The Correctness-Security Gap in Compiler Optimization

Vijay D'Silva, Mathias Payer, Dawn Song

LangSec 2015

1	Compilers and Trust
2	Correctness vs. Security by Example
3	Correctness vs. Security, Formally
4	Future Directions

Appendix: HISTORICAL REMARKS ON COMPILER CONSTRUCTION

F. L. Bauer

Technical University of Munich

Munich, Germany

Historical Remarks on Compiler Construction

D. E. KNUTH [81] has observed (in 1962!) that the early history of cimpiler construction is difficult to assess. Maybe this, or maybe the general unhistorical attitude of our century is responsible for the widespread ignorance about the origins of compiler construction. In addition, the overwhelming lead of the USA in the general development of computers and their application, together with the language barrier, has in fact favoured negligence of early developments in Middle Europe and in the Soviet Union.

Reflections on Trusting Trust

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

KEN THOMPSON

pile" is called to compile the next line of source. Figure 3.2 shows a simple modification to the compiler that will deliberately miscompile source whenever a particular pattern is matched. If this were not deliberate, it would be called a compiler "bug." Since it is deliberate, it should be called a "Trojan horse."

The actual bug I planted in the compiler would match code in the UNIX "login" command. The replacement code would miscompile the login command so that it would accept either the intended encrypted password or a particular known password. Thus if this code were installed in binary and the binary were used

MORAL

The moral is obvious. You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) No amount of source-level verification or scrutiny will protect you from using untrusted code. In demonstrating the possibility of this kind of attack, I picked on the C compiler. I could have picked on any program-handling program such as an assembler, a loader, or even hardware microcode. As the level of program gets lower, these bugs will be harder and harder to detect. A well-installed microcode bug will be almost impossible to detect

CORRECTNESS OF A COMPILER FOR ARITHMETIC EXPRESSIONS*

JOHN McCARTHY and JAMES PAINTER

1967

1 Introduction

This paper contains a proof of the correctness of a simple compiling algorithm for compiling arithmetic expressions into machine language.

The definition of correctness, the formalism used to express the description of source language, object language and compiler, and the methods of proof are all intended to serve as prototypes for the more complicated task of proving the correctness of usable compilers. The ultimate goal, as outlined in references [1], [2], [3] and [4] is to make it possible to use a computer to check proofs that compilers are correct.

Testing	Verification
Whalley, '94, vpoiso	
McKeeman, '98	Goerigk, '00 (in ACL2)
McPeak & Wilkerson, '03, Delta	Lacey et al. '02
	Lerner et al. '05, Rhodium
Yang et al. '11, C-Smith	Leroy et al. '06, CompCert
Regehr et al. '12, C-Reduce	Tatlock & Lerner, '10, XCert
Taming Compiler Fuzzers, PLDI '13	*Formal Verification of a Realistic Compiler*, CACM '08

Formal Verification of a Realistic Compiler

By Xavier Leroy

Communications of the ACM, Vol. 52 No. 7, Pages 107-115
10.1145/1538788.1538814

Comments

VIEW AS: SHARE:

This paper reports on the development and formal verification (proof of semantic preservation) of CompCert, a compiler from Clight (a large subset of the C programming language) to PowerPC assembly code, using the Coq proof assistant both for programming the compiler and for proving its correctness. Such a verified compiler is useful in the context of critical software and its formal verification: the verification of the compiler guarantees that the safety properties proved on the source code hold for the executable compiled code as well.

Back to Top

1. Introduction

Can you trust your compiler? Compilers are generally assumed to be semantically transparent: the compiled code should behave as prescribed by the semantics of the source program. Yet, compilers—and especially optimizing compilers—are complex programs that perform complicated symbolic transformations. Despite intensive testing, bugs in compilers do occur, causing the compilers to crash at compile-time or—much worse—to silently generate an incorrect executable for a correct source program.

1	A very brief history of compiler correctness
2	Correctness vs. Security by Example
3	Correctness vs. Security, Formally
4	Future Directions

Dead Store Elimination

```
#include <string>
using std::string;

#include <memory>

// The specifics of this function are
// not important for demonstrating this bug.
const string getPasswordFromUser() const;

bool isPasswordCorrect() {
    bool isPasswordCorrect = false;
    string Password("password");

    if(Password == getPasswordFromUser()) {
        isPasswordCorrect = true;
    }

    // This line is removed from the optimized code
    // even though it secures the code by wiping
    // the password from memory.
    memset(Password, 0, sizeof(Password));

    return isPasswordCorrect;
}
```

From the GCC mailing list, 2002
https://gcc.gnu.org/bugzilla/show_bug.cgi?id=8537

From: "Joseph D. Wagner" <wagnerjd@prodigy.net>
To: <fw@gcc.gnu.org>,
<gcc-bugs@gcc.gnu.org>,
<gcc-prs@gcc.gnu.org>,
<nobody@gcc.gnu.org>,
<wagnerjd@prodigy.net>,
<gcc-gnats@gcc.gnu.org>
Cc:
Subject: RE: optimization/8537: Optimizer Removes Code Necessary for Security
Date: Sun, 17 Nov 2002 08:59:53 -0600

Direct quote from:
http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Bug-Criteria.html

"If the compiler produces valid assembly code that does not correctly execute the input source code, that is a compiler bug."

So to all you naysayers out there who claim this is a programming error or poor coding, YES, IT IS A BUG!

