Detecting Strict Aliasing Violations in the Wild

Pascal Cuoq[1], Loïc Runarvot[1], and Alexander Cherepanov[23]

[1] TrustInSoft
[2] Openwall
[3] National Research University Higher School of Economics

Abstract. Type-based alias analyses allow C compilers to infer that memory locations of distinct types do not alias. Idiomatic reliance on pointers on the one hand, and separate compilation on the other hand, together make it impossible to get this aliasing information any other way. As a consequence, most modern optimizing C compilers implement some sort of type-based alias analysis. Unfortunately, pointer conversions, another pervasive idiom to achieve code reuse in C, can interact badly with type-based alias analyses. This article investigate the fine line between the allowable uses of low-level constructs (pointer conversions, unions) that should never cause the predictions of a standard-compliant type-based alias analysis to be wrong, and the dangerous uses that can result in bugs in the generated binary. A sound and precise analyzer for strict aliasing violations is briefly described.

Keywords: strict aliasing, type-based alias analysis, C, static analysis

1 Introduction

Until approximately 1999[11,10], the static analysis literature tended towards ignoring low-level aspects of C programs completely. Sound analyzers (either actual prototypes or hypothetical implementations of the entirety of the analyzer described in an article) would not deal with low-level programming idioms that are, for better or for worse, present in C code as it exists and as it gets written. An example, seen in safety-critical embedded code, is to take the address of the first member of a struct that contains only floats, and proceed to initialize the struct via a loop, through pointer arithmetic, as if it were an array. Rejecting this construct outright means giving up on making the analyzer useful for this kind of code. Alternately, the analyzer might maintain soundness by becoming very imprecise in presence of such low-level constructs. This also makes the analyzer unusable in practice. As sound static analysis gained industrial adoption as a useful tool for the certification of safety-critical embedded software, the design choice of accepting the low-level construct and handling it precisely became more common[9].

Attempts to handle low-level constructs with precision in sound static analyzers sometimes works at cross-purposes with increasingly sophisticated optimizations, based on *undefined behavior*[4], in C compilers. In presence of constructs

---

[4] See `http://blog.regehr.org/archives/213`

that invoke undefined behavior for some or all inputs, compilers are allowed to generate binary code:

– without any diagnostic at compile-time,
– that does not raise any exception at run-time,
– and only works as intended for inputs that do not invoke undefined behavior— this can be the empty set in the case of intentional reliance on undefined behavior by the developer.

Optimizations based on undefined behavior are useful[5]. But these optimizations can have the unfortunate effect of making static analyzers intended to be sound unsound in practice. To be fair, the problem, as long as one is aware of it, can easily be circumvented by disabling the optimization, aligning the semantics of the compiler and the analyzer. GCC understands `-fwrapv` for wrapping signed arithmetic overflows, and `-fno-strict-aliasing` for no type-based alias analysis. Awareness is the only difficulty in this plan. For instance, legacy C libraries that have worked well for 25 years and are now deployed everywhere may violate the rules in a way that new versions of C compilers written in 2017 suddenly become sophisticated enough to take advantage of.

This article is concerned with the optimization named `-fstrict-aliasing` in GCC and Clang, and with guaranteeing that programs do not invoke the kind of undefined behavior that allows this optimization to change the behavior of the program from what was intended by the developer. With funding from the Linux Foundation's Core Infrastructure Initiative, we are building a static analyzer to detect violations of strict aliasing, so that legacy C libraries at the heart of the Internet can be diagnosed with strict aliasing violations, and fixed before the problem becomes urgent. This is work in progress.

2   Strict Aliasing in the C Standards

When the C programming language was standardized in 1980s the Committee considered the question whether an object may be accessed by an lvalue of a type different from the declared type of the object. This would hamper optimization and, thus, "[t]he Committee has decided that such dubious possibilities need not be allowed for"[6]. However, certain prevalent exceptions were recognized: types differently qualified and with different signedness may alias and any type may be accessed as a character type. Interaction between aggregates (and unions) and their members was also accounted for. The resulting rules were included in C89 and got known as "strict aliasing rules".

In 1997, it was pointed out[7] that the text in the C89 standard does not cover the case of allocated objects which do not have a declared type. The standard was corrected and the strict aliasing rules in C99 have the following form:

---

[5] See https://gist.github.com/rygorous/e0f055bfb74e3d5f0af20690759de5a7
[6] C89 Rationale, http://std.dkuug.dk/jtc1/sc22/wg14/docs/rationale/c89/rationale.ps.gz
[7] http://open-std.org/jtc1/sc22/wg14/www/docs/n640.ps

[C99, 6.5:6] The effective type of an object for an access to its stored value is the declared type of the object, if any.[75)] If a value is stored into an object having no declared type through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value. If a value is copied into an object having no declared type using memcpy or memmove, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. For all other accesses to an object having no declared type, the effective type of the object is simply the type of the lvalue used for the access.

