Security Enhanced LLVM

Jeremy Bennett

Why?

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Why?

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Why?

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Why?

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

EMBECOSM®

Why?

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

EMBECOSM®

Why?

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

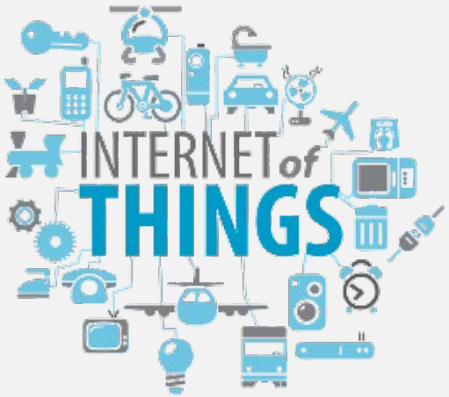
EMBECOSM®

Why?

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

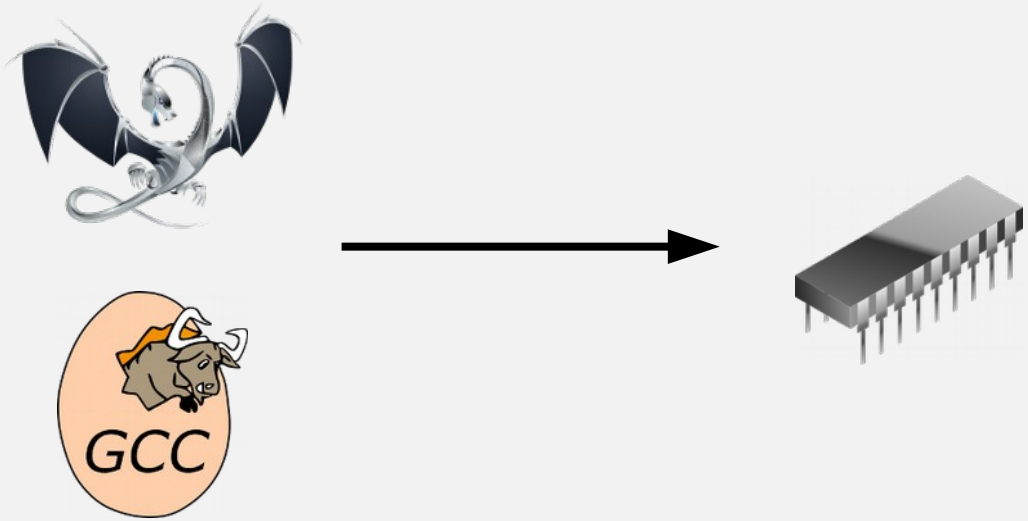
EMBECOSM®

Why?



Copyright © 2017 Embecosm. Freely available under a Creative Commons license

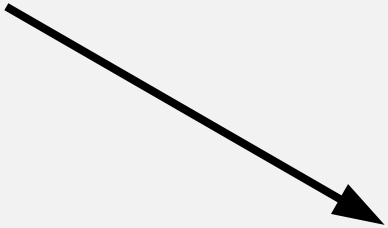
Why the Compiler?

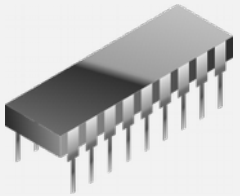


Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Why the Compiler?

