Lecture 24

Subverting a Type System, Hiding Exploit in Compilers

turning bitflip into an exploit; bootstrapping

Ras Bodik
Shaon Barman
Thibaud Hottelier

Hack Your Language!
CS164: Introduction to Programming Languages and Compilers, Spring 2012
UC Berkeley

Announcement

Classroom presentations start on Thursday

See piazza for announcements and talk schedule

Today's outline: Two Parts

Safety guarantees we get from the type system

under what assumptions do we get privacy? (ie, which constructs need to be banned from the language)

how hardware failures can subvert type system guarantees

Hiding an exploit in a self-generating compiler

Bootstrapping the compiler

"teaching" the compiler a value that gets preserved as the compiler is recompiled

Private object fields

Recall the lecture on embedding OO into Lua

We created an object with a private field

- the private field could store a password that could be checked against a guessed password for equality but the stored password could not be leaked

Next slide shows the code

Object with a private field

```
// Usage of an object with private field

def safeKeeper = SafeKeeper(“164rocks”)
print safeKeeper.checkPassword(“164stinks”)  -->  False

function SafeKeeper (password)
    def pass_private = password

    def checkPassword (pass_guess) {
        pass_private == pass_guess
    }
    // return the object, which is a table
    { checkPassword = checkPassword }
}
```

Let’s try to read out the private field!

Assume I agree to execute any code you give me.
Can you print the password (without trying all passwords)?

```
def safeKeeper = SafeKeeper(“164rocks”)
def yourFun = <paste any code here>
// I am even giving you a ref to keeper
yourFun(safeKeeper)
```

Which features of the 164 language do we need to disallow to prevent reading of pass_private?

1. overriding == with our own method that prints its arguments
2. access to the environment of a function and printing the content of the environment

Same in Java

```
class SafeKeeper {
    private long pass_private;
    SafeKeeper(password) { pass_private = password }

    Boolean checkPassword (long pass_guess) {
        return pass_private == pass_guess
}   }

SafeKeeper safeKeeper = new SafeKeeper(920342094223942)
print safeKeeper.checkPassword(1000000000001)  -->  False
```

Redoing the exercise in Java illustrates that the issues exist in a statically typed language, too.

Challenge: how to read out the private field?

Different language. Same challenge.

```
SafeKeeper safeKeeper = new SafeKeeper(19238423094820)
<paste your code here; it can refer to ‘safeKeeper’>
```

Which features of Java do we need to disallow to prevent reading of pass_private?

read about the ability to read private fields with java reflection API

Poor attacker

It’s frustrating to the attacker that

(1) he holds a pointer *a* to the Java object, and

(2) knows that password is at address *a*+16 bytes

yet he can’t read out password_private from that memory location.

Why can’t any program read that field?

1. Type safety prevents variables from storing incorrectly-typed values.

   B b = new A()   disallowed by compiler unless A extends B

2. Array-bounds checks prevent buffer overflows
3. Can’t manipulate pointers (addresses) and hence cannot change where the reference points.