[C99, 6.5:7] An object shall have its stored value accessed only by an lvalue expression that has one of the following types:[76)]
  – a type compatible with the effective type of the object,
  – a qualified version of a type compatible with the effective type of the object,
  – a type that is the signed or unsigned type corresponding to the effective type of the object,
  – a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,
  – an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or
  – a character type.

75) Allocated objects have no declared type.
76) The intent of this list is to specify those circumstances in which an object may or may not be aliased.

There were no changes in the text in C11 except for renumbering footnotes.

The rules are symmetric with regard to signedness of types but not to qualified/unqualified versions.

The rules are quite clear for objects declared with one of the basic types. Everything more complex poses some kind of problems.

The natural aliasing between aggregates (and unions) and their members is permitted by the fifth item in C99, 6.5:7, but the formulation is quite sloppy. The problem was pointed out[8] at least in 1997, a later discussion can be found in defect reports 1409[9] and 1520[10]. A shared understanding of the intended meaning seems to exist, although nobody has found yet a fixed wording.

Unions have members of different types which naturally alias each other. Possibility of uncontrolled access to these members would undermine the idea of

---

[8] http://open-std.org/jtc1/sc22/wg14/3406
[9] http://open-std.org/jtc1/sc22/wg14/www/docs/n1409.htm
[10] http://open-std.org/jtc1/sc22/wg14/www/docs/n1520.htm

strict aliasing. Thus, we have to conclude that strict aliasing rules govern the use of members of unions. But there is an exception—it's always permitted to read any member of a union by the . operator (and the `->` operator). The relevant part of the C99 standard is:

> [C99, 6.5.2.3:3] A postfix expression followed by the . operator and an identifier designates a member of a structure or union object. The value is that of the named member,[82)] and is an lvalue if the first expression is an lvalue.

> 82) If the member used to access["read" in C11] the contents of a union object is not the same as the member last used to store a value in the object, the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type as described in 6.2.6 (a process sometimes called "type punning"). This might be a trap representation.

3   Examples

This section lists examples of simple functions where a memory access can be optimized, or not, depending on the interpretation of the strict aliasing rules. On the right-hand side of each example, the assembly code generated by an optimizing compiler is shown[11]. While reading the examples, bear in mind that in the x86-64 calling convention, `%rax` or its 32-bit subregister `%eax` is used for the return value when it is an integer or a pointer. `%rdi` or `%edi` holds the function's first integer/pointer argument, and `%rsi` or `%esi` holds the second one. The result, when a float, is instead placed in the floating-point register `%xmm0`, and `%xmm0` also holds the function's first float argument if any.

```
int ex1(int *p, float *q) {          ex1:
  *p = 1;                              movl $1, (%rdi)
  *q = 2.0f;                           movl $1, %eax
  return *p;                           movl $0x40000000, (%rsi)
}                                      ret
```

```
unsigned ui(unsigned *p, int *q) {   ui:
  *p = 1;                              movl $1, (%rdi)
  *q = 2;                              movl $2, (%rsi)
  return *p;                           movl (%rdi), %eax
}                                      ret
```

---

[11] https://godbolt.org/g/ggZzQo

```
long lll(long *p, long long *q) {          lll:
  *p = 1;                                    movq $1, (%rdi)
  *q = 2;                                    movl $1, %eax
  return *p;                                 movq $2, (%rsi)
}                                            ret


int x;                                     pp:
unsigned y;                                  movq $x, (%rdi)
int *pp(int **p, unsigned **q) {             movl $x, %eax
  *p = &x;                                   movq $y, (%rsi)
  *q = &y;                                   ret
  return *p;
}
typedef int (*f1)(int);
typedef int (*f2)(float);
int foo(int);                              pf:
int bar(float);                              movq $foo, (%rdi)
f1 pf(f1 *p, f2 *q) {                        movl $foo, %eax
  *p = foo;                                  movq $bar, (%rsi)
  *q = bar;                                  ret
  return *p;
}
struct s { int a; };
struct t { int b; };                       st1:
int st1(struct s *p, struct t *q) {  movl $1, (%rdi)
  p->a = 1;                                movl $1, %eax
  q->b = 2;                                movl $2, (%rsi)
  return p->a;                             ret
}
struct s { int a; };
struct t { int b; };                       st2:
int st2(struct s *p, struct t *q) {          movl $1, (%rdi)
  int *pa = & (p->a);                        movl $2, (%rsi)
  int *qb = & (q->b);                        movl (%rdi), %eax
  *pa = 1;                                   ret
  *qb = 2;
  return *pa;
}
```
The assembly code shown was produced by GCC 6.2.0

For each of the example functions in this section, the question is whether it behaves the way a programmer with a naïve view of memory use in C would expect, when somehow invoked with aliasing pointers as arguments, regardless of how the aliasing has been created at the call-site. Reading the assembly generated for one example by a particular compiler is faster and less distracting that building a caller that creates the aliasing condition.