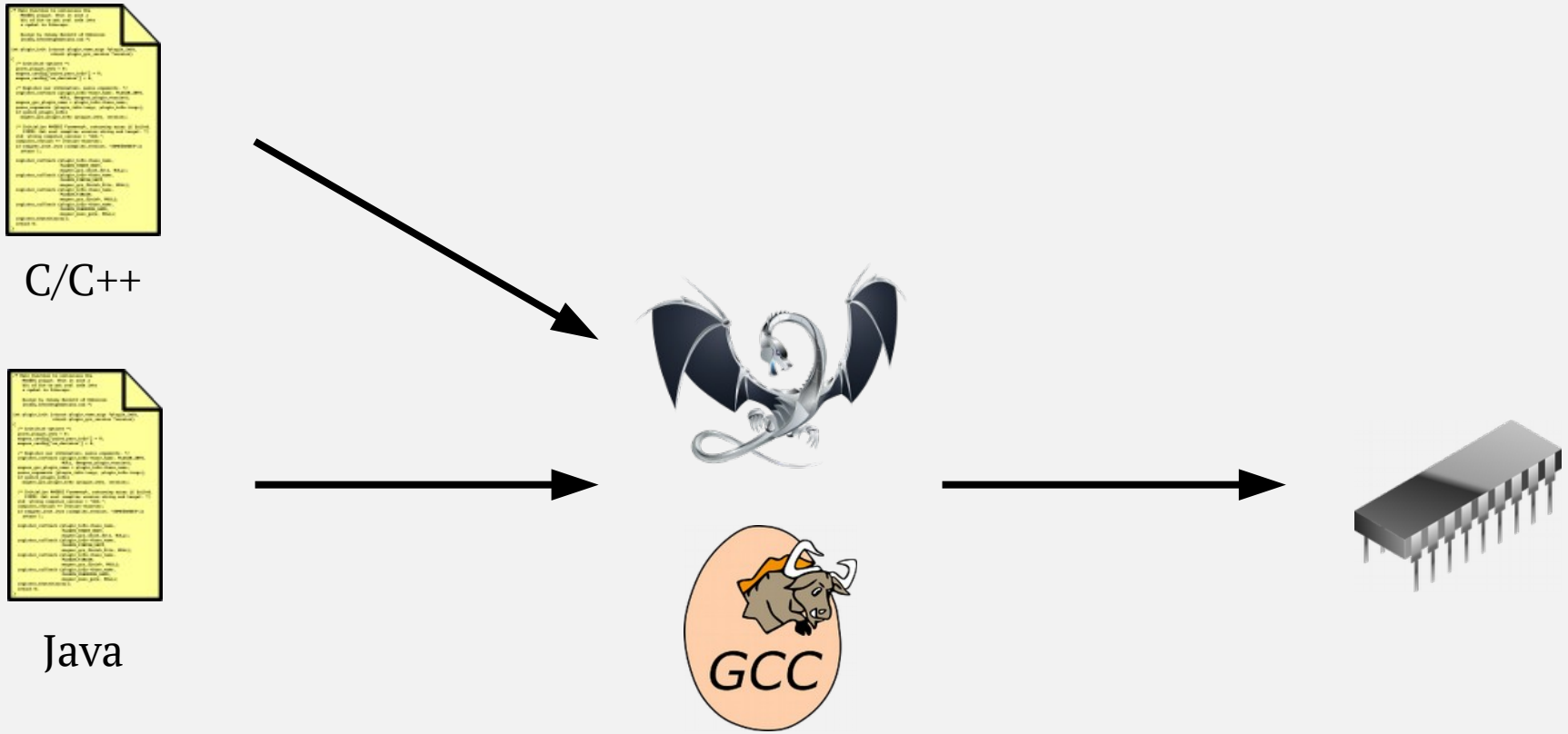
C/C++





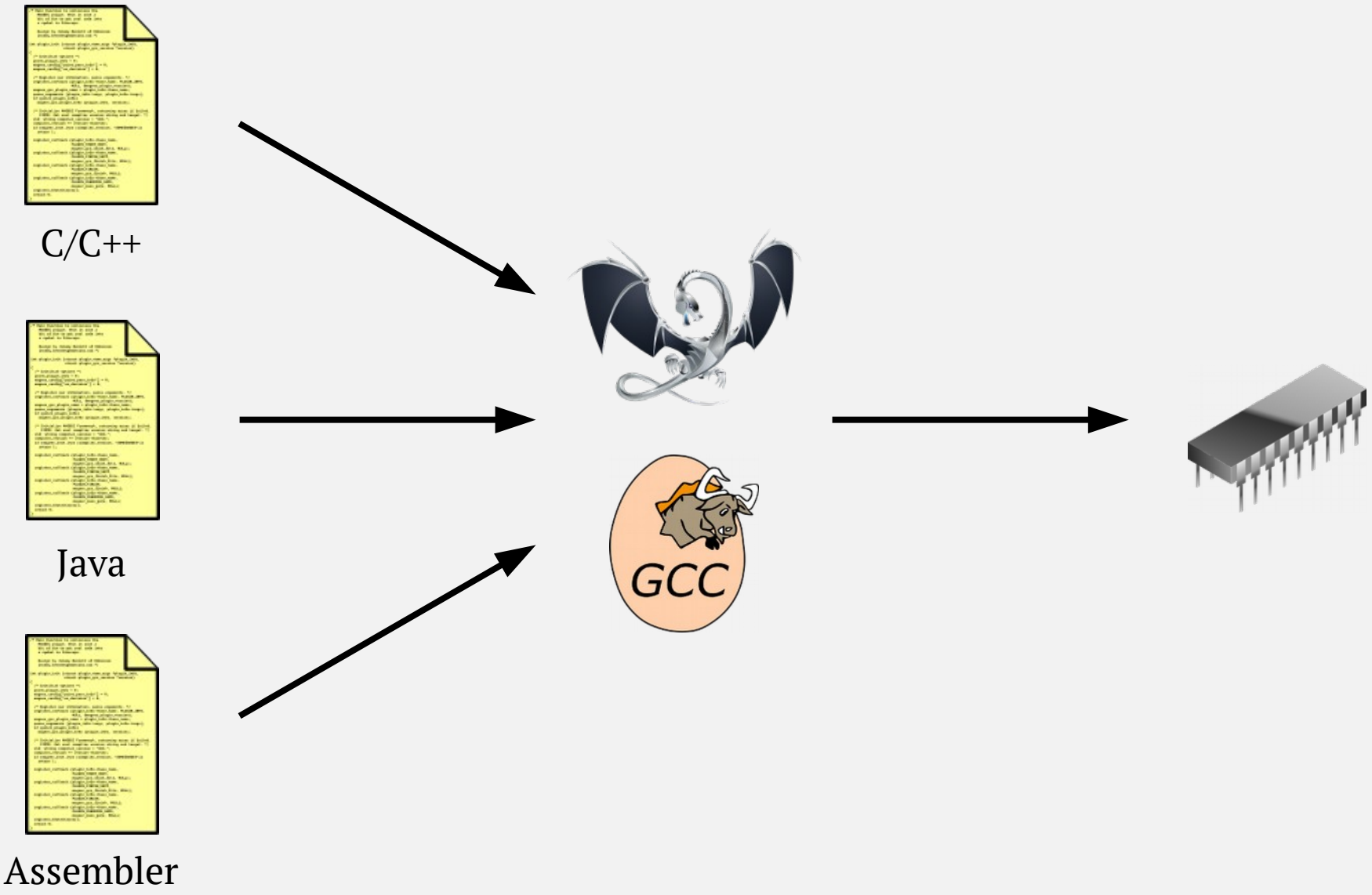
Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Why the Compiler?