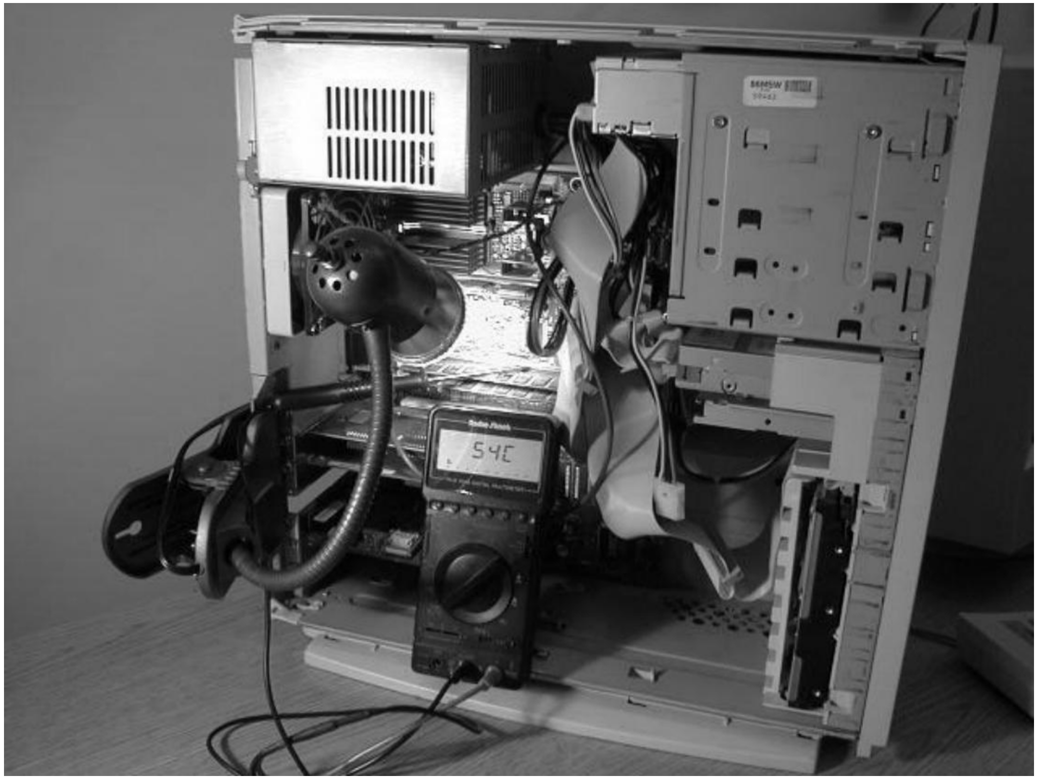
Together, these checks prevent execution of arbitrary user code…

Unless the computer breaks!

Memory Errors

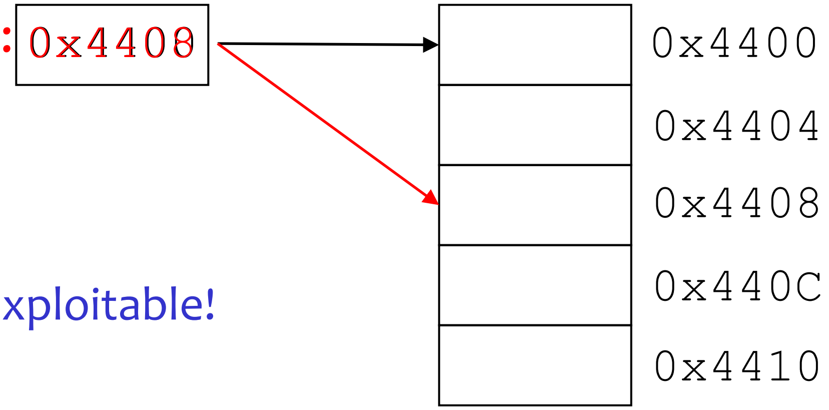
A flip of some bit in memory

- Can be caused by cosmic ray, or deliberately through radiation (heat)



Memory Errors

after bit 3 is flipped:



0x4408

0x4400

0x4404

0x4408

0x440C

0x4410

Exploitable!

Attack in C language

Before we describe the attack in Java, how would one forge (manufacture) a pointer in C

```
union {  int i; char * s; } u;
```

Here, i and s are names for the same location.

u.i = 1000

u.s[0] --> reads the character at address 1000

http://stackoverflow.com/questions/4748366/can-we-use-pointer-in-union

Overview of the Java Attack

Step 1: use a memory error to obtain two variables p and q, such that

1. p == q (i.e., p and q point to same memory loc) and
2. p and q have incompatible, custom static types

Cond (2) normally prevented by the Java type system.

Step 2: use p and q from Step 1 to write values into arbitrary memory addresses

- Fill a block of memory with desired machine code
- Overwrite dispatch table entry to point to block
- Do the virtual call corresponding to modified entry

The two Custom Classes For Step 1 Attack

```
class A {
  A a1;
  A a2;
  B b;    // for Step 1
  A a4;
  int i; // for address
         // in Step 2
}
```