For the sake of completeness, here is what a problematic caller would look like for the first example `ex1`:

```
int main(int c, char *v[]) {
  static_assert(sizeof(int) == sizeof(float),
    "Only for 32-bit int and IEEE 754 binary32 float");
  void *p = malloc(sizeof(float));
  ex1((int *)p, (float *)p);
}
```

The `main` function here is creating the conditions for `ex1` to misbehave, and, in a "code smell" sense, it can be said to be where the programmer's error lay. Experienced programmers familiar with strict aliasing rules in particular would worry about the conversions of the same `p` to two distinct pointer types. Regardless, it is the code inside function `ex1` that, when invoked in this context, violates the rules. Any reasonably precise analyzer can only hope to diagnose the problem there. The two pointer conversions in the above `main` are not invalid, and would constitute a valid program if put together with a different implementation for the function `ex1`. We do not show additional examples of calling contexts precisely in order to avoid wrongly thinking of the calling context as the place where the strict aliasing issue is located. Warning about pointer conversions is, essentially, what GCC's `-Wstrict-aliasing` option does, and this is not satisfactory because, to be blunt, pointer conversion is the sole code reuse mechanism available in the C language, and as such it is used as much as necessary, both in patterns that ends up violating strict aliasing rules and in patterns that do not. This is especially true of legacy code written in the 1990s, a time at which C was used to program high-level concepts for which a high-level language would hopefully be the obvious choice for new implementations today.

The example `ex1` shows the simplest, least controversial form of strict aliasing optimization. The only C developers who disagree with it reject the concept of type-based alias inference as a whole.

The example `ui` is not expected to be optimized, as C11 makes allowances for accessing an `unsigned` effective type with an `int` lvalue and vice-versa. In contrast, even when the standard integer types `int` and `long` (or respectively `long` and `long long`) happen to have the same width, compilers can assume that an lvalue of one type is not used to access an object with the other, as the standard allows them to—the types `int` and `long` are not compatible even when they have the same width.

In the example `pp`, GCC 6.2.0 (but none of the Clang versions available at the time of this writing) correctly uses the fact that the types `int*` and `unsigned*` are not compatible with each other to optimize the returned value into `&x`. Similarly, in example `pf`, GCC version 6.2.0 takes advantage of the incompatibility of the types "pointer to function taking an int and returning an int" and "pointer to function taking a float and returning an int" to optimize the returned value to the address of `foo`.

An example similar to `st1` was a crucial part of an internal committee discussion about the notion of type compatibility as early as 1995[12]. This example has popped again occasionally, for instance in GCC's mailing list in 2010[13] and later in Section 4.1.2

---

[12] Example y.c in `http://std.dkuug.dk/jtc1/sc22/wg14/docs/c9x/misc/tag-comp at.txt.gz`

[13] `https://gcc.gnu.org/ml/gcc/2010-01/msg00013.html`

of "C memory object and value semantics: the space of de facto and ISO standards"[14]. GCC versions 4.4.7 and 4.5.3 optimize `st2` identically to `st1`, but later GCC versions do not. It is not clear whether this change is incidental or results from a decision to limit the scope of strict aliasing optimizations: the reasoning that justifies the optimization of `st1` in GCC justifies the optimization of `st2`, too. A consequence for any sound static detector of strict aliasing violations is that the information of "pointed struct member" must be propagated associated to `pa` and `qb` in order to detect that the harmless-looking assignments `*pa = 1`, `*qb = 2` and retrieval `return *pa;` violate GCC's memory model because of previous statements.

```c
int ar1(int (*p)[8], int (*q)[8]) {
  (*p)[3] = 1;
  (*q)[4] = 2;
  return (*p)[3];
}
```

```
ar1:
 movl $1, 12(%rdi)
 movl $1, %eax
 movl $2, 16(%rsi)
 ret
```

```c
int ar2(int c, int (*p)[8],
        int (*q)[8]) {
  int z = 0;
  if (2 < c && c < 4) {
    (*p)[c+z] = 1;
    (*q)[4] = 2;
    return (*p)[c];
  }
  else
    return 0;
}
```

```
ar2:
 xorl %eax, %eax
 cmpl $3, %edi
 je .L
 rep ret
.L5:
 movl $1, 12(%rsi)
 movl $1, %eax
 movl $2, 16(%rdx)
 ret11
```

```c
int ar3(int (*p)[8], int (*q)[7]) {
  (*p)[3] = 1;
  (*q)[3] = 2;
  return (*p)[3];
}
```

```
ar3:
 movl $1, 12(%rdi)
 movl $2, 12(%rsi)
 movl 12(%rdi), %eax
 ret
```

```c
enum e1 { A = 0 };
enum e2 { B = 1 };
int ex1_enum(enum e1 *p, enum e2 *q)
{
  *p = A;
  *q = B;
  return *p;
}
```

```
ex1_enum:
 movl $0, (%rdi)
 xorl %eax, %eax
 movl $1, (%rsi)
 ret
```

[14] https://www.cl.cam.ac.uk/~pes20/cerberus/notes30.pdf, Draft, Revision 1571, 2016-03-17

```
enum e1 { A };
unsigned ex2_enum(unsigned *p, enum e1 *q)    ex2_enum:
{                                                movl $1, (%rdi)
  *p = 1;                                        movl $1, %eax
  *q = A;                                        movl $0, (%rsi)
  return *p;                                     ret
}
```

The assembly code shown was produced by GCC 6.2.0

The same optimization that GCC exhibits when compiling the function st1, GCC 6.2.0 also applies to array types in example ar1, where index 3 of an array of 8 ints is assumed not to alias with index 4 of an array of 8 ints. Clang versions available as of this writing do not optimize ar1.