From the GCC mailing list, 2002
https://gcc.gnu.org/bugzilla/show_bug.cgi?id=8537

Motivating Questions:

Can a formally verified, correctly implemented compiler optimization introduce a security bug not present in the source?

Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rules

PLDI 2003

Sorin Lerner
Univ. of Washington
lerns@cs.washington.edu

Todd Millstein
UCLA
todd@cs.ucla.edu

Erika Rice

Craig Chambers

Proving Correctness of Compiler Optimizations by Temporal Logic *

POPL 2002

David Lacey   Neil D. Jones   Eric Van Wyk   Carl Christian Frederiksen

Formal Certification of a Compiler Back-end

_or_: Programming a Compiler with a Proof Assistant

POPL 2006

Xavier Leroy

Simple Relational Correctness Proofs for Static Analyses and Program Transformations (revised)

POPL 2004

Nick Benton
Microsoft Research

Motivating Questions:

Can a formally verified, correctly implemented compiler optimization introduce a security bug not present in the source?

YES!

This is the Correctness-Security Gap

Motivating Questions:

Can a formally verified, correctly implemented compiler optimization introduce a security bug not present in the source?

YES!

How prevalent is the problem?
Also, what gives?

Memory Persistence	Dead store elimination Function call inlining Code motion
Side Channels	Subexpression Elimination Strength reduction Peephole Optimizations
Language Specifics	Undefinedness in C/C++ Memory model issues Synchronization issues

Function Call Inlining

```
char *getPWHash() {
// code performing a secure computation
// assuming a trusted execution environment.
}
void compute() {
  // local variables
long i, j;
char *sha;
// Code in this function does not assume
// a trusted execution environment.
...
//call secure function
sha=getPWHash();

...
}
```

(from the paper)

Side Channels

```
*SCALE=\(2); # 2 or 8, that is the question:-) Value of 8 results
# in 16KB large table, which is tough on L1 cache, but eliminates
# unaligned references to it. Value of 2 results in 4KB table, but
# 7/8 of references to it are unaligned. AMD cores seem to be
# allergic to the latter, while Intel ones - to former [see the
# table]. I stick to value of 2 for two reasons: 1. smaller table
# minimizes cache trashing and thus mitigates the hazard of side-
# channel leakage similar to AES cache-timing one; 2. performance
# gap among different µ-archs is smaller.
...
&set_label("roundsdone",16);
    &mov    ("esi",&DWP(0,"ebx"));      # reload argument block
    &mov    ("edi",&DWP(4,"ebx"));
    &mov    ("eax",&DWP(8,"ebx"));

    for($i=0;$i<8;$i++) { &pxor(@mm[$i],&QWP($i*8,"edi")); }     # L^=inp
    for($i=0;$i<8;$i++) { &pxor(@mm[$i],&QWP($i*8,"esi")); }     # L^=H
    for($i=0;$i<8;$i++) { &movq(&QWP($i*8,"esi"),@mm[$i]); }     # H=L

    &lea    ("edi",&DWP(64,"edi"));    # inp+=64
    &sub    ("eax",1);             # num--
    &jz (&label("alldone"));
    &mov    (&DWP(4,"ebx"),"edi");      # update argument block
    &mov    (&DWP(8,"ebx"),"eax");
    &jmp    (&label("outerloop"));
```

https://github.com/openssl/openssl/blob/e0fc7961c4fbd27577fb519d9aea2dc788742715/crypto/whrlpool/asm/wp-mmx.pl

Common Subexpression Elimination

```c
int crypt(int k*){
int key = 0;
if (k[0]==0xC0DE){
  key=k[0]*15+3;
  key+=k[1]*15+3;
  key+=k[2]*15+3;
} else {
  key=2*15+3;
  key+=2*15+3;
  key+=2*15+3;
}
```

```c
int crypt(int k*){
int key = 0;
if (k[0]==0xC0DE){
  key=k[0]*15+3;
  key+=k[1]*15+3;
  key+=k[2]*15+3;
} else {
  // replaced by
  tmp = 2*15+3;
  key = 3*tmp;
}
```

(from the paper)

Undefinedness (null dereferences)

```
static unsigned int
tun_chr_poll(struct file *file,
poll_table * wait)
{
  struct tun_file *tfile = file-
>private_data;
  struct tun_struct *tun =
__tun_get(tfile);
  struct sock *sk = tun->sk;
  unsigned int mask = 0;

  if (!tun)
    return POLLERR;

   ...
}
```

→

```
static unsigned int
tun_chr_poll(struct file *file,
poll_table * wait)
{
  struct tun_file *tfile = file-
>private_data;
  struct tun_struct *tun =
__tun_get(tfile);
  struct sock *sk = tun->sk;
  unsigned int mask = 0;

    return POLLERR;

   ...
}
```

Fun with NULL pointers, part 1

By **Jonathan Corbet**
July 20, 2009

By now, most readers will be familiar with the local kernel exploit recently posted by Brad Spengler. This vulnerability, which affects the 2.6.30 kernel (and a test version of the RHEL5 "2.6.18" kernel), is interesting in a number of ways. This article will look in detail at how the exploit works and the surprising chain of failures which made it possible.

http://lwn.net/Articles/341773/

Date Mon, 7 May 2007 11:55:15 -0700 (PDT)
From Linus Torvalds <>
Subject Re: [patch] CFS scheduler, -v8

On Mon, 7 May 2007, Johannes Stezenbach wrote:
>
> One baffling example where gcc rewrites code is when
> conditionals depend on signed integer overflow:

Yes. This is one of my favourite beefs with gcc. Some of the optimization decisions seem to make no sense.

