produced by older LLVM versions when they are loaded.

Creation of `undef` values is replaced with either `freeze poison` or the zero constant (when `freeze` cannot be used, as in global variable initializers that only accept constants), since both are refinements of `undef`. Additionally, `load` instructions are changed to freezing loads, which is also a refinement: uninitialized memory and poison will yield a non-deterministic value instead of `undef`, while regular values remain unchanged.

Over time, frontends can migrate to non-freezing loads and to using `poison` in place of `undef` where possible. In our prototype, only Clang (LLVM's C/C++ frontend) was modified to use non-freezing loads, except for bitfields.

4.2 Extending Alive2 With Bytes

Alive2 [37] already includes an internal byte type to represent memory contents (memory is modeled as a set of SMT arrays from object index to byte). Implementing our extensions primarily involved exposing this internal representation for use by LLVM IR registers. The non-exact `bytecast` reused the type punning functionalities of the assembly mode [3].

The main complication arose from Alive2's internal optimizations, in particular partial order reduction. Its preprocessor analyzes each function before verification and, among other things, attempts to reduce bitwidths (e.g., minimizing the number of bits used for pointer offsets), and determines the optimal byte size (e.g., using 2-byte addressable memory if only 2-byte aligned accesses occur). Some of these optimizations rely on assumptions that are no longer sound when instructions over the new byte type are present.

Since these optimizations are crucial for Alive2's performance, we updated them rather than disabling them. Two optimizations in particular required changes.

The first simplifies memory refinement when no pointer stores occur in the function: pointer bytes are compared using SMT expression equality rather than full refinement (which is much more expensive), since any pointer byte in memory must have been stored before function entry. With the byte type, programs can now create pointer bytes via, e.g., a `bitcast` of a pointer to a byte. The preprocessor was updated to account for these new ways of creating pointer bytes.

The second optimization applies when the function only performs pointer-type memory accesses. Alive2 previously assumed all bytes contained pointers, as integer and poison bytes are indistinguishable when loaded as pointers (they both produce poison). However, the non-exact `bytecast` introduces a new way to observe byte values with type punning. Consequently, this optimization is disabled whenever such instructions are present.

Table 1. Benchmark applications, including their category, number of lines of C/C++ code (kLoC), number of bitfield variable declarations, and number of static bitfield reads and writes. The version number refers to the Phoronix benchmark version.

No	Benchmark	Category	kLoC	Bitfields	Bitfield reads/writes
1	aircrack-ng-1.3.0	Security	67	–	–
2	botan-1.6.0	Security	148	–	–
3	build-llvm-1.5.0	Compiler	2,209	697	2,718 / 1,426
4	compress-7zip-1.11.0	Compression	247	–	–
5	compress-zstd-1.6.0	Compression	90	1	1 / 2
6	draco-1.6.1	Texture Processing	50	1	2 / 3
7	encode-flac-1.8.1	Audio Compression	63	2	34 / 18
8	espeak-1.7.0	Speech Synthesizer	45	–	–
9	graphics-magick-2.2.0	Image Processing	267	9	9 / 9
10	john-the-ripper-1.8.0	Security	315	–	–
11	jpegxl-1.6.0	Image Compression	137	2	28 / 5
12	luajit-1.1.0	Compiler	69	–	–
13	mafft-1.6.2	HPC	94	–	–
14	ngspice-1.0.0	Circuit Simulation	528	11,106	12,517 / 11,329
15	openssl-3.0.1	Security	598	99	275 / 182
16	rnnoise-1.1.0	Audio Processing	147	–	–
17	simdjson-2.0.1	Parallel Processing	74	9	26 / 31
18	sqlite-speedtest-1.0.1	Database	251	63	242 / 147
19	tjbench-1.2.0	Parallel Processing	57	–	–
20	z3-1.0.1	SMT Solver	512	225	1,078 / 539

5 Evaluation

We evaluated our prototype on 20 benchmark programs spanning multiple domains, totaling 6.0 million lines of code (LoC), with the largest program exceeding 2M LoC. Table 1 lists these benchmarks. To evaluate the impact of our changes on bitfield compilation, we ensured that over half of the benchmarks use bitfields. For these, we report the number of bitfield variable declarations in the source code, as well as the number of read and write operations (static count, i.e., the number of such operations appearing in the source). Three benchmarks (LLVM, ngspice, and Z3) make particularly heavy use of bitfields.

For each program, we measured the impact of our changes on run-time performance, binary size, compilation time, peak compilation memory usage, and the total number of LLVM IR instructions. We also analyzed the miscompilation bugs fixed by our prototype on the LLVM test suite and on some benchmarks.