The example ar2 shows that there are no a priori bounds to the ingenuity of compiler in order to infer that the indexes being accessed are necessarily different. As a consequence, a sound and precise static analyzer cannot limit itself to constant array indexes.

The example ar3, where the array types pointed by the arguments differ in size, seems easier to optimize than ar1, but is surprisingly optimized by neither Clang nor GCC as of this writing. Optimizing ar1 already constrains the developer never to view a large array as a superposition of overlapping smaller arrays, so GCC could optimize ar3. Showing this example to GCC developers is taking them in the direction of not optimizing ar1 instead[15].

In C, enum types have an underlying integer type, chosen by the compiler according to the set of values to hold by the enum. GCC chooses unsigned int for an enum destined to contain only the values 0 and 1 or 0. Despite these two enum types being compatible with unsigned int, GCC 6.2.0 optimizes programs as if a memory location of effective type one such enum could not be modified by an access to unsigned int, or an access to another such enum. We think this is a bug[16], but meanwhile, it is possible to detect that a program might be miscompiled by GCC 6.2.0 by treating in the analyzer an enum type based on unsigned int as if it were incompatible with unsigned int and other enum types based on unsigned int.

```
union u { int a; float b; };

int fr(float f) {                fr:
  union u t = { .b = f };          movd %xmm0, %eax
  return t.a;                      retq
}
/* q is really a pointer to a union u */
                                 u1:
int u1(int *p, void *q) {
  *p = 1;                          movl $1, (%rdi)
  *&((union u *)q)->b = 2;         movl $1073741824, (%rsi)
  return *p;                       movl $1, %eax
}                                  retq
```

---

[15] https://gcc.gnu.org/ml/gcc/2016-11/msg00111.html
[16] See https://gcc.gnu.org/bugzilla/show_bug.cgi?id=71598

```
int u2(int *p, union u *q) {
  *p = 1;
  q->b = 0.1;
  return *p;
}

void *mem(void);

int u3() {
  union u *p1 = mem();
  union u *p2 = mem();
  float *fp = &p2->b;
  p1->a = 1;
  *fp = 3.0;
  return p1->a;
}

int u4(void) {
  union u *p1 = mem();
  union u *p2 = mem();
  int* ip = &p1->a;
  *ip = 1;
  p2->b = 3.0;
  return *ip;
}

int u5(void) {
  union u *p1 = mem();
  union u *p2 = mem();
  p1->a = 1;
  p2->b = 3.0;
  return p1->a;
}
```

```
u2:
 movl $1, (%rdi)
 movl $1036831949, (%rsi)
 movl $1, %eax
 retq
u3:
 pushq %rbx
 callq mem
 movq %rax, %rbx
 callq mem
 movl $1, (%rbx)
 movl $1077936128, (%rax)
 movl $1, %eax
 popq %rbx
 retq
u4:
 pushq %rbx
 callq mem
 movq %rax, %rbx
 callq mem
 movl $1, (%rbx)
 movl $1077936128, (%rax)
 movl $1, %eax
 popq %rbx
 retq
u5:
 pushq %rbx
 callq mem
 movq %rax, %rbx
 callq mem
 movl $1, (%rbx)
 movl $1077936128, (%rax)
 movl $1, %eax
 popq %rbx
 retq
```

The assembly code shown was produced by Clang 3.9.0

The interactions of unions with type-punning and type-based alias analyses have caused enormous amounts of discussion, starting with a C99 standard that initially implied that reading from a union member other than the one used to setting the value of the union produced unspecified results (6.5.2.3:3) and a defect report about a regression of the type-punning powers of **union** with respect to C89[17]. Type-punning through unions remains ambiguously described in the C11 standard, and compilers that want to take advantage of type-based alias analyses for optimizations need to

---

[17] DR283, http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_283.htm

define their own rules[18], and convey them to the developer, which they do not always do clearly.