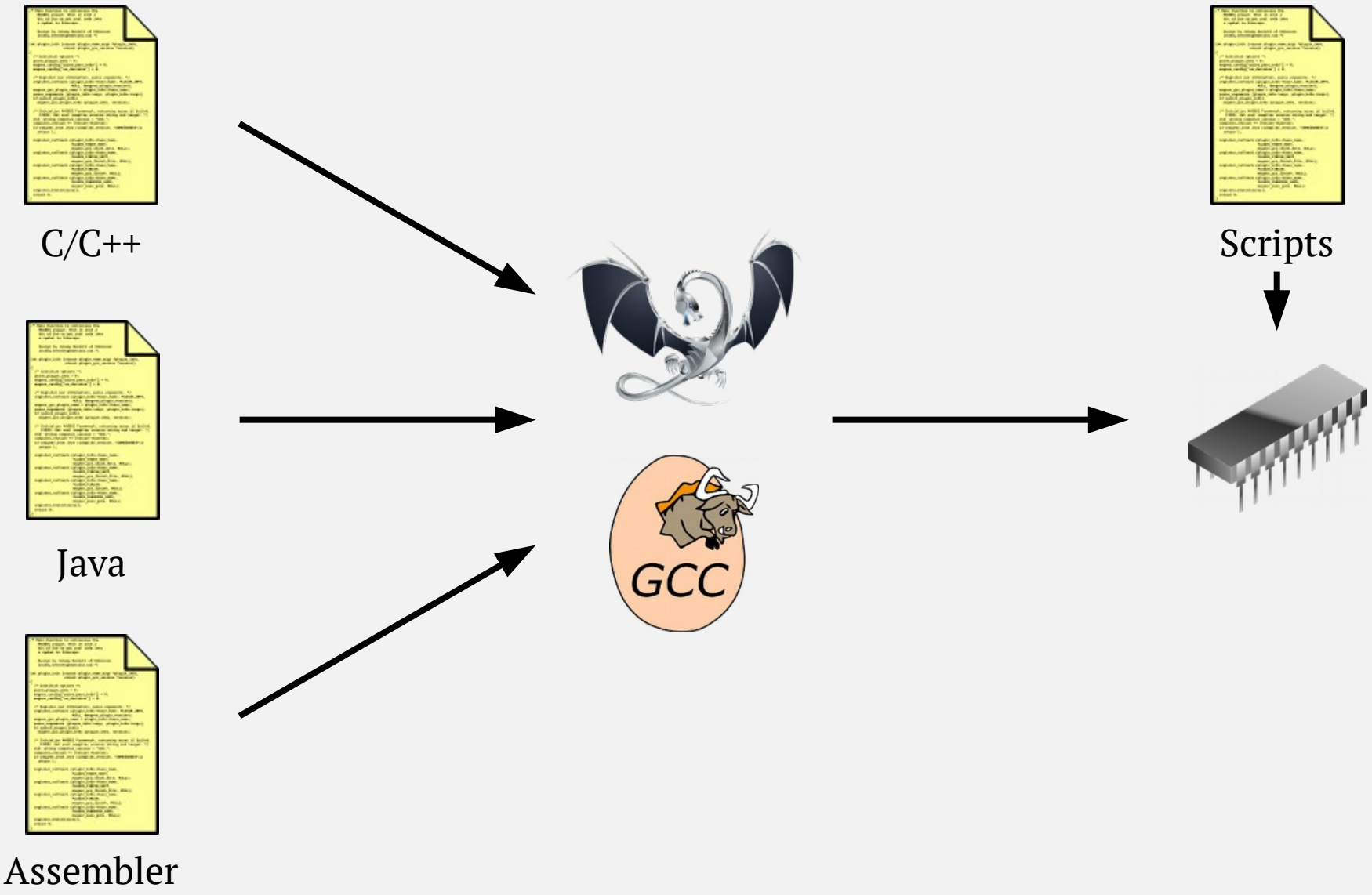
Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Why the Compiler?