Experiments were conducted on a server with an AMD EPYC 9455 (Turin) 48-Core CPU, 128 GB of RAM, running Ubuntu 24.04.3 LTS. We used the Phoronix Test Suite (PTS)'s scripts to download, build, and run the benchmark programs.[7] No modifications were made to the source code or build systems of any program. To minimize environmental noise, we disabled unnecessary system services, ran all benchmarks in single-threaded mode, and used `taskset` and `nice` to pin programs to a single CPU core and assign them maximum scheduler priority. We also disabled address space

[7] https://github.com/phoronix-test-suite/phoronix-test-suite/

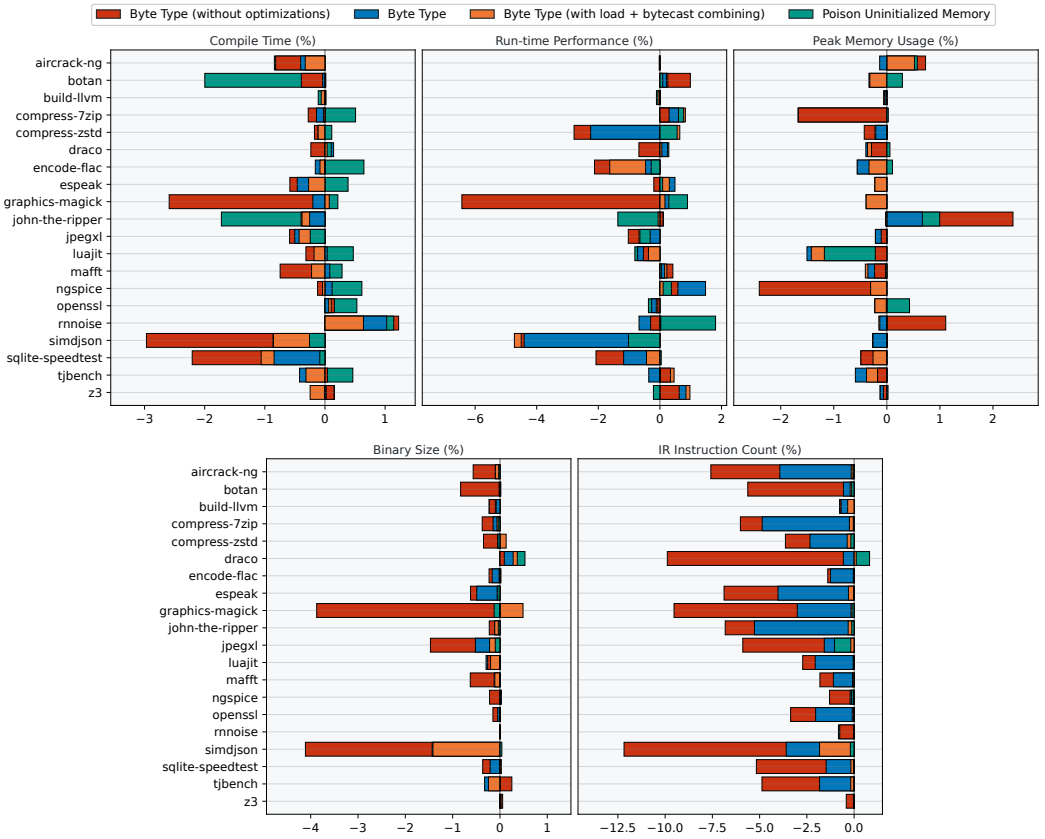


Fig. 5. Benchmark results: compile time, run-time performance, peak compilation memory usage, binary size, and total number of LLVM IR instructions. Results are reported as the improvement percentage relative to the baseline. Positive values indicate improvements (e.g., faster compilation time or smaller binaries). We show 4 bars: three for enabling the byte type with and without optimizations and with folding `bytecast` onto `load`, and another for switching uninitialized memory to poison and lowering bitfields using freezing loads.

layout randomization (ASLR), and relied on the statistical methods built into PTS to ensure that the relative standard deviation of results remained below 2%.

5.1 Run-Time Performance

Fig. 5 shows the results of evaluating the 20 benchmark programs using four prototype variants: (1) enabling the byte type in LLVM IR and using it in Clang to compile `char` variables; (2) the same, but with the new optimizations for `bytecast` instructions enabled; (3) variant (2) plus folding non-`exact` `bytecast` onto `load` instructions; and (4) changing the value of uninitialized memory to `poison` and modifying Clang's bitfield lowering to use freezing loads. The third variant is included to explore the potential impact if the semantics of `load` was changed to allow type punning.

As shown in the top-center plot, introducing the byte type causes performance regressions of up to 6.4% (average of 1.6%), with 5 benchmarks exceeding the 2% threshold. These slowdowns arise because adding a new IR instruction requires updating cost models and instruction matchers; without these changes, many optimizations become conservative and skip code with the new

instruction. With our new optimizations, the worst slowdown drops to 4.4%, the average to 0.8%, and only two benchmarks exceed the 2% threshold. Enabling type punning in `load` instructions reduces the average slowdown further to 0.2%.