One extreme example of `union` use for type-punning is the function `fr` to convert a float to its representation as an `int`. This function is compiled to the intended binary code by all the compilers we tried. At the other end of the spectrum, the very function `ex1` that we used as first example can be an example of type-punning through unions when it is invoked in the following context:

```
int main(int c, char *v[]) {
  union { float f; int i; } u;
  ex1(&u.i, &u.f);
}
```

Obviously, compilers do not want to compile the function `ex1` cautiously just because any other compilation unit might invoke it with the addresses of distinct members of a same `union`. Between these two extremes exists a limit of what a compiler defines as reasonable use of a `union` for type-punning. Only programs within the limit are guaranteed to be translated to code that behaves as intended. All three of GCC, ICC and Clang fit in this general framework, but with different limits between reasonable and unreasonable uses of union for type-punning. GCC documents its expectations comparatively well[19], and sticks to what it documents: from the documentation, we expect functions `u1` through `u5` not to be optimized by GCC 6.2.0, and indeed, they are not. Clang does not document what usages of unions it deems acceptable that we could find. All the examples `u1` through `u5` are optimized, implying that perhaps the only acceptable use of a union for type-punning recognized by Clang is that of a variable accessed directly, without any pointer dereference being involved. ICC appears to adopt a middle-ground, by optimizing functions `u3` and `u4`, but not `u5`.

4   Detecting strict aliasing violations

In this section we sketch out the functioning principles of a static analyzer for detecting strict aliasing violations. The analyzer is a forward abstract interpreter [2] that assumes that the values of expressions are computed at the same time as the effective types, or have been computed and saved in a usable form[4]. The analyzer propagates "memory states", starting with the entry point of the program, until a fixpoint has been reached. In this case, a "memory state" assigns possible effective types to each bit of memory. The bit-level memory model is necessary in order to handle low-level constructs such as unions and pointer conversions, when they are used in accordance to strict aliasing rules.

The lattice used for each memory bit is the lattice of sets of effective types, ordered by inclusion (the power set lattice). The empty set of effective types is the least element.

---

[18] See for instance `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65892#c9` or the words "GCC doesn't implement C99 aliasing as written in the standard regarding unions. We don't do so because we determined that this can't possibly have been the intent of the standard as it makes type-based aliasing relatively useless" in `https://gcc.gnu.org/ml/gcc/2010-01/msg00263.html`

[19] Documentation at `https://gcc.gnu.org/onlinedocs/gcc-6.2.0/gcc/Optimize-Options.html`

It would technically not be necessary to keep information about all the possible effective types an object can have during the analysis. As soon as two sufficiently distant effective types are possible effective types for an object, there exists no declared type, compatible with both, with which this object can be accessed without a warning. In other words, it would not lead to a loss of precision to simplify the abstract domain used by identifying with the greatest element all sets containing at least two incompatible effective types. Our implementation avoids making all these sets of distant types the same in order to improve the usefulness of warning messages. In particular, the attitude of the analyzer's user towards the message "there may be a violation here because this `int` lvalue is used to access some unknown effective type" may be "I see why the analyzer is imprecise here, this is a false positive". The same user, provided with the better warning message "there may be a violation here because this `int` lvalue is used to access effective types `long` and `float`" may be "I see that the analyzer is imprecise here when it predicts that a `float` can be accessed, this is a false positive; but accessing a `long` can happen and is a genuine strict aliasing bug".

4.1   Effective Types

We describe the grammar of effective types using an OCaml-like notation.

```
type ival = ...

type integer_type =
  | Bool
  | Char | SignedChar | UnsignedChar
  | Short | UnsignedShort
  | Int | UnsignedInt
  | Long | UnsignedLong
  | LongLong | UnsignedLongLong

type float_type = Float | Double | LongDouble

type function_type =
  { return_type : simple_type;
    formals : simple_type list }

type simple_type =
  | Structure of structure
  | StructureField of field * simple_type
  | Array of simple_type * expr (* size *)
  | ArrayElement of simple_type
    * expr (* declared size for array *)
    * ival (* set of actual values for the index *)
  | Union of union_t
  | Enum of enum_t
  | IntegerType of integer_type
  | FloatType of float_type
  | FunctionType of function_type
  | VariadicFunctionType of function_type
  | PointerType of simple_type
```

```
| FirstAccessType
| VoidType
| MayAlias
```

Listing 1.1. Effective Types

The effective types used by the analyzer unsurprisingly resemble the static types familiar to the C programmer. Below are the most notable departures from the grammar of static types.

An effective type can be "member m of . . . ", (resp. "array element at index . . . of array . . . "). This is not the same effective type as the type of the struct member m (resp. the type of elements of the array). In order to handle example functions `st1`, `st2`, `ar1`, . . . , all the available information about the location of the subobject inside its containing objects must be memorized in the effective type.

The `FirstAccessType` constructor indicates that the effective type will be that of the lvalue used for reading until some effective type is written, following C11 6.5:6. The effective type `FirstAccessType` is used for the contents of memory blocks allocated through `calloc`, as well as for contents written by `read`, `fread`, `memset`, . . . This constructor is not necessary for the contents of a block allocated through `malloc`, because in this case the contents of the allocated block are uninitialized ("indeterminate" in C standard parlance). Reads of uninitialized dynamically allocated memory can be assumed not to happen in a defined execution, and any such reads that can happen in the conditions of the analysis should have been warned about by the companion value analysis. Since the value analysis already warns about such uses of dynamically allocated memory, the allocated block should rather be set to bottom (the empty set of effective types) for maximum accuracy.