Your example is a good one, but my private beef has been in alias handling. Alias analysis is an important part of optimization, and there's two kinds: the static (and exact, aka "safe") kind that you can do regardless of any language definitions, because you *know* that you aren't actually changing behaviour, and the additional type-based heuristics that the C language allows.

So which ones would you expect a compiler to consider more important?

And which one do you think gcc will use?

Right. You can have static analysis that *guarantees* that two objects alias, but if gcc determins that they have different types and thus might not alias, it decides to use the heuristic instead of the firm knowledge, and generate code that doesn't work.

"Because the language definition allows it".

Oh well.

Linus

https://lkml.org/lkml/2007/5/7/213

SOSP 2013

Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior

Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama
MIT CSAIL

Abstract

This paper studies an emerging class of software bugs called *optimization-unstable code*: code that is unexpectedly discarded by compiler optimizations due to undefined behavior in the program. Unstable code is present in many systems, including the Linux kernel and the Postgres database. The consequences of unstable code range from incorrect functionality to missing security checks.

To reason about unstable code, this paper proposes a novel model, which views unstable code in terms of optimizations that leverage undefined behavior. Using this model, we introduce a new static checker called STACK that precisely identifies unstable code. Applying STACK to widely used systems has uncovered 160 new bugs that have been confirmed and fixed by developers.

```
char *buf = ...;
char *buf_end = ...;
unsigned int len = ...;
if (buf + len >= buf_end)
    return;   /* len too large */
if (buf + len < buf)
    return;   /* overflow, buf+len wrapped around */
/* write to buf[0..len-1] */
```

Figure 1: A pointer overflow check found in several code bases. The code becomes vulnerable as gcc optimizes away the second `if` statement [13].

unstable code happens to be used for security checks, the optimized system will become vulnerable to attacks.

This paper presents the first systematic approach for reasoning about and detecting unstable code. We implement this approach in a static checker called STACK, and use it to show that unstable code is present in a wide

1	A very brief history of compiler correctness
2	Correctness vs. Security by Example
3	Correctness vs. Security, Formally
4	Future Directions

Observations

Compiler correctness proofs show that the "behaviour" of the code is the same before and after a transformation.

Behaviour is defined as some observable aspect of execution, typically state.

Execution is defined with respect to a hypothetical abstract machine.

A Simple, Correct Transformation

```c
int increment(int a) {
  int b = a;
  b++;
  return b;
}
```

```c
int increment(int a) {


  return a + 1;
}
```

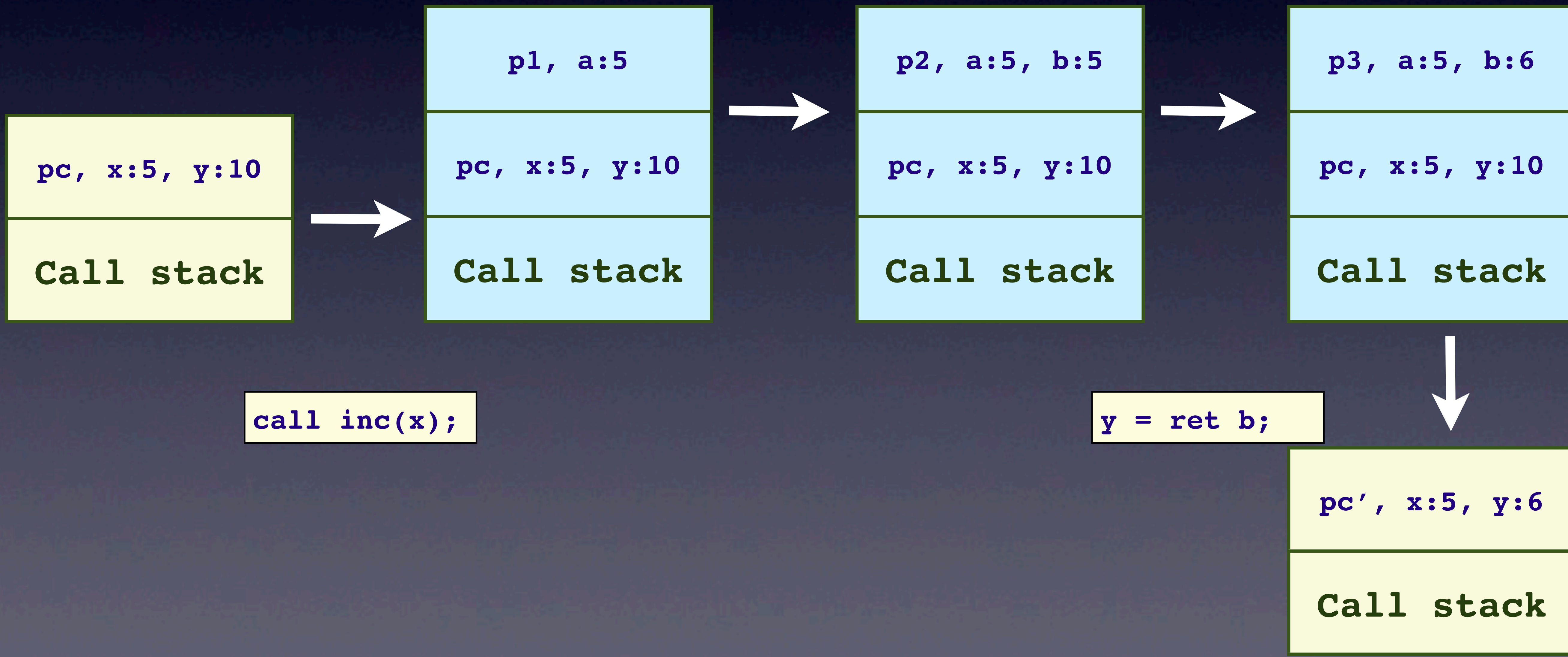
A Simple, Correct Transformation

```
int increment(int a) {
  int b = a;
  b++;
  return b;
}
```