The simdjson benchmark is the only case that retains a noticeable slowdown. This occurs because LLVM's jump-threading optimization can no longer merge certain basic blocks because its range analysis (LazyValueInfo) currently operates only on integer types. Extending this analysis to support the byte type is left for future work.

Switching uninitialized memory to poison has an average net-zero impact on run-time performance, with only one benchmark (John the Ripper) showing a larger slowdown of 1.4%. We traced this slowdown to a loop alignment issue in one source file: some loops containing vector instructions became misaligned due to slight code-size changes elsewhere. This behavior reflects a limitation of LLVM's loop-alignment heuristics rather than a direct consequence of our changes.

In addition to the optimizations specific to the byte type, we implemented two general-purpose LLVM optimizations for `freeze` instructions: one in InstCombine to simplify `select` instructions with `freeze poison` operands, and another extending the SLP vectorizer to support `freeze`. We have upstreamed both optimizations, which are already included in LLVM 21, our baseline, ensuring a fair comparison, as these improvements benefit both the baseline and our prototype.

Overall, we improve LLVM's correctness at a small cost. As with the introduction of `freeze` a few years ago [32], we expect the remaining regressions to be recovered over time by the community.

5.2 Binary Size

Fig. 5, bottom-left plot, shows that enabling the byte type has minimal impact on binary size. With optimizations enabled, the average increase is only 0.2% (0.7% without optimizations, and 0.1% with load type punning).

As examples of the impact of our optimizations, adjusting the loop vectorizer's cost model to account for the byte type made it vectorize fewer loops, which eliminated binary-size regressions caused by over-vectorization: 2.7% in pbzip2, 1.5% in GraphicsMagick, and 1.1% in tjbench. Enhancing the loop-idiom recognizer to detect `strlen`-like patterns involving the byte type fixed a 1% code-size regression in eSpeak. Finally, updating SimplifyCFG's cost model for the `bytecast` instruction enabled additional basic blocks to be merged in Aircrack-ng, which in turn unlocked further optimizations and reduced binary size by 1%.

To better understand the impact of our prototype on code size, we analyzed individual functions rather than only the overall binary size, which can mask localized differences. Fig. 6 presents two histograms per benchmark: one showing the distribution of function sizes, and another showing size differences between the baseline and each prototype variant. Across all benchmarks, the differences are centered around zero, as expected, since our changes should not affect the generated assembly. Minor variations are typically due to different register allocation decisions: because the allocator is greedy and the problem NP-hard, small perturbations in the IR can lead to large differences in the final code. However, a few benchmarks exhibit outliers with size differences of up to a few kilobytes. Most cases we investigated and fixed were due to cost model adjustments that had not yet been done; most remaining discrepancies are likely due to the same reason.

5.3 Compilation Metrics

We also measured several compilation metrics, including compilation time, peak memory usage, and the total number of LLVM IR instructions. These are important metrics given the substantial resources required to compile large programs.

Optimizations for the byte type improve all metrics, reducing the average compilation slowdown from 0.5% to 0.15%. This improvement is expected: fewer IR instructions (the average instruction