The `MayAlias` constructor corresponds to the type attribute `__may_alias__` provided by the GCC[20] and Clang compilers to inform the optimizer that lvalues of a certain type are expected to be used to access memory of a different effective type.

The possibility that the types in stdint.h are mapped to "extended integer types" (in the sense of the C11 clause 6.2.5:4) can be taken into account by adding as many constructors as necessary to `integer_type`. This is particularly relevant for the types `int8_t` and `uint8_t` because a 8-bit extended integer type that these would have been defined as aliases of would not need to benefit from the exception for "character types" in 6.5:7[21].

Note that the effective types "member m of type int of . . . " and "int" are unordered. It may initially seem that the latter should be included in the former, but not doing so allows to distinguish the case of a pointer that can only point to a certain `struct` member m of type `int` from the case of a pointer that may point to a `struct` member m of type `int` or to an `int` variable, say, depending on the execution path that has been followed.

4.2   Notable Analysis Rules and Checks

Compared to, say, a more traditional value analysis, the followed aspects of the strict aliasing violation analysis deserve mention:

---

[20] `https://gcc.gnu.org/onlinedocs/gcc-4.0.2/gcc/Type-Attributes.html`
[21] See discussion at `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=66110`

– When an lvalue read access occurs in an expression being evaluated, the type of the lvalue is checked against the effective type contained by the memory model for the location being accessed. This is the check that detects a problem in the program `int x = 1; 0.0f + *(float*)&x;` and also in the program
`void *p = malloc(4); *(int*)p = 1; 0.0f + *(float*)p;`
– Upon assignment to an lvalue, if the location designated by the lvalue being assigned is a variable or a subobject of a variable, then the static type of the lvalue is checked against the type of the variable. This is the check that detects a problem in the program `int x; *(float*)&x = 1.0f;`.
– Union types are handled specially only in the case of an assignment directly to or a read directly from a variable. Outside these cases, `union` types are ignored: the effective types have to match as if there was no `union`. This is intended to catch the cases where Clang might optimize despite the `union` type.

4.3   A short example

```
int x;
FILE *stream = ...;
void *p = malloc(sizeof(int));
if (fread(p, sizeof(int), 1, stream) == 1)
  x = *(int *)p;
else
  /* ... */
```

After the third line of the example above, the allocated object pointed by `p` has no effective type. Assuming the call to `fread` succeeds, it sets the effective type of that memory zone to `FirstAccessType`. The pointed block having the effective type `FirstAccessType` results in the read access `*(int *)p` succeeding. Since the effective type of the memory zone pointed by `p` is `FirstAccessType`, the effective type of the expression `*(int *)p` is determined by the type of the lvalue, and thus automatically matches it: `IntegerType(Int)`.

5   Analyzing legacy C libraries for strict aliasing violations

The analysis summarized in Sec. 4 is implemented as a Frama-C plug-in[3]. It works best when applied on definite inputs. In these conditions, the value analysis plug-in[1] avoids false positives, and builds a result graph[4] that contains *all* the information that has been inferred about the execution of the program, so that the analyzers exploiting these results are not limited by any loss of information about the order in which program variables take their values.

Finally, the strict aliasing violation analysis is itself designed to assign exactly one effective type to each memory location, avoiding imprecisions and the resulting false positives, for programs applied to definite inputs resulting in bounded execution. "Subjective false positive" may exist, where compilers do not currently exploit a strict-aliasing-violating pattern, and it turns out to be impossible to convince the maintainer of the library that they are doing something wrong. As long as the C standard's definition of allowed memory accesses is as poorly formalized as it is, and as long as the standard's ambiguity is invoked as excuse for compilers to invent their own, undocumented, rules, these "subjective false positives" seem unavoidable.

5.1   Expat

We applied the strict aliasing analyzer described in this article to Expat, a widely-used C library for parsing XML. The first of several strict aliasing violation detected by our analyzer has been reported[22]. This violation is caused by uses of struct types with a common initial sequence as a poor man's subtyping, as is otherwise extremely common in object-oriented code written in C. In this case, the struct-with-common-initial-sequence pattern is used in an attempt at implementing a generic hash-table data structure.

```
typedef struct {
    char *name;
} NAMED;

typedef struct {
    char *name;
    char *rawName;
    /* [...] */
} TAG;

typedef struct {
    char *name;
    PREFIX *prefix;
    /* [...] */
} ELEMENT_TYPE;

typedef struct {
    NAMED **v;
    size_t size;
    /* [...] */
} HASH_TABLE
```