→

```
int increment(int a) {


  return a + 1;
}
```

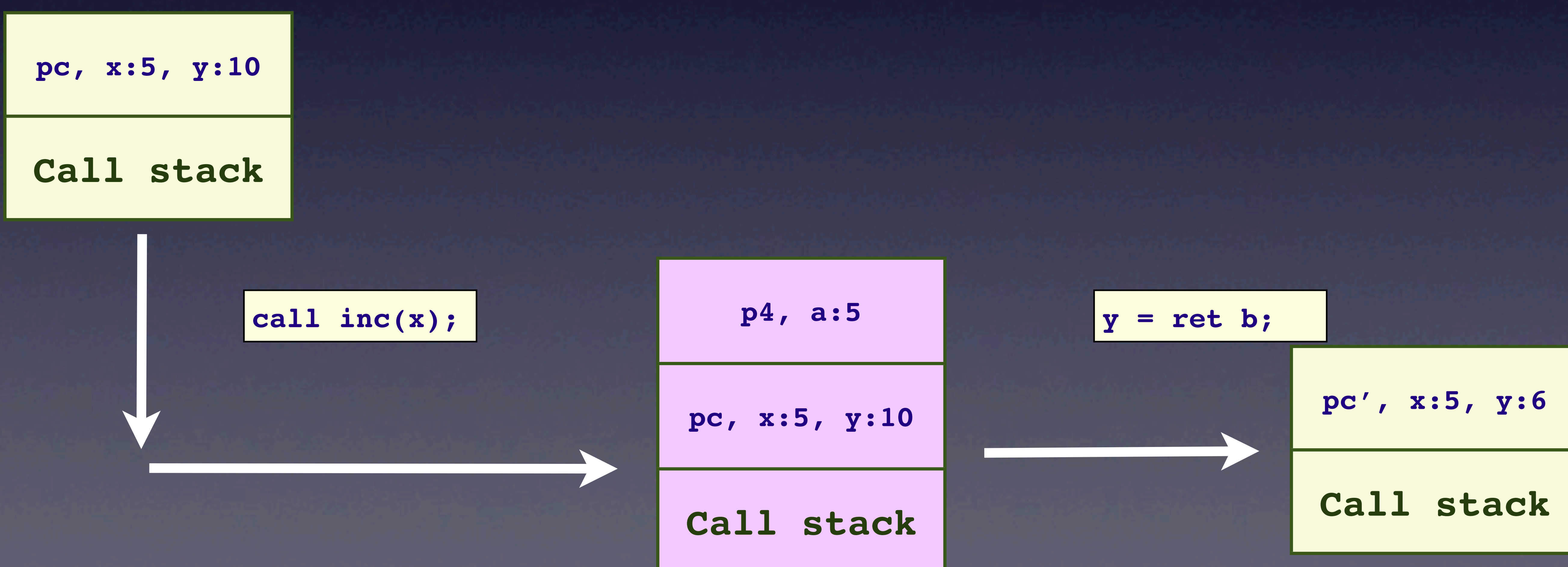


A Simple, Correct Transformation

```c
int increment(int a) {
  int b = a;
  b++;
  return b;
}
```

```c
int increment(int a) {


  return a + 1;
}
```