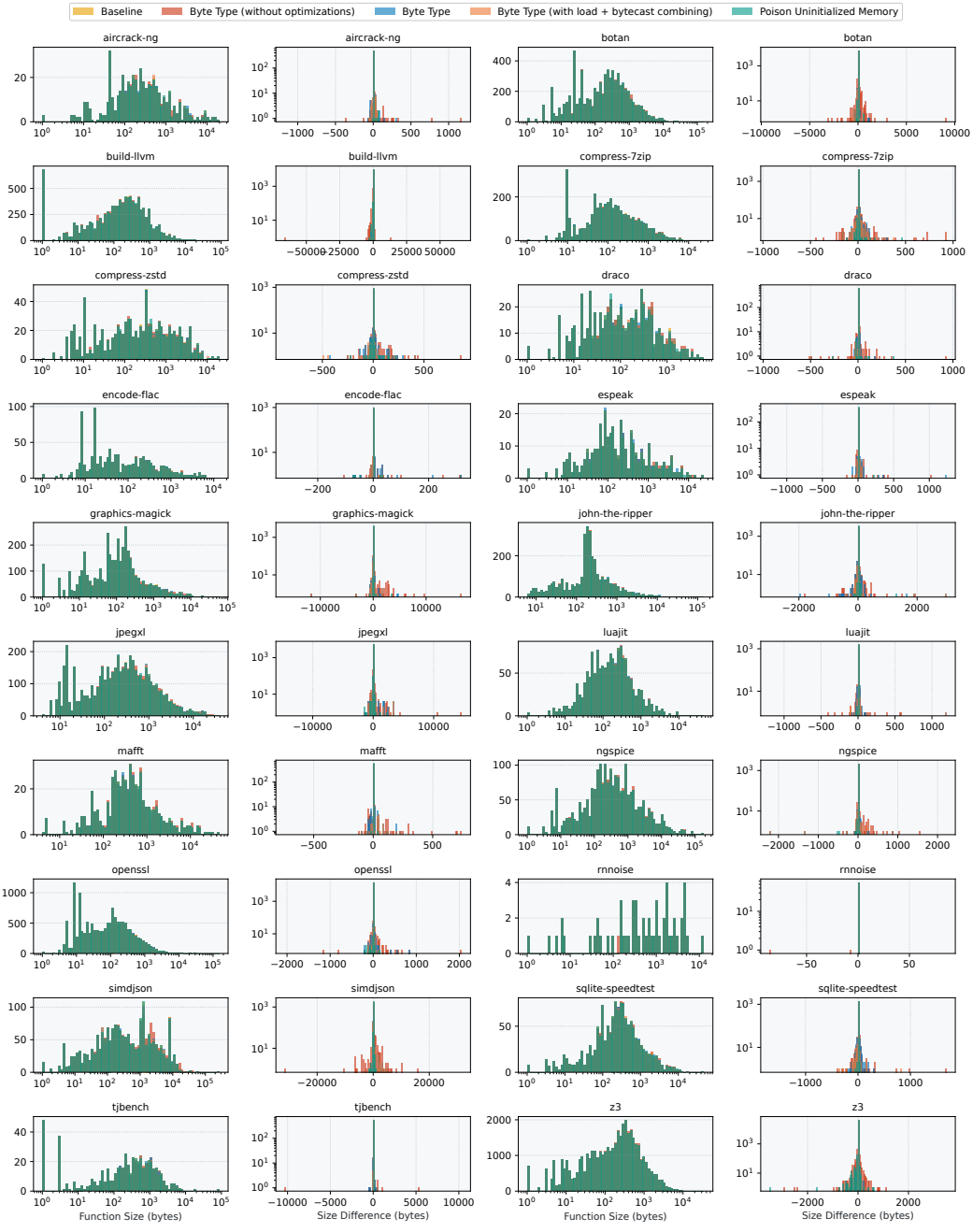


Fig. 6. Histograms (1st and 3rd columns) showing the distribution of function sizes for each benchmark. The 2nd and 4th columns show the histogram of function size differences between the baseline and each prototype variant; a bar at zero is ideal, indicating no change in function size. Bars use translucent colors, so overlapping values blend (e.g., green results from overlapping baseline and other variants).

Table 2. LLVM unit tests (under the `llvm/test/Transforms` directory) that were previously flagged as unsound by Alive2. The second column describes the incorrect transformation exposed by each test.

Test	Unsoundness Reason
ExpandMemCmp/AArch64/memcmp.ll	`memcmp` to integer load/store pairs
ExpandMemCmp/X86/bcmp.ll	`bcmp` to integer load/store pairs
ExpandMemCmp/X86/memcmp-x32.ll	`memcmp` to integer load/store pairs
ExpandMemCmp/X86/memcmp.ll	`memcmp` to integer load/store pairs
GVN/metadata.ll	Unsound pointer coercions
GVN/pr24397.ll	Unsound pointer coercions
InstCombine/bcmp-1.ll	`bcmp` to integer load/store pairs
InstCombine/memcmp-1.ll	`memcmp` to integer load/store pairs
InstCombine/memcpy-to-load.ll	`memcpy` to integer load/store pairs
PhaseOrdering/swap-promotion.ll	`memcpy` to integer load/store pairs
SROA/alignment.ll	`memcpy` to integer load/store pairs

count increase drops from 4.8% to 2.0%) reduce memory usage (overhead reduced from 0.4% to 0.2%), which in turn lowers cache misses and compilation time. Allowing type punning in `load` instructions decreases the IR size further, bringing the instruction count overhead down to 0.2%. Finally, replacing the value of uninitialized memory with `poison` has a negligible effect, yielding an average net-zero change in compilation time and peak memory usage.

The optimizations we implemented reduce the number of `bytecast` instructions substantially, from an average of 2.5% to 1.6% of all instructions. With type punning, this ratio is reduced to 0.1%.

5.4 Correctness Validation

To ensure the correctness of our prototypes, we ran LLVM's test suite with Alive2 [37] and validated the compilation of two benchmark programs using translation validation.

Table 2 summarizes the 11 LLVM unit tests (8% of the total) that were previously flagged as unsound by Alive2 in baseline LLVM and are now fixed. These tests were fixed by using the byte type for lowering memory copy functions and optimizations that widen or merge loads. No new test failures were reported.