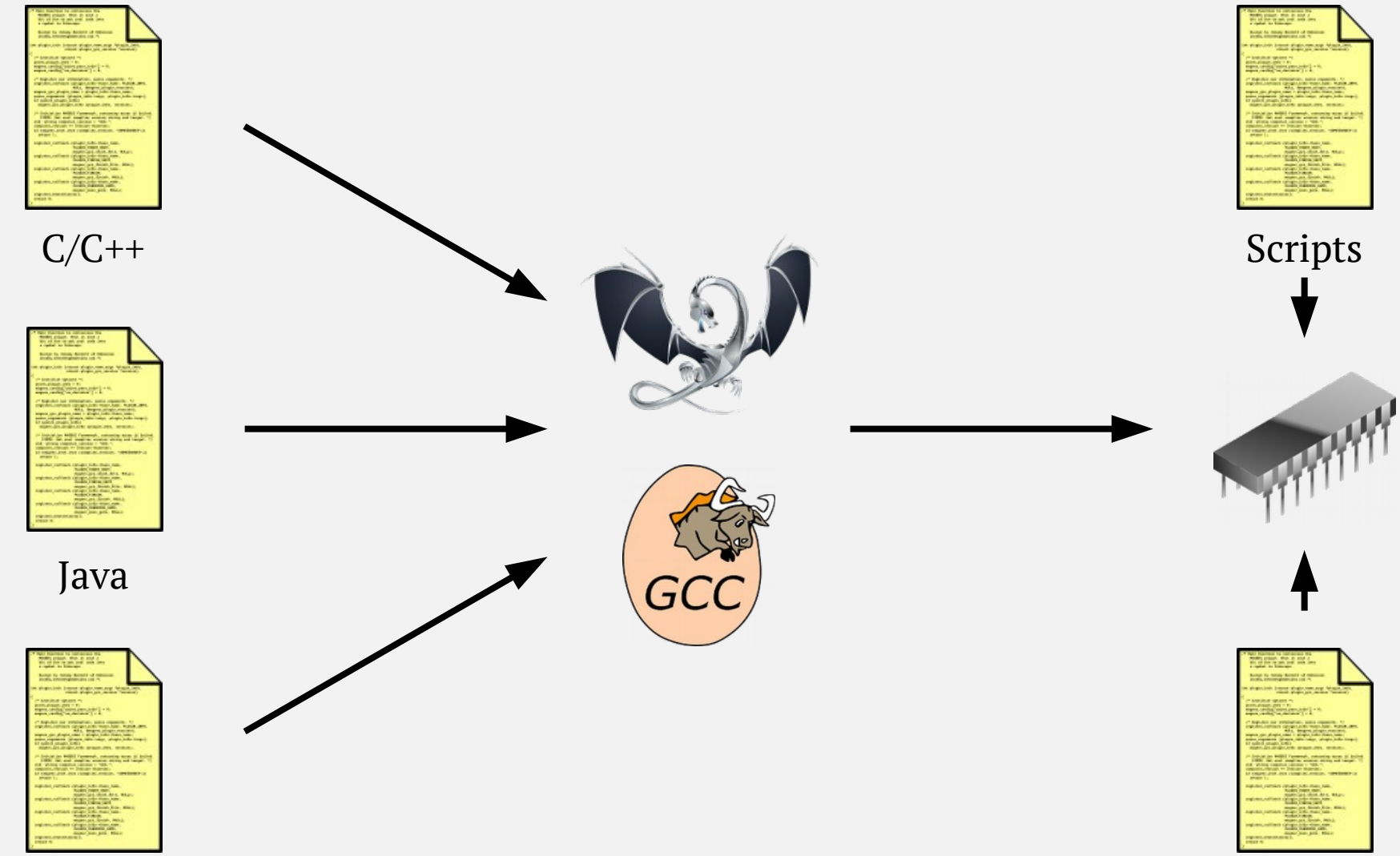
Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Why the Compiler?