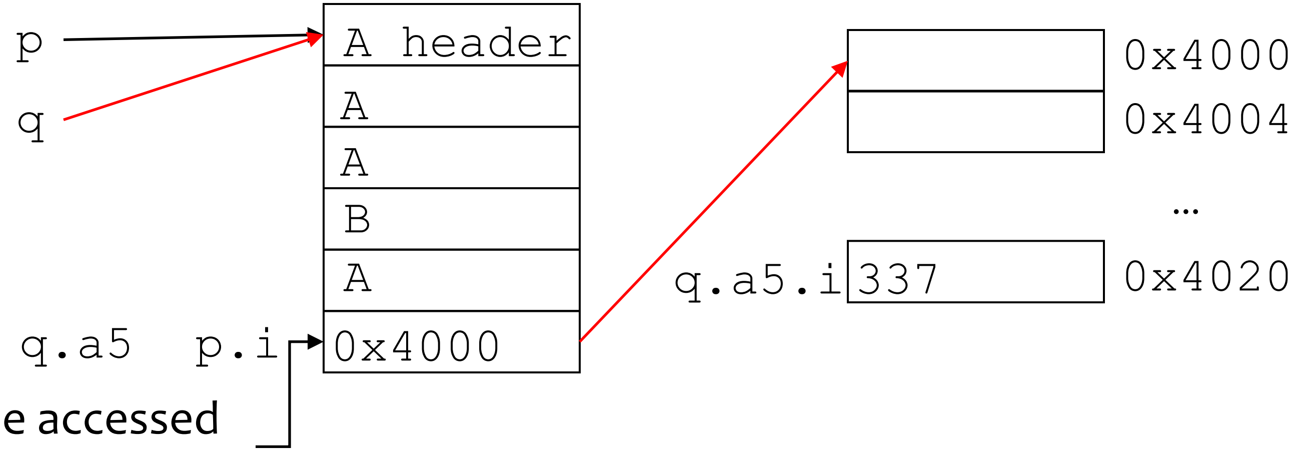
```
class B {
  A a1;
  A a2;
  A a3;
  A a4;
  A a5;
}
```

Assume 3-word object header

Step 2 (Writing arbitrary memory)

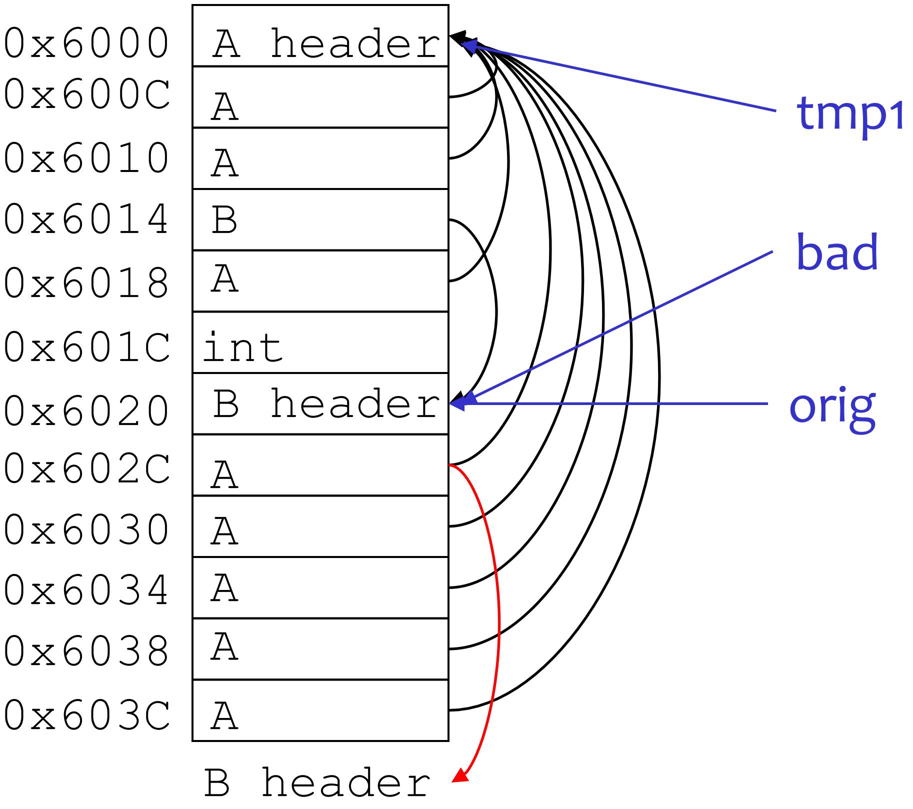
```
int offset = 8 * 4;     // Offset of i field in A
A p; B q;               // Initialized in Step 1, p == q;
                        // assume both p and q point to an A

void write(int address, int value) {
  p.i = address – offset;
  q.a5.i = value; // q.a5 is an integer treated as a pointer
}
```