We applied translation validation to four benchmarks: Draco, eSpeak, FLAC, and TJBench. Validation time did not change noticeably in any case. In eSpeak, one Alive2 bug report related to `memcpy` lowering was fixed, and no new issues were reported. In Draco, 330 Alive2 bug reports were fixed, also linked to `memcpy` lowering, but two new classes of bug reports appeared.

The first concerns the removal of `store undef` instructions, an optimization known to be unsound for a long time but kept by LLVM developers for performance reasons. Eliminating this optimization will likely require removing `undef` from LLVM. The second class of reports are false positives caused by Alive2's flow-insensitive tracking of escaped stack objects, which can make some function calls return non-escaped local pointers. Compiling `char` variables to bytes introduces additional opportunities for such cases, as calls that previously returned integers may now return pointers. This lack of precision in Alive2 predates our work and lies outside the scope of this paper.

A less obvious benefit of removing undef values from LLVM IR is faster translation validation. Proving refinement in the presence of undef requires a quantifier alternation [37], which is expensive to reason about. By paving the way for eliminating undef, this work not only simplifies the IR but also holds the potential to improve correctness by reducing validation time.

6 Related Work

Undefined behavior (UB) in source languages, primarily C and C++, has been the subject of extensive research, addressing its role in enabling compiler optimizations [15, 17, 18, 20, 43, 44, 59], its security implications [14, 23, 51–53], and the de facto semantics assumed by developers, which often diverge from the language standards [40]. Because UB is sometimes implicit in the C and C++ specifications, several efforts have sought to systematically enumerate all cases of UB in C [19] and C++ [54].

At the compiler IR level, UB has been studied primarily through formalizations of LLVM IR semantics. Early work revealed how UB enables aggressive optimizations [9, 22, 32, 36, 38, 58], while also exposing the non-trivial complications introduced by the `undef` value, which can render seemingly innocuous optimizations unsound. Other studies examined how production compilers like GCC exploit UB in practice [46]. Whereas prior efforts focused on characterizing and reasoning about existing IR designs and their tradeoffs, our work instead eliminates an entire class of UB to simplify LLVM's value lattice and reduce bugs stemming from misunderstandings of its semantics.

Several studies [44, 49] have shown that compilers are not monotonic with respect to UB: introducing additional UB can sometimes lead to worse generated code, even though, in theory, the presence of more UB should only expand the set of permissible optimizations.

Recent work on memory models for programming languages and IRs has focused mainly on pointer provenance and on enabling optimizations in low-level code that mixes casts between integers and pointers, as well as bitwise and arithmetic operations on integers derived from pointers [2, 4, 5, 21, 24, 26, 30, 33]. In particular, hardware architectures with fat pointers, such as CHERI's capability system, crucially rely on compilers to maintain accurate pointer provenance [13, 57]. It is worth noting that the semantics of source languages involve different considerations than those of a compiler IR. For example, many optimizations that are sound at the IR level would be unsound if applied directly in C, due to differences in aliasing rules, provenance tracking, and the treatment of UB [25, 27, 39, 40, 50]. This distinction is intentional: programming languages need not permit all optimizations themselves, as long as they can be efficiently compiled into an IR such as LLVM's that enables them safely.

CompCert [34] also features a deferred UB value, `undef`, which is semantically equivalent to LLVM's `poison`. It represents the value of uninitialized memory [35]. Because branching on `undef` triggers UB in CompCert, certain optimizations such as loop unswitching are unsound and therefore not performed. Mullen et al. [41] discuss how the presence of `undef` interferes with peephole optimizations in CompCert, while Besson et al. [6] proposed assigning uninitialized memory a fixed non-deterministic value, akin to initializing memory with `freeze poison` in LLVM. GCC, in contrast, defines uninitialized memory as having a non-deterministic and non-consistent value, similar to LLVM's `undef`.

While the current semantics of C++ allows compilers to translate reads of uninitialized memory into `poison`, a proposal for C++26 would require a different treatment [29]. The proposal requires stack variables to be initialized to a fixed non-deterministic value, similar to `freeze poison`. Given its potential performance impact, it remains uncertain whether this proposal will be adopted.

LLVM 15 changed how pointers are represented. Previously, pointer types encoded their pointee type (e.g., `i32*`); now all pointers use a single opaque `ptr` type. This change reflects the IR semantics that the pointee type belongs to instructions (e.g., `load`, `store`) rather than to pointer operands. While this reduced bugs caused by LLVM developers incorrectly relying on pointer types, it also led to more mixed-type load merging (which we fixed using the new byte type). Zhou et al. [60] propose recovering pointee types to improve alias analysis.