The two structs `TAG` and `ELEMENT_TYPE` have the same initial sequence as the struct the hashtable is nominally intended to store pointers to, `NAMED`. The `lookup` function retrieves an existing element, or, if none is found, allocates one of the size given as parameter. This new element's `name` member is initialized through the `NAMED` struct type:

```
static NAMED *
lookup(XML_Parser parser, HASH_TABLE *table, KEY name,
        size_t createSize)
{
    /* [...] find the element or resize the table */
    /* The element was not found into the table: create it. */
    table->v[i] = (NAMED *)table->mem->malloc_fcn(createSize);
    if (!table->v[i])
        return NULL;
    memset(table->v[i], 0, createSize);
    table->v[i]->name = name;
```

---

[22] https://sourceforge.net/p/expat/bugs/538/

```
    (table->used)++;
    return table->v[i];
}
```

In the analysis described in Sec. 4, the assignment `table->v[i]->name = name` sets the effective type of the memory location being written to "member name of the struct `NAMED`". This means that subsequent read accesses to this part of memory must be made through a pointer to the struct `NAMED`. Reading the memory location through a pointer to another struct may interact badly with compiler optimizations, as shown in the example functions `st1` and `st2`.

```
static ELEMENT_TYPE *
getElementType(XML_Parser parser, const ENCODING *enc,
               const char *ptr, const char *end)
{
    DTD * const dtd = _dtd;
    const XML_Char *name = poolStoreString(&dtd->pool, enc,
                                           ptr, end);
    ELEMENT_TYPE *ret;

    if (!name)
        return NULL;
    ret = (ELEMENT_TYPE *) lookup(parser, &dtd->elementTypes,
                                  name, sizeof(ELEMENT_TYPE));
    if (!ret)
        return NULL;
    if (ret->name != name) {
        ...
    } else {
        ...
    }
}
```

The function `getElementType` exemplifies how the library Expat uses the value returned by `lookup`. The member `name` is read through a pointer to the structure `ELEMENT_TYPE`. This leads to a violation of strict aliasing as shown by the following warning:

```
expat/lib/xmlparse.c:6470:[sa] warning: Reading a cell with
 effective type (struct __anonstruct_NAMED_13).name[char *]
 through the lvalue ret->name of type
 (struct __anonstruct_ELEMENT_TYPE_22).name[char *].
 Callstack: getElementType :: expat/lib/xmlparse.c:4080 <-
            doProlog :: expat/lib/xmlparse.c:3801 <-
            prologProcessor :: expat/lib/xmlparse.c:3618 <-
            prologInitProcessor :: expat/lib/xmlparse.c:1693 <-
            XML_ParseBuffer :: expat/xmlwf/xmlfile.c:184 <-
            processStream :: expat/xmlwf/xmlfile.c:243 <-
            XML_ProcessFile :: expat/xmlwf/xmlwf.c:853 <-
            main
```

As part of the code normalization in the analyzer's front-end, the anonymous structures receive names: in the example above, `struct __anonstruct_ELEMENT_TYPE_22` is the name given to `struct { char *name; PREFIX *prefix; ... }`.

The resolution of this bug report was to add `-fno-strict-aliasing`, a perfectly reasonable solution for legacy C code.

5.2   Zlib

We applied our strict aliasing analyzer to the general-purpose data compression library Zlib. One strict aliasing violation was found and reported, and appears as a comment in the source code[23]. The violation[24] is caused by accessing four `unsigned char` through a pointer to `unsigned int`:

```
#define DOLIT4 c ^= *buf4++; c = crc_table[3][c & 0xff] ^ \
           /* ... */
#define DOLIT32 DOLIT4; DOLIT4; DOLIT4; DOLIT4; DOLIT4;\
               DOLIT4; DOLIT4; DOLIT4

local unsigned long crc32_little(crc, buf, len)
    unsigned long crc;
    const unsigned char FAR *buf;
    unsigned len;
{
    register z_crc_t c;
    register const z_crc_t FAR *buf4;
    /* ... */
    buf4 = (const z_crc_t FAR *)(const void FAR *)buf;
    while (len >= 32) {
        DOLIT32;
        len -= 32;
    }
    while (len >= 4) {
        DOLIT4;
        len -= 4;
    }
    /* ... */
}
```

In the simplified pattern above, the type `z_crc_t` is defined as `unsigned int`. Our analyzer, when handling the statement `buf4 = (const z_crc_t FAR *)(const void FAR *)buf`, sets the effective type of the variable `buf4` to "pointer to `unsigned char`" by ignoring the pointer conversions. Accessing to the object through the pointer `buf4` is a violation of strict aliasing rules, as shown by the following warning of the analyzer:

```
zlib/crc32.c:267:[sa] warning: Reading a cell with effective type char
```

---

[23] https://github.com/madler/zlib/commit/e08118c401d5434b7b3a57039263f4f
a9b1f7d1a

[24] https://github.com/pascal-cuoq/zlib-fork/commit/d7cde11e0b44f4e97cc1f
d5250d826967841e614

```
through the lvalue *tmp_0(buf4) of type unsigned int.
Callstack: crc32_little :: zlib/crc32.c:224 <-
           crc32 :: zlib/inflate.c:1182 <-
           inflate :: zlib/gzread.c:191 <-
           gz_decomp :: zlib/gzread.c:248 <-
           gz_fetch :: zlib/gzread.c:347 <-
           gzread :: zlib/test/minigzip.c:439 <-
           gz_uncompress :: zlib/test/minigzip.c:540 <-
           file_uncompress :: zlib/test/minigzip.c:629 <-
           main
```

In the warning, the temporary variable `tmp_0`, introduced by code normalization, corresponds to the variable `buf4` at that point of the function `crc32_little`.