Example: write 337 to address 0x4020



this location can be accessed as both q.a5 and p.i

Step 1 (Exploiting The Memory Error)



0x6000	A header
0x600C	A
0x6010	A
0x6014	B
0x6018	A
0x601C	int
0x6020	B header
0x602C	A
0x6030	A
0x6034	A
0x6038	A
0x603C	A
	B header

tmp1

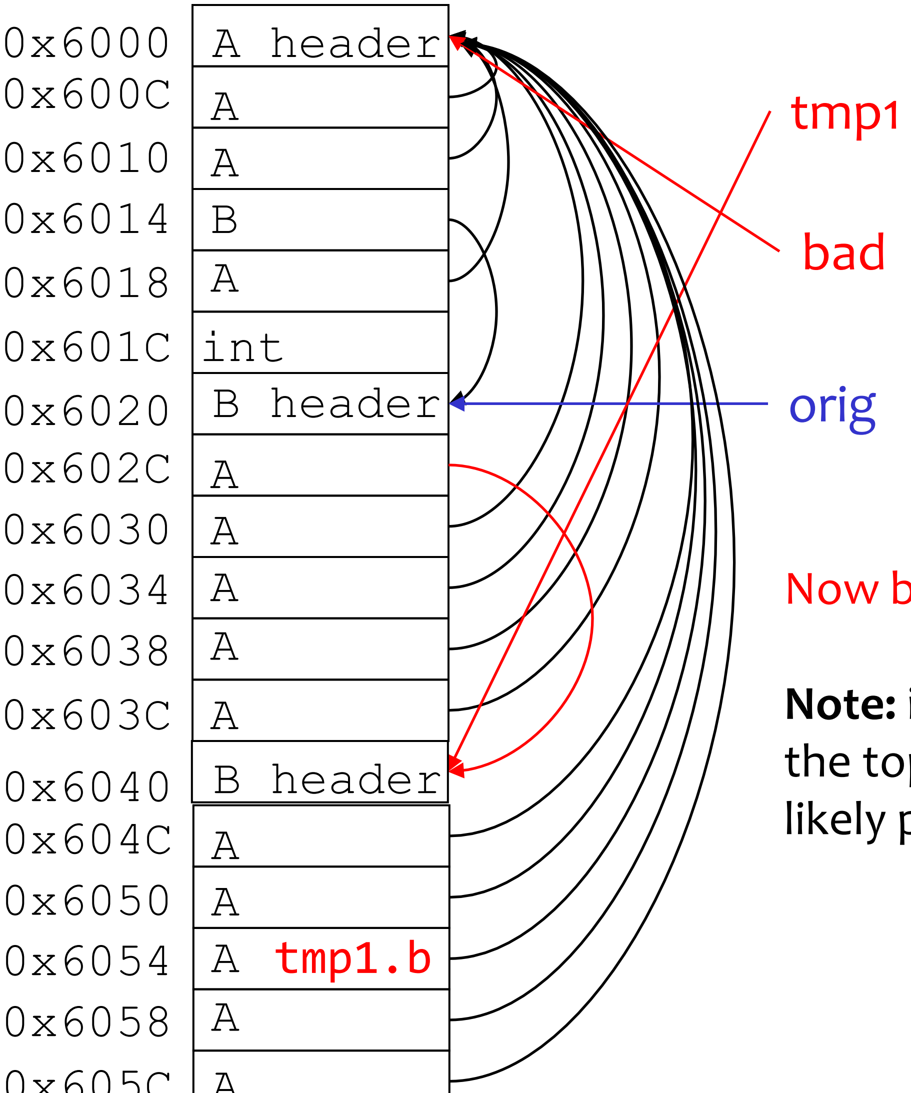
bad

orig

```
B orig;
A tmp1 = orig.a1;
B bad = tmp1.b;
```

The heap has one A object, many B objects. All fields of type A point to the only A object that we need here. Place this object close to the many B objects.