Instead of introducing a dedicated byte type in LLVM IR, some have proposed redefining the integer type to represent bytes [62]. However, this approach poses several challenges, as many

integer optimizations become unsound when applied to bytes, potentially causing performance regressions. Others have suggested disentangling provenance from pointers [16], effectively treating pointers as integers and passing provenance information explicitly as an additional argument to memory operations. While this design could address some provenance-related issues that the byte type also solves, it does not eliminate the need for a type capable of representing raw memory data, such as distinguishing poison at the bit level.

7 Discussion and Future Work

Since the writing of this paper, the first patch introducing the byte type to LLVM IR has been merged into the LLVM codebase. The LLVM community has also decided to allow type punning in `load` instructions, motivated in part by our finding that it facilitates folding with casts, as well as by Rust's requirement for such semantics. Moreover, the `bytecast` instruction was dropped entirely from the design. In its place, `bitcast` will be used, offering only a non-exact `bytecast` to match the semantics of `load`. The rationale for this decision is that retaining the exact variant of `bytecast` alongside a type-punning `load` would prevent the removal of spurious `load`+`store` sequences, since such sequences would no longer be no-ops: they can alter the type of the stored data, a difference that an exact `bytecast` would be able to observe. The chosen design therefore precludes any IR instruction from distinguishing the underlying type of a byte.

One topic still under discussion within the community is whether bitwise instructions should be extended to operate on bytes, or whether new instructions should instead be introduced to extract parts of bytes. A related proposal calls for a new instruction to modify part of a byte. The community appears to favor introducing these two new instructions for engineering reasons, namely to avoid crashes in code that expects bitwise instructions to have integer operands. From a theoretical standpoint, however, these instructions are merely syntactic sugar for bitwise operations, and our model therefore requires no modification.

Fully realizing the removal of `undef` in practice requires additional work to preserve existing optimizations that remain valid. For example, a `freeze poison` with a single use is equivalent to `undef`, allowing optimizations such as rewriting `undef + x` into `x` to remain valid by matching `freeze poison` instead. Other optimizations can be restricted to `poison` values, while some will no longer apply and must be removed. Fortunately, identifying the optimizations that require updates is straightforward, and this work can be performed incrementally.

8 Conclusion

Undef values have long been a major source of bugs in LLVM, as their unintuitive semantics can render otherwise valid optimizations unsound. In this paper, we introduce an LLVM IR extension that makes it possible to model uninitialized memory as poison instead of undef, thereby eliminating undef's last major use.

Our evaluation shows that the new design has negligible impact on performance and code size, while resolving long-standing correctness issues in LLVM. Moreover, the revised IR semantics is backward compatible, ensuring a smooth migration path. Overall, our work simplifies LLVM IR's semantics and provides a solid foundation for future compiler optimizations.

Acknowledgments

The authors thank Eli Friedman, Ralf Jung, Nikita Popov, and Alina Sbîrlea, for discussions and feedback on earlier drafts. This work was supported in part by an unrestricted gift from Google and national funds through Fundação para a Ciência e a Tecnologia (FCT), under projects UID/50021/2025 (DOI: https://doi.org/10.54499/UID/50021/2025) and UID/PRR/50021/2025 (DOI: https://doi.org/10.54499/UID/PRR/50021/2025).

References

[1] C. Scott Ananian. 1999. *The Static Single Information Form.* Master's thesis. MIT. https://dspace.mit.edu/handle/1721.1/149908

[2] Calvin Beck, Irene Yoon, Hanxi Chen, Yannick Zakowski, and Steve Zdancewic. 2024. A Two-Phase Infinite/Finite Low-Level Memory Model: Reconciling Integer–Pointer Casts, Finite Space, and undef at the LLVM IR Level of Abstraction. *Proc. ACM Program. Lang.* 8, ICFP, Article 263 (Aug. 2024). doi:10.1145/3674652

[3] Ryan Berger, Mitch Briles, Nader Boushehrinejad Moradi, Nicholas Coughlin, Kait Lam, Nuno P. Lopes, Stefan Mada, Tanmay Tirpankar, and John Regehr. 2025. Translation Validation for LLVM's AArch64 Backend. *Proc. of the ACM on Programming Languages* 9, OOPSLA2 (Oct. 2025). doi:10.1145/3763147

[4] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2015. A Concrete Memory Model for CompCert. In *ITP.* doi:10.1007/978-3-319-22102-1_5

[5] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2017. CompCertS: A Memory-Aware Verified C Compiler Using Pointer as Integer Semantics. In *ITP.* doi:10.1007/978-3-319-66107-0_6

[6] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2019. A Verified CompCert Front-End for a Memory Model Supporting Pointer Arithmetic and Uninitialised Data. *J Autom Reasoning* 62 (April 2019). doi:10.1007/s10817-017-9439-z

[7] Benoit Boissinot, Philip Brisk, Alain Darte, and Fabrice Rastello. 2012. SSI Properties Revisited. *ACM Trans. Embed. Comput. Syst.* 11S, 1, Article 21 (June 2012). doi:10.1145/2180887.2180898

