SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

# Poisoned by the Compiler: Optimizations Gone Wrong

Christian Schreiner
John Regehr

Some optimizations don't hold for all data values. How to ensure correctness without sacrificing execution speed?

Strong semantics limit the compiler's ability to juggle code around to make it run faster. The LLVM compiler suite uses a concept called “poison values” to weaken the semantics in a controlled way, so many more optimizations become possible. Unfortunately, there are occasional subtle flaws, and they become much more visible with the next generation of compiler optimizations. This project aims to resolve these flaws.

Classic example of how an optimization can be subtly wrong:

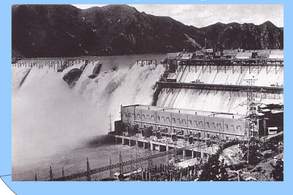
```
int compute_tax( int income )
{
   tax= income* FEDERAL_RATE/ 100 +
        income* STATE_RATE/ 100;
   return tax;
}
```

Compiles into:

```
int compute_tax( int income )
{
   // One division fewer! One multiplication fewer!
   tax= income* (FEDERAL_RATE+ STATE_RATE)/ 100;
   return tax;
}
```

**Question:** What happens when the business grows enough that the variable “income” exceeds 500 million?

(Assume a 32-bit integer length. The sum of the two tax rates is at least 10.)

Hint:  (i.e. catastrophic overflow)

The LLVM compiler suite designates this sort of undefined value as “poison”. It is an error to output a poisoned value, or a value derived from a poisoned value.

Example: `int32_t foo= 27 << 512;` creates a poison value.

Often the compiler finds speculative execution instructions will execute faster than the alternatives. (N.b. for the last 25+ years, branch instructions have been rather expensive.) Poison is how the compiler determines when it can ignore speculative overflows.

```
int16_t irrelevant_poison( int16_t xx, int16_t yy )
{
  int16_t result= xx;
  if ( yy < 7 ) {
    result= xx+ 7;
  } else {
    result= xx* 2+ 3;
  }
  return result;
}

int16_t foo= irrelevant_poison( 32750, -1 );
```

Compiles into:

```
int16_t irrelevant_poison( int16_t xx, int16_t yy )
{
   int16_t tmp1= xx+ 7;
   int16_t tmp2= xx*2+ 3;
   return (yy < 7) ? tmp1 : tmp2;
}

int16_t foo= irrelevant_poison( 32750, -1 );
```

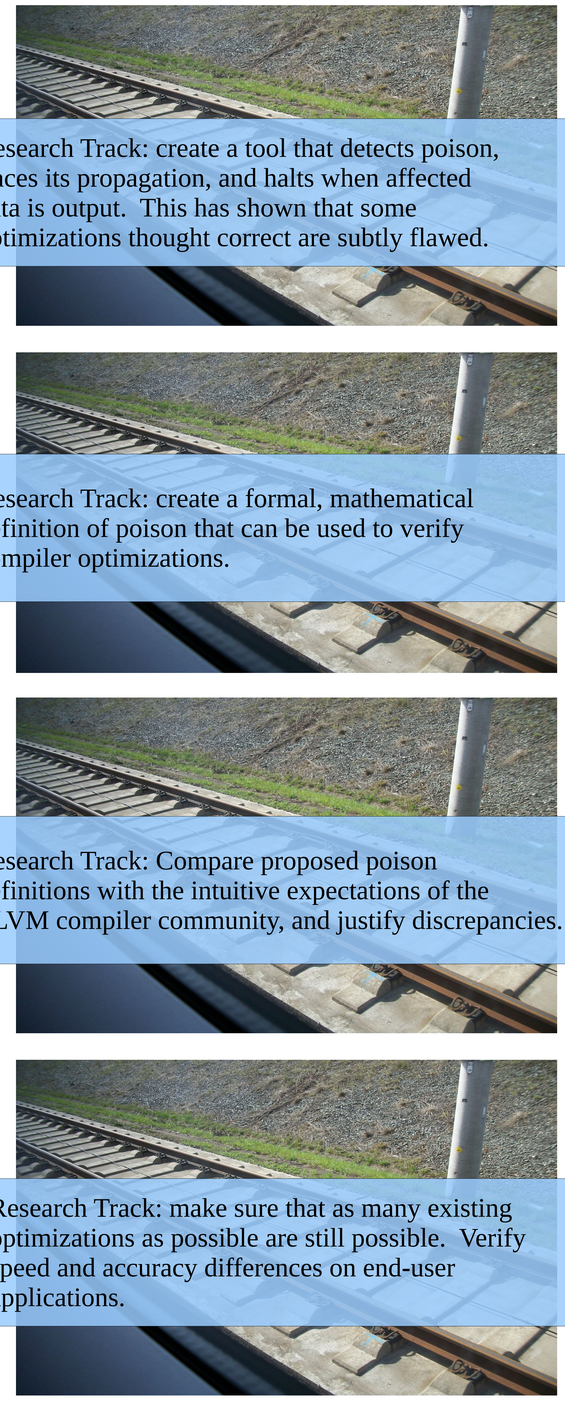
**Question:** Where is the poison created? Why doesn't it propagate?

**Challenge:** some definitions of poison are ambiguous.

```
int16_t seed= INT16_MAX- 1;

seed+= 5; // generates poison
seed= seed << 8;
seed= seed & 0xff;
```

Question: Should “seed” contain poison?



Research Track: create a tool that detects poison, traces its propagation, and halts when affected data is output. This has shown that some optimizations thought correct are subtly flawed.

Research Track: create a formal, mathematical definition of poison that can be used to verify compiler optimizations.

Research Track: Compare proposed poison definitions with the intuitive expectations of the LLVM compiler community, and justify discrepancies.

Research Track: make sure that as many existing optimizations as possible are still possible. Verify speed and accuracy differences on end-user applications.



## Acknowledgments

The LLVM Developer Mailing List discussions of “poison”, including entries by David Majnemer, Sanjoy Das, Nuno Lopes, Chandler Carruth, John Regehr, and many others, 2009-2015. http://lists.cs.uiuc.edu/mailman/listinfo/llvmdev

Poison symbol is the European Union standard toxic symbol, defined in EU Directive 67/548/EEC.

"Sui-ho Dam under construction" by Unknown - Japanese book "Showa History: History of Japanese colony" published by Mainichi Newspapers Company. Licensed under Public Domain via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:Sui-ho_Dam_under_construction.JPG#mediaviewer/File:Sui-ho_Dam_under_construction.JPG

"ICE TrainTrack" by Photnart - Own work. Licensed under CC0 via Wikimedia Commons http://commons.wikimedia.org/wiki/File:ICE_TrainTrack.JPG#mediaviewer/File:ICE_TrainTrack.JPG