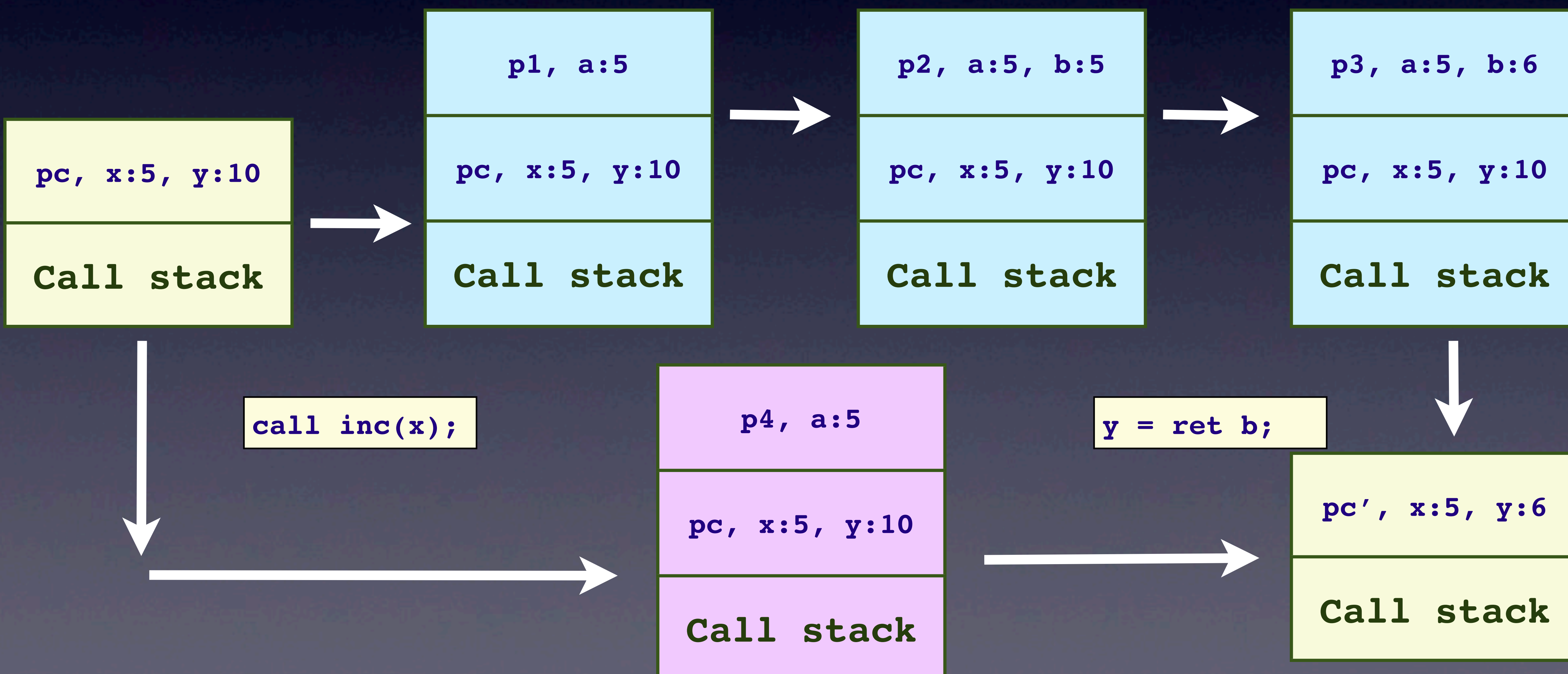


A Simple, Correct Transformation

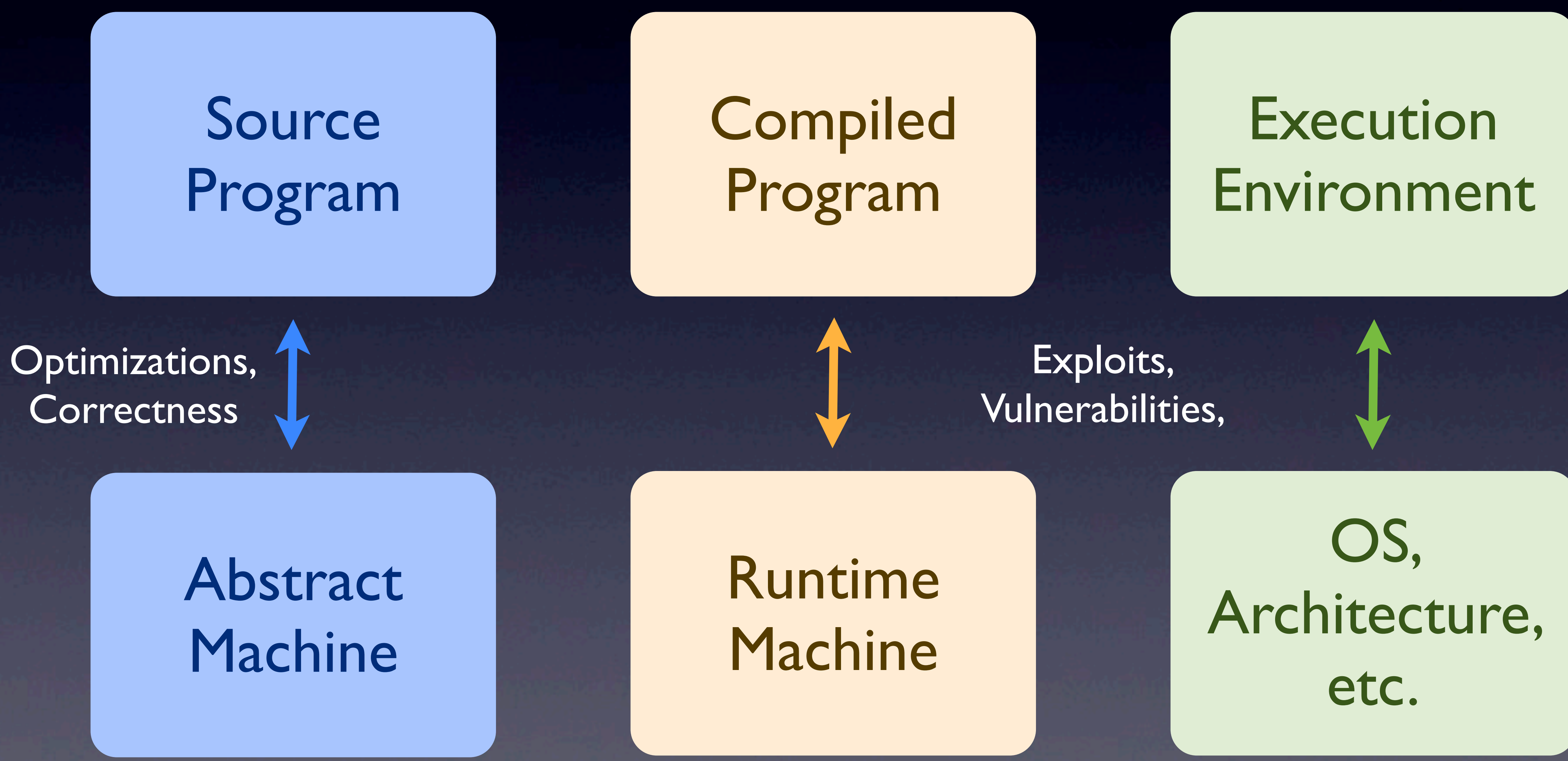
```
int increment(int a) {
  int b = a;
  b++;
  return b;
}
```

```
int increment(int a) {


  return a + 1;
}
```



Compiler Writer vs. Attacker



More Observations

Attackers reason about details (residual state, timing, etc.) not modelled by the abstract semantics machine.

