

a theory of platform-dependent low-level software

marius nita

dan grossman

craig chambers

In general, a language **semantics** is defined as a relation over program states:

$$P \rightarrow P'$$

But in C, some steps may be **platform-dependent**, e.g.

$$\text{sizeof}(\text{int}) \rightarrow ???$$

So we **incorporate platforms** into the step relation:

$$\Pi \vdash P \rightarrow P'$$

A Π is a collection of functions that capture a platform's type **layout policy**, for example:

$$\Pi.\text{sizeof}(\text{int}) = 4$$

$$\Pi.\text{alignof}(\text{double}) = 8$$

Then, an **analysis** A takes a program and yields a **layout constraint** C :

$$A(P) = C$$

C is a **logic formula**, e.g.:

$$\text{sizeof}(\text{int}) = 4 \wedge \text{alignof}(\text{short}) = 2$$

If C is true on a platform Π , we write

$$\Pi \models C$$

Our **key theorem** states that given a program P , if $A(P) = C$, then for all Π such that $\Pi \models C$, P will not fail with an unchecked error when run on platform Π .

Because type-layout dependencies are introduced by pointer-to-pointer casts, one corollary of our theorem is that **cast-free programs** do not fail with unchecked errors on any **sensible platform**.

Roughly, a sensible platform is faithful to the required parts of the C standard.

We have **implemented a tool** that takes C programs and warns about possibly unportable pointer-to-pointer casts. Our tool's analysis is static and does **not** require **physical access** to target platforms.

In one **case study**, our tool issued a warning about a cast from sp_time^* to timeval^* , as defined below:

```
struct sp_time { long sec; long usec; }
struct timeval { time_t tv_sec;
                 suseconds_t tv_usec; }
```

The **tool's output** is as follows:

```
events.c:150: sp_time * ==> timeval *
Host (GCC/32-bit X86):
Src: ptr_4(bbbb bbbb)
Dest: ptr_4(bbbb bbbb)
Target (GCC/LP-64):
Src: ptr_8(bbbbbbbb bbbbbbbb)
Dest: ptr_8(bbbbbbbb bbbb----
```

The type `suseconds_t` is an alias for `int`. On LP-64 platforms (64-bit long and 32-bit int), some bits in `usec` are lost in the cast. If the platform is big-endian, most bits are lost.

This bug is **impossible** to catch by **testing** on the host platform, and very hard to catch by testing on the target platform.

One **problem** with the C standard is that it leaves the layouts of types unspecified. Pointer casts can make **implicit assumptions** about type layout that are **not portable** from one platform to another. For example,

```
struct S { void *buf; int len; };
struct D { void *buf; size_t len; };
...
struct S ss[100];
struct D *ds = (struct D*)ss;
...
// treat ds[N].len as the length of ds[N].buf
```

This code is correct on platforms on which $\text{sizeof}(\text{int}) == \text{sizeof}(\text{size_t})$ and leads to out-of-bounds buffer accesses on others.

a WASP project
wasp.cs.washington.edu