Step 1 (Exploiting The Memory Error)



```
B orig;
A tmp1 = orig.a1;
B bad = tmp1.b;
```

flip bit 0x40 in orig.a1

Now bad points to an A object!

Note: it is a coincidence that orig.a points to the top of the object header. It could equally likely point into a an object of type B.

Step 1 (cont)

Iterate until you discover that a flip happened.

```
A p; // pointer to single A object
while (true) {
  for (int i = 0; i < b_objs.length; i++) {
    B orig = b_objs[i];

    A tmp1 = orig.a1; // Step 1, really check all fields
    B q = tmp1.b;

    Object o1 = p; Object o2 = q; // check if we found a flip
    if (o1 == o2) {
      writeCode(p,q); // now we’re ready to invoke Step 2
} } }
```

Results (Govindavajhala and Appel)

With software-injected memory errors, took over both IBM and Sun JVMs with 70% success rate

think why not all bit flips lead to a successful exploit

Equally successful through heating DRAM with a lamp

Defense: memory with error-correcting codes

- ECC often not included to cut costs

Most serious domain of attack is smart cards

Reflections on Trusting Trust

a Berkeley graduate, former cs164 student (maybe :-)

better known for his work on Unix

Ken Thompson, Turing Award, 1983

we also know him for his regex-to-NFA compilation

Stage I: What does this program print?

```
char s[] = {
' ', '0', ' ', '}', ';', '\n', '\n', '/', '*', ' ', 'T', …, 0 };

/* The string is a representation of the body of this program
 * from '0' to the end.
 */

main() {
   int i;
   printf("char s[] = {\n");
   for (i=0; s[i]; i++) printf("%d, ", s[i]);
   printf("%s", s);
}
```

Stage I Lesson

- The array of chars in green is the “DNA” that allows the program to reproduce itself forever.
- So, each time the program prints itself, it needs to print this array, so that it can print itself again

Stage II: A portable compiler

A compiler for C can compile itself

- because the compiler is written in C

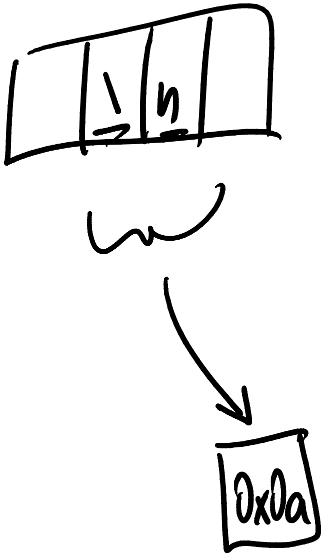
It is therefore portable to other platforms.

- Just recompile it on the new platform.

Stage II: An example of a portable feature

How Compiler Translates Escaped Char Literals:

```
…
c = next();
if (c != ‘\\’) return c;
c = next();
if (c == ‘\\’) return ‘\\’;
if (c == ‘n’) return ‘\n’;
…
```



Note that this is portable code:

- ‘\n’ is 0x0a on an ASCII platform but 0x15 on EBDIC
- the same compiler code will work correctly on both

Stage II: the bootstrapping problem

You want to extend the language with the ‘\v’ literal

- which can again be *n* on one machine on *m* on another
- you want to write into the compiler the portable expression ‘\v’

```
c = next();
if (c != ‘\\’) return c;
c = next();
if (c == ‘\\’) return ‘\\’;
if (c == ‘n’) return ‘\n’;
if (c == ‘v’) return ‘\v’;
```

Stage II: solving the bootstrapping problem

Your compiled (.exe) compiler does not accept \v, so you teach it:

- write this code first, compile it, and make it your binary C compiler
  - now your exe compiler accepts \v in input programs
- then edit 11 to ‘\v’ in the compiler source code
  - now your compiler source code is portable
- how about other platforms?