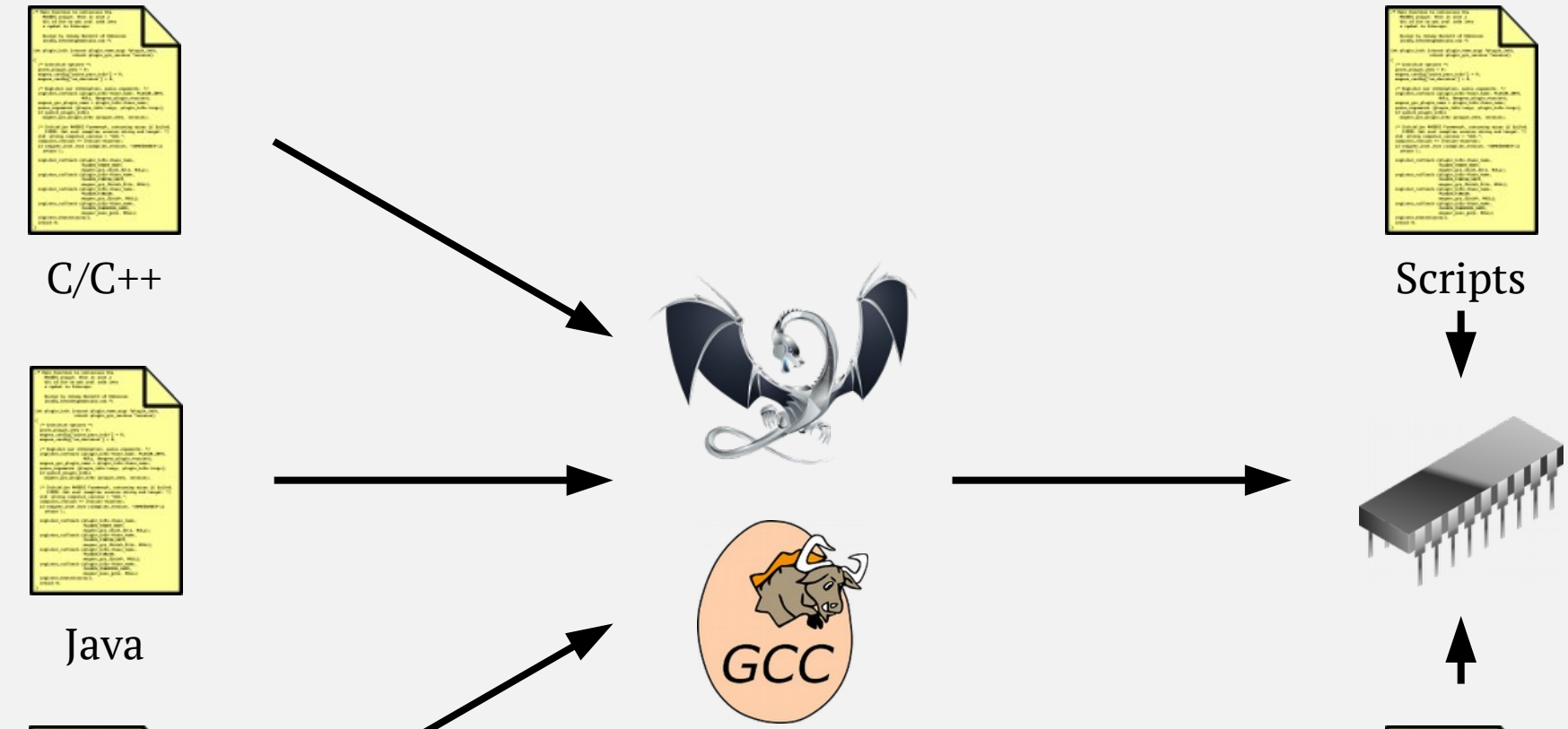
Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Why the Compiler?



Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Why the Compiler?



The compiler gets to look at (almost) all the code

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

How The Compiler Can Help

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Warning of bad practice

Advising the programmer when code appears to follow bad practice

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

How The Compiler Can Help

Warning of bad practice

Advising the programmer when code appears to follow bad practice

Providing heavy lifting

Automating complex tasks to make them easier for the programmer

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Using LLVM to guarantee program integrity
LLVM Dev Meeting, Nov 2016

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Previously...

Using LLVM to guarantee program integrity
LLVM Dev Meeting, Nov 2016

- Hardware adds support for:
  - instruction integrity
  - control flow integrity

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Previously...