Correctness guarantees do not preserve security because those exploits are not even possible in the machine used in proofs!

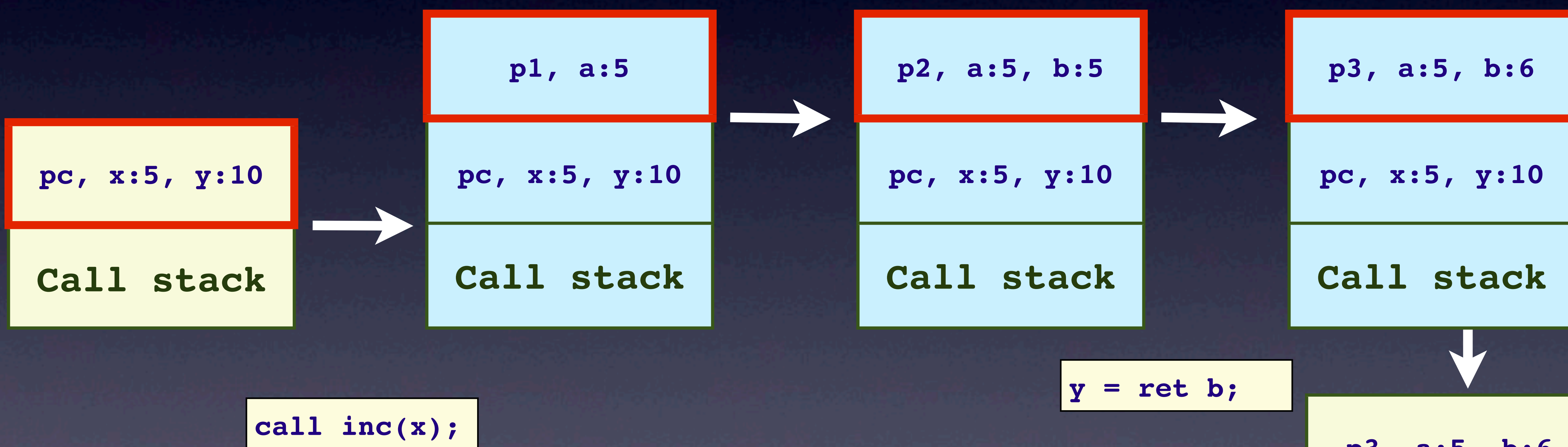
However ...

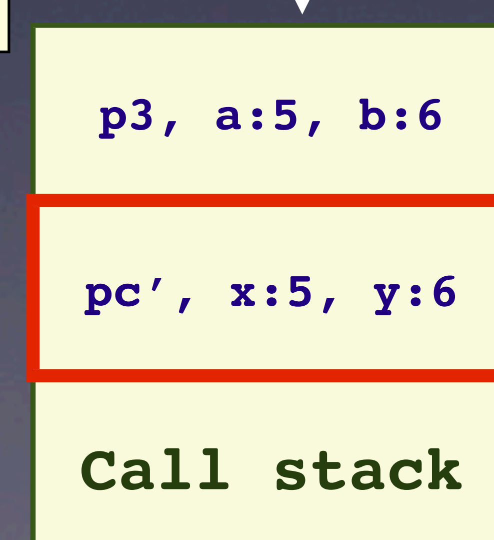
A Less Abstract Execution

```
int increment(int a) {
  int b = a;
  b++;
  return b;
}
```

```
int increment(int a) {


  return a + 1;
}
```





Call stack is closer to implementation.