[8] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. 2013. Simple and efficient construction of static single assignment form. In *CC.* doi:10.1007/978-3-642-37051-9_6

[9] Soham Chakraborty and Viktor Vafeiadis. 2017. Formalizing the concurrency semantics of an LLVM fragment. In *CGO.* doi:10.1109/CGO.2017.7863732

[10] Fred Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. 1996. Effective representation of aliases and indirect memory operations in SSA form. In *CC.* doi:10.1007/3-540-61053-7_66

[11] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991). doi:10.1145/115372.115320

[12] Manjeet Dahiya and Sorav Bansal. 2017. Modeling Undefined Behaviour Semantics for Checking Equivalence Across Compiler Optimizations. In *HVC.* doi:10.1007/978-3-319-70389-3_2

[13] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. 2019. CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment. In *ASPLOS.* doi:10.1145/3297858.3304042

[14] Will Dietz, Peng Li, John Regehr, and Vikram Adve. 2015. Understanding Integer Overflow in C/C++. *ACM Trans. Softw. Eng. Methodol.* 25, 1, Article 2 (Dec. 2015). doi:10.1145/2743019

[15] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. 1998. Type-based alias analysis. In *PLDI.* doi:10.1145/277650.277670

[16] Jeroen Dobbelaere. 2019. Full 'restrict' support in LLVM. https://discourse.llvm.org/t/full-restrict-support-in-llvm/53267

[17] Johannes Doerfert, Brian Homerding, and Hal Finkel. 2019. Performance exploration through optimistic static program annotations. In *ISC High Performance 2019.* doi:10.1007/978-3-030-20656-7_13

[18] Dan Frumin, Léon Gondelman, and Robbert Krebbers. 2019. Semi-automated Reasoning About Non-determinism in C Expressions. In *ESOP.* doi:10.1007/978-3-030-17184-1_3

[19] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 2015. Defining the undefinedness of C. In *PLDI.* doi:10.1145/2737924.2737979

[20] Eduard Kamburjan and Nathan Wasser. 2022. The Right Kind of Non-Determinism: Using Concurrency to Verify C Programs with Underspecified Semantics. In *ICE.* doi:10.4204/EPTCS.365.1

[21] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. 2015. A formal C memory model supporting integer-pointer casts. In *PLDI.* doi:10.1145/2737924.2738005

[22] Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. 2018. Crellvm: verified credible compilation for LLVM. In *PLDI.* doi:10.1145/3192366.3192377

[23] Andreas D. Kellas, Alan Cao, Peter Goodman, and Junfeng Yang. 2023. Divergent Representations: When Compiler Optimizations Enable Exploitation. In *SPW.* doi:10.1109/SPW59333.2023.00035

[24] Yonghyun Kim, Minki Cho, Jaehyung Lee, Jinwoo Kim, Taeyoung Yoon, Youngju Song, and Chung-Kil Hur. 2025. Archmage and CompCertCast: End-to-End Verification Supporting Integer-Pointer Casting. *Proc. ACM Program. Lang.*