6   Related work

The closest forms of analyses we are aware of are `libcrunch`[7] and `SafeType`[5]. The tool `libcrunch` takes a dynamic approach and instruments pointer casts for violations to be revealed when executing. Since our analyzer handles whole-programs only and can behave as a C interpreter when deterministic inputs are provided, it is the most directly comparable of the two. Safetype is a static analysis implemented inside a compiler, that is, a modular static analysis that does not have access to the whole program. This in itself is a source of both false positives and false negatives.

Each of libcrunch and SafeType warns at the level of the pointer conversion, for instance when the address of an `int` ends up being converted to a `float*`. SafeType can also warn about memory accesses with the wrong type. Our analysis warns at the level of forbidden memory access only.

7   Conclusion

We have provided a number of examples showing the difficulty of analyzing C programs precisely and soundly for strict aliasing violations. We think that working from examples is crucial in this endeavor because the description of the rules in the C standards are particularly open to interpretation by both C developers and compiler authors.

An analyzer for strict aliasing violations is being implemented. Our target is legacy C code. We think that this justifies our chosen, and as far as we know, original approach of warning only for actual strict aliasing violations, as opposed to warning for suspicious uses of pointers that may not technically break the rules. Legacy code should not be modified willy-nilly: billions of systems may rely on it, and at the same time, this software is not always maintained by the original developer, or even actively maintained at all. Contrary to first appearances, a simple makefile change to explicitly disable strict aliasing optimizations is an extremely satisfying outcome after successfully identifying an illegal pattern with our analyzer. The analyzer can also help to eliminate the bad patterns one by one, tweaking the code until the analyzer eventually remains silent, but such is the respect due to legacy code that we do not expect this usage to be very common.

Out of 18000 Debian packages indexed by Debian Code Search[25], 1001 packages contain the string `-fno-strict-aliasing`, and 131 contain the string `may_alias`,

---

GCC's extension to get the benefits of the type-based alias analysis while informing the compiler that some specific memory accesses may be to a different effective type than expected. Our goal is to make every package that needs it use one of these two options. According to Debian Sources[26], 45% of the lines of code in Debian are written in C, so a lot of work remains after the first two successful analyses of Expat and Zlib.

References

1. Canet, G., Cuoq, P., Monate, B.: A value analysis for C programs. In: Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation. pp. 123–124. SCAM '09, IEEE Computer Society, Washington, DC, USA (2009), `http://dx.doi.org/10.1109/SCAM.2009.22`
2. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 238–252. POPL '77, ACM, New York, NY, USA (1977), `http://doi.acm.org/10.1145/512950.512973`
3. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: A software analysis perspective. In: Proceedings of the 10th International Conference on Software Engineering and Formal Methods. pp. 233–247. SEFM'12, Springer-Verlag, Berlin, Heidelberg (2012), `http://dx.doi.org/10.1007/978-3-642-33826-7_16`
4. Cuoq, P., Rieu-Helft, R.: Result graphs for an abstract interpretation-based static analyzer. To appear
5. Ireland, I.: SafeType: Detecting type violations for type-based alias analysis of C. Ph.D. thesis, University of Alberta (2013)
6. ISO: ISO/IEC 9899:2011 Information technology — Programming languages — C (Dec 2011), `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853`
7. Kell, S.: Dynamically diagnosing type errors in unsafe code. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 800–819. OOPSLA 2016, ACM, New York, NY, USA (2016), `http://doi.acm.org/10.1145/2983990.2983998`
8. Krebbers, R.: The C standard formalized in Coq. Ph.D. thesis, Radboud University (dec 2015)
9. Miné, A.: Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. SIGPLAN Not. 41(7), 54–63 (Jun 2006), `http://doi.acm.org/10.1145/1159974.1134659`
10. Siff, M., Chandra, S., Ball, T., Kunchithapadam, K., Reps, T.: Coping with Type Casts in C, pp. 180–198. Springer Berlin Heidelberg, Berlin, Heidelberg (1999), `http://dx.doi.org/10.1007/3-540-48166-4_12`
11. Yong, S.H., Horwitz, S., Reps, T.: Pointer analysis for programs with structures and casting. SIGPLAN Not. 34(5), 91–103 (May 1999), `http://doi.acm.org/10.1145/301631.301647`

---

[26] `https://sources.debian.net`