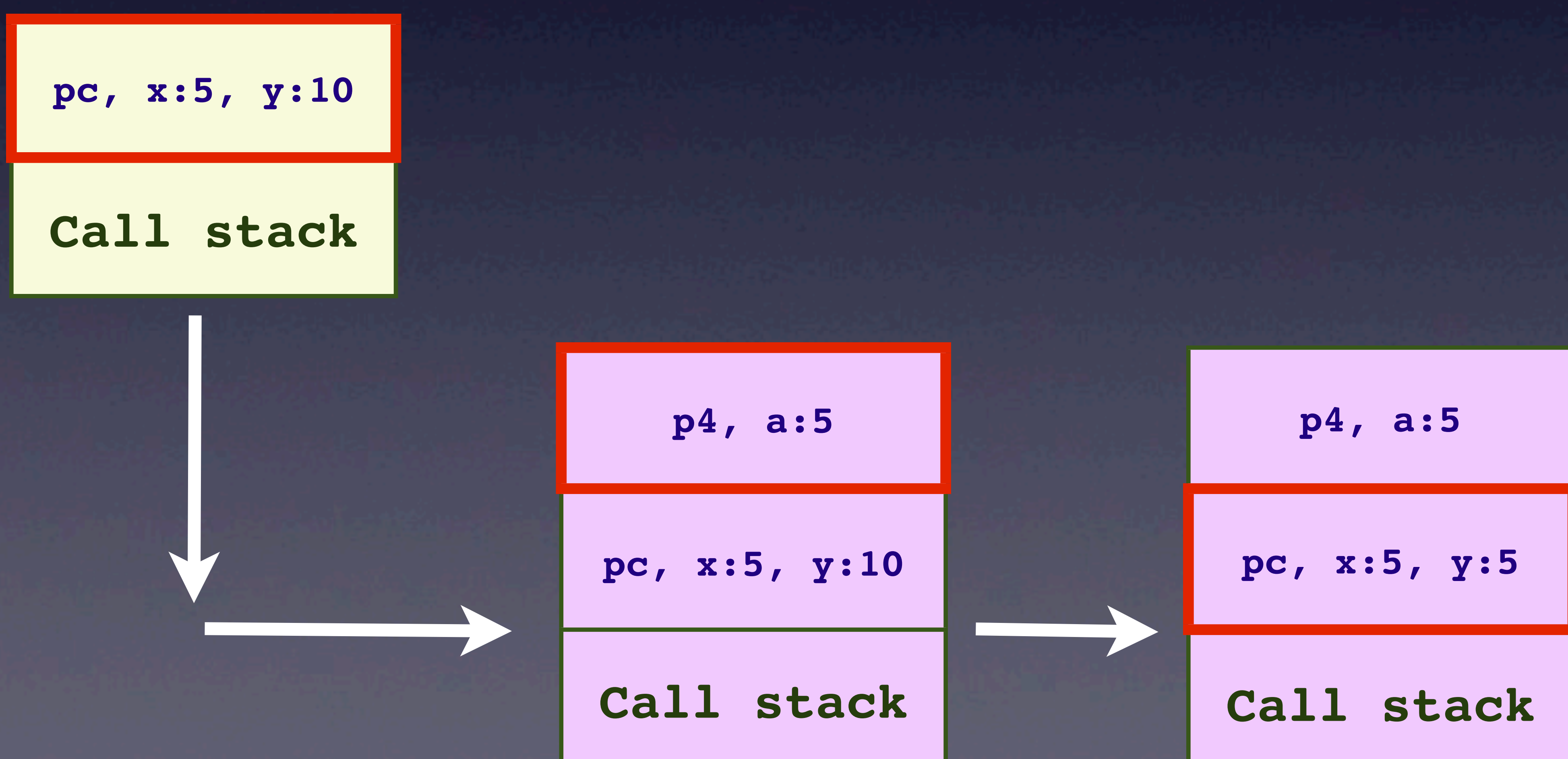
A Simple, Correct Transformation

```c
int increment(int a) {
  int b = a;
  b++;
  return b;
}
```