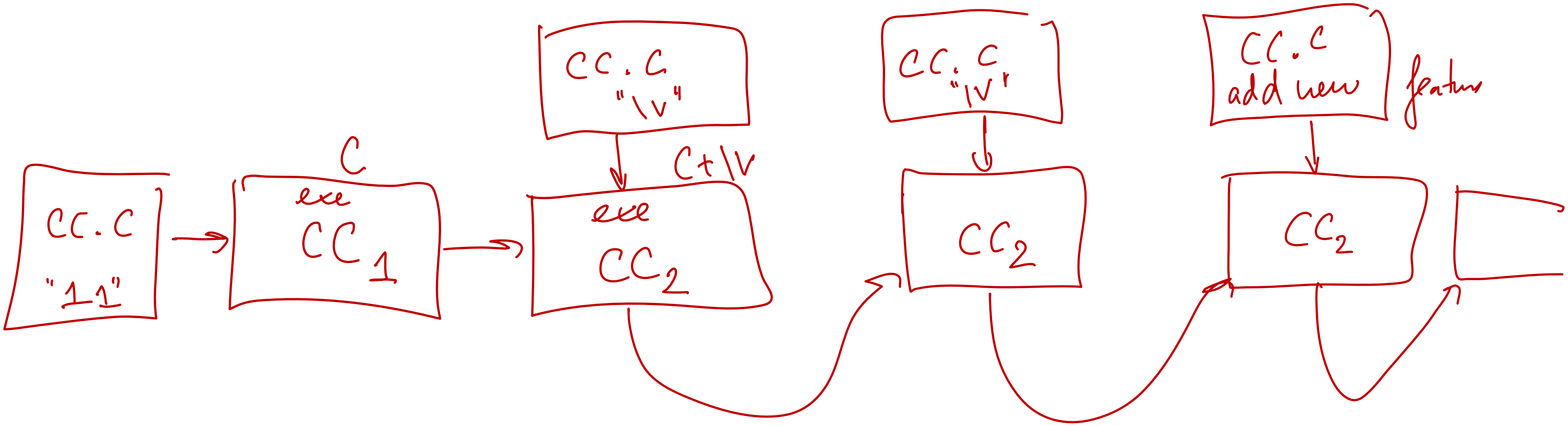
```
c = next();
if (c != ‘\\’) return c;
c = next();
if (c == ‘\\’) return ‘\\’;
if (c == ‘n’) return ‘\n’;
if (c == ‘v’) return 11;
```

Stage II: discussion

By compiling ‘\v’ into 11 just **once**, we **taught** the compiler forever that ‘\v’ == 11 (on that platform).

The term “taught” is not too much of a stretch

- no matter how many times you now recompile the compiler, it will perpetuate the knowledge



Stage III

This is a routine that compiles one line of source code

```
compile(char s[]) {
  …
}
```

Unix utils are written in C and compiled with the C compiler.

He who controls the compiler …

Stage III

This is a routine that compiles one line of source code

```
compile(char s[]) {
  if (match(s, “pattern”)) {
     compile(“bug”);
     return;
  }
}
```

translates pattern to bug

What is an interesting “pattern” and “bug”

Stage III

You can make the login program accept your secret pswd

```
pattern:  if(hash(pswd)==stored[user])

bug: if(hash(pswd)==stored[user] || \
        hash(pswd)==8132623192L)
```

Thompson created a backdoor into Unix, by having the compiler hack the login program

Stage III

This hack of the compiler would be easy to spot by reading the compiler code. So, the trick is this:

```
compile(char s[]) {
  if (match(s, “pattern1”)) {
     compile(“bug1”); return;
  }
  if (match(s, “patter2”)) {
     compile(“bug2”); return;
  }
}
```

What is an interesting “pattern” and “bug”

Stage III

The second pattern/bug is triggered when the compiler compiles itself

- it compiles the clean compile function into the one with the two hacks

The first pattern/bug triggered when the compiler compiles login.c

- as before

Your excercise

Figure out what pattern2/bug2 needs to be

How resilient is Thompson’s technique to changes in the compiler source code? Will it work when someone entirely rewrites the compiler?

Summary

PL knowledge useful beyond language design and implementation

Helps programmers understand the behavior of their code

Compiler techniques can help to address other problems like security (big research area)

Safety and security are hard

- Assumptions must be explicit