9, POPL, Article 45 (Jan. 2025). doi:10.1145/3704881
[25] Robbert Krebbers. 2013. Aliasing Restrictions of C11 Formalized in Coq. In *CPP*. doi:10.1007/978-3-319-03545-1_4
[26] Robbert Krebbers. 2016. A Formal C Memory Model for Separation Logic. *J Autom Reasoning* 57 (Dec. 2016). doi:10.1007/s10817-016-9369-1
[27] Robbert Krebbers and Freek Wiedijk. 2015. A Typed C11 Semantics for Interactive Theorem Proving. In *CPP*. doi:10.1145/2676724.2693571
[28] Jaeseong Kwon, Bongjun Jang, Juneyoung Lee, and Kihong Heo. 2025. Optimization-Directed Compiler Fuzzing for Continuous Translation Validation. *Proc. ACM Program. Lang.* 9, PLDI, Article 172 (June 2025). doi:10.1145/3729275
[29] Thomas Köppe. 2024. P2795R5: Erroneous behaviour for uninitialized reads. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2795r5.html
[30] Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. 2018. Reconciling high-level optimizations and low-level code in LLVM. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018). doi:10.1145/3276495
[31] Juneyoung Lee, Dongjoo Kim, Chung-Kil Hur, and Nuno P. Lopes. 2021. An SMT Encoding of LLVM's Memory Model for Bounded Translation Validation. In *CAV*. doi:10.1007/978-3-030-81688-9_35
[32] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming undefined behavior in LLVM. In *PLDI*. doi:10.1145/3062341.3062343
[33] 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, Article 20 (Jan. 2022). doi:10.1145/3498681
[34] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. doi:10.1145/1538788.1538814
[35] Xavier Leroy and Sandrine Blazy. 2008. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. *J Autom Reasoning* 41 (July 2008). doi:10.1007/s10817-008-9099-0
[36] Liyi Li and Elsa L. Gunter. 2020. K-LLVM: A Relatively Complete Semantics of LLVM IR. In *ECOOP*. doi:10.4230/LIPIcs.ECOOP.2020.7
[37] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *PLDI*. doi:10.1145/3453483.3454030
[38] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Pptimizations with Alive. In *PLDI*. doi:10.1145/2737924.2737965
[39] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. 2019. Exploring C semantics and pointer provenance. *Proc. ACM Program. Lang.* 3, POPL, Article 67 (Jan. 2019). doi:10.1145/3290380
[40] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the depths of C: elaborating the de facto standards. In *PLDI*. doi:10.1145/2908080.2908081
[41] Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. 2016. Verified peephole optimizations for CompCert. In *PLDI*. doi:10.1145/2908080.2908109
[42] Diego Novillo. 2007. Memory SSA - A Unified Approach for Sparsely Representing Memory Operations. In *GCC Developers' Summit*. https://www.airs.com/dnovillo/Papers/mem-ssa.pdf
[43] Ankush Phulia, Vaibhav Bhagee, and Sorav Bansal. 2020. OOElala: order-of-evaluation based alias analysis for compiler optimization. In *PLDI*. doi:10.1145/3385412.3385962
[44] Lucian Popescu and Nuno P. Lopes. 2025. Exploiting Undefined Behavior in C/C++ Programs for Optimization: A Study on the Performance Impact. *Proc. ACM Program. Lang.* 9, PLDI, Article 161 (June 2025). doi:10.1145/3729260
[45] Fabrice Rastello and Florent Bouchez Tichadou. 2022. *SSA-based Compiler Design*. Springer. doi:10.1007/978-3-030-80515-9
[46] Zefan Shen. 2022. The Impact of Undefined Behavior on Compiler Optimization. In *ESSE*. doi:10.1145/3501774.3501781
[47] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *ISSTA*. doi:10.1145/2931037.2931074
[48] Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing static analyses for precision and soundness. In *CGO*. doi:10.1145/3368826.3377927
[49] Theodoros Theodoridis and Zhendong Su. 2024. Refined Input, Degraded Output: The Counterintuitive World of Compiler Behavior. *Proc. ACM Program. Lang.* 8, PLDI, Article 174 (June 2024). doi:10.1145/3656404
[50] Andrew Tolmach, Chris Chhak, and Sean Anderson. 2024. Defining and Preserving More C Behaviors: Verified Compilation Using a Concrete Memory Model. In *ITP*. doi:10.4230/LIPIcs.ITP.2024.36
[51] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Undefined behavior: what happened to my code?. In *APSYS*. doi:10.1145/2349896.2349905
[52] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards optimization-safe systems: analyzing the impact of undefined behavior. In *SOSP*. doi:10.1145/2517349.2522728

[53] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. 2023. Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs. In *USENIX Security*. https://www.usenix.org/system/files/usenixsecurity23-xu-jianhao.pdf

[54] Shafik Yaghmour. 2019. C++ Proposal P1705R1: Enumerating Core Undefined Behavior. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1705r1.html

[55] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *PLDI*. doi:10.1145/1993498.1993532

[56] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. 2021. Modular, compositional, and executable formal semantics for LLVM IR. *Proc. ACM Program. Lang.* 5, ICFP, Article 67 (Aug. 2021). doi:10.1145/3473572

[57] Vadim Zaliva, Kayvan Memarian, Brian Campbell, Ricardo Almeida, Nathaniel Filardo, Ian Stark, and Peter Sewell. 2025. A CHERI C Memory Model for Verified Temporal Safety. In *CPP*. doi:10.1145/3703595.3705878

[58] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL*. doi:10.1145/2103656.2103709

[59] Peter Zhong, Shu-Hung You, Simone Campanoni, Robert Bruce Findler, Matthew Flatt, and Christos Dimoulas. 2024. A Calculus for Unreachable Code. arXiv:2407.04917

[60] Jinmeng Zhou, Ziyue Pan, Wenbo Shen, Xingkai Wang, Kangjie Lu, and Zhiyun Qian. 2025. Type-Alias Analysis: Enabling LLVM IR with Accurate Types. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA097 (June 2025). doi:10.1145/3728974

[61] Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. 2021. An empirical study of optimization bugs in GCC and LLVM. *Journal of Systems and Software* 174 (2021). doi:10.1016/j.jss.2020.110884

[62] Zoxc. 2022. A memory model for LLVM IR supporting limited type punning. https://discourse.llvm.org/t/a-memory-model-for-llvm-ir-supporting-limited-type-punning/61948

Received 2025-11-13; accepted 2026-04-03