Using LLVM to guarantee program integrity
LLVM Dev Meeting, Nov 2016

- Hardware adds support for:
  - instruction integrity
  - control flow integrity
- LLVM adds `__attribute__ ((protected))`

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Simple Things

```
int mangle (uint32_t k)
{
  uint32_t  res = 0;
  int i;

  for (i = 0; i < 8; i++)
   {
      uint32_t b = k >> (i * 4) & 0xf;
      res |= b << ((7 - i) * 4);
   }
  return res;
}

int main (int argc,
          char *argv[])
{
  uint32_t km;
  km = mangle (atoi (argv[1]));
  return (&km)[2];
}
```

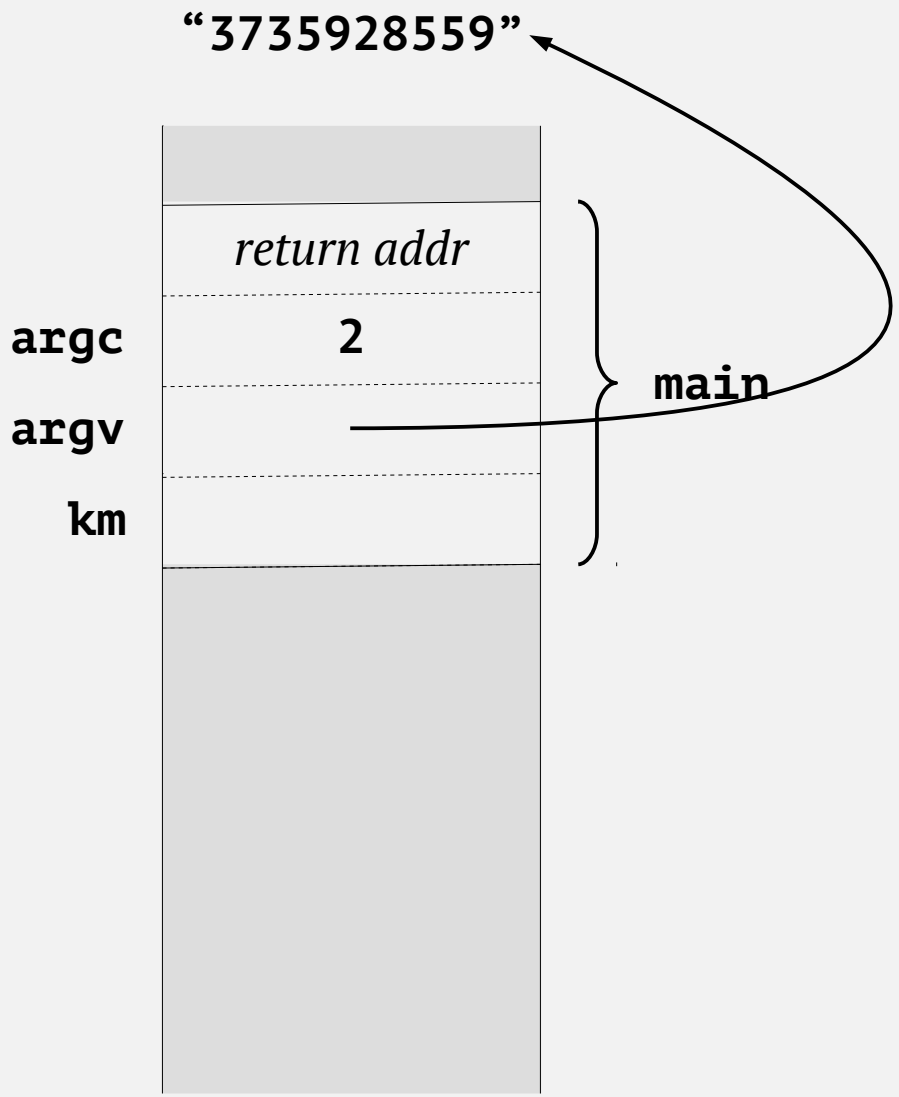
Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Simple Things

```
int mangle (uint32_t k)
{
  uint32_t  res = 0;
  int i;

  for (i = 0; i < 8; i++)
   {
      uint32_t b = k >> (i * 4) & 0xf;
      res |= b << ((7 - i) * 4);
   }
  return res;
}

int main (int argc,
          char *argv[])
{
  uint32_t km;
  km = mangle (atoi (argv[1]));
  return (&km)[2];
}
```