→

```c
int increment(int a) {


  return a + 1;
}
```

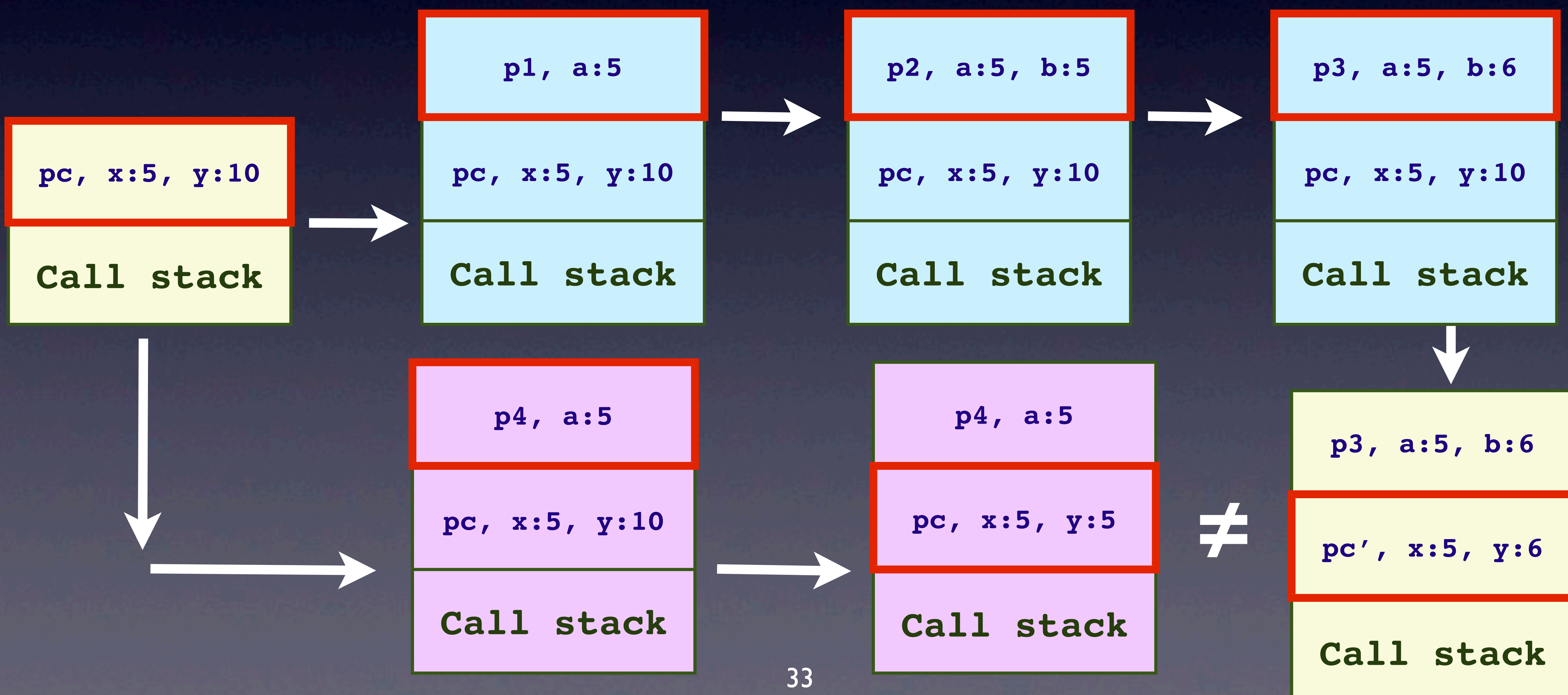


A Less Abstract Execution

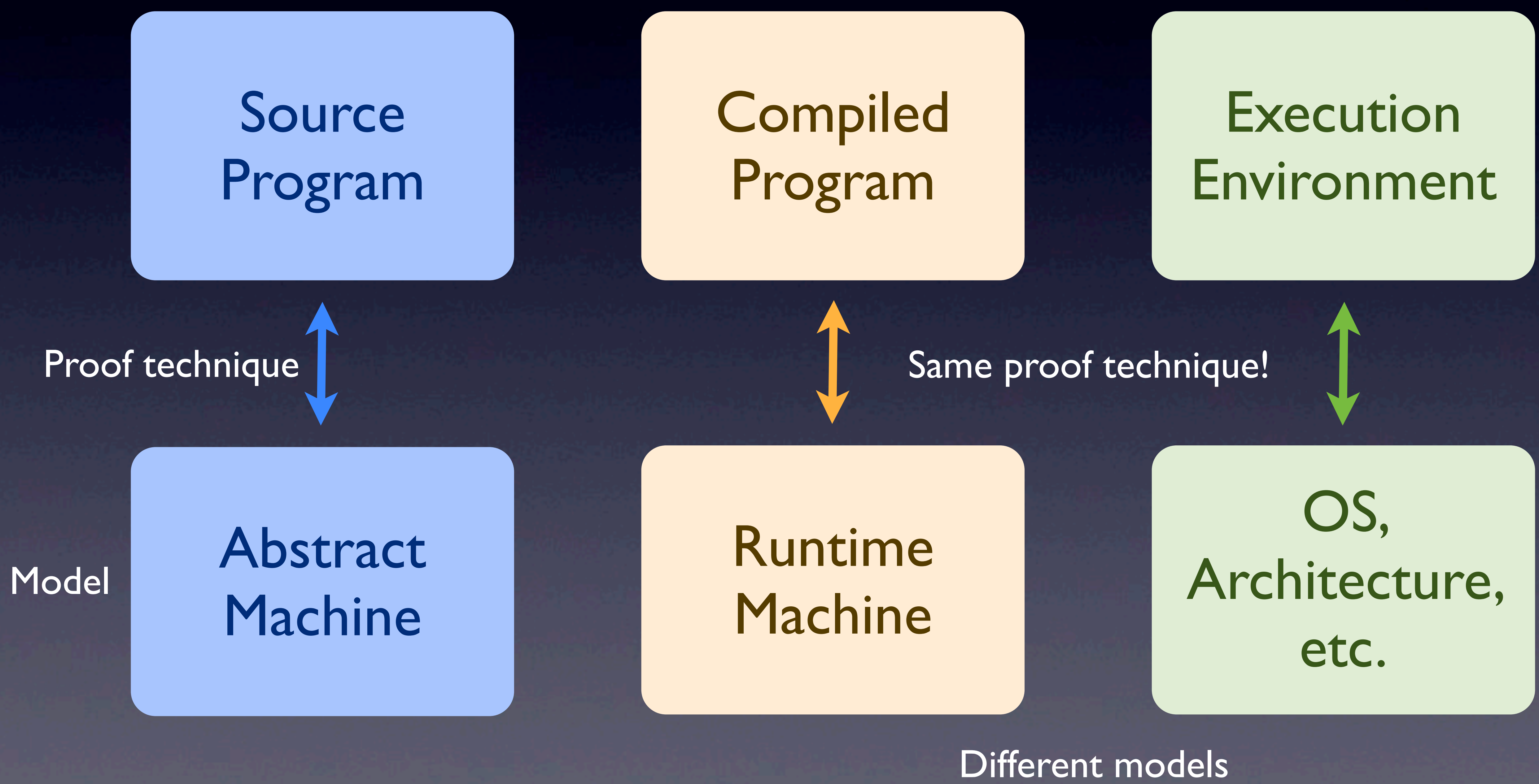
```
int increment(int a) {
  int b = a;
  b++;
  return b;
}
```

```
int increment(int a) {


  return a + 1;
}
```



Formal Model vs. Proof Technique



1	Compilers and Trust
2	Correctness vs. Security by Example
3	Correctness vs. Security, Formally
4	Future Directions

Testing for a Correctness-Security Gap

Memory Persistence	Dead store elimination Function call inlining Code motion
Side Channels	Subexpression Elimination Strength reduction Peephole Optimizations
Language Specifics	Undefinedness in C/C++ Memory model issues Synchronization issues

New, Formal Machine Models

Source Semantics Machine

IR Semantics Machine

Assembler Semantics Machine

Timing Machines

Memory Hierarchy Machines

Power Machines

Parameterized Correctness Proofs

Optimization

Machine

Attacker

Is the code before and after optimization, equivalent from the viewpoint of an attacker *observing the machine?*

Weak Memory Models	Security-Preserving Compilers
Litmus tests	?
Memory barriers	
Fence insertion	
New formal models	
Correctness modulo memory	
...	