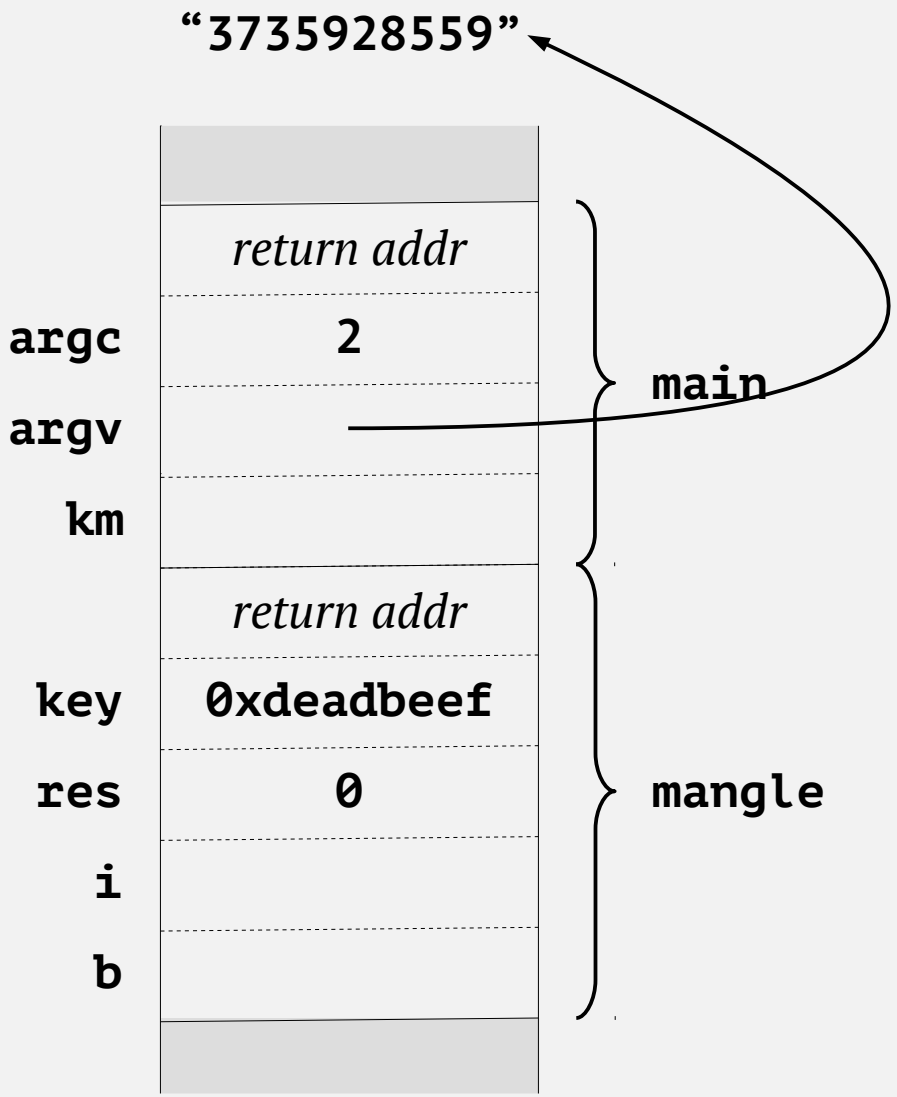
Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Simple Things

```
int mangle (uint32_t k)
{
  uint32_t  res = 0;
  int i;

  for (i = 0; i < 8; i++)
   {
      uint32_t b = k >> (i * 4) & 0xf;
      res |= b << ((7 - i) * 4);
   }
  return res;
}

int main (int argc,
          char *argv[])
{
  uint32_t km;
  km = mangle (atoi (argv[1]));
  return (&km)[2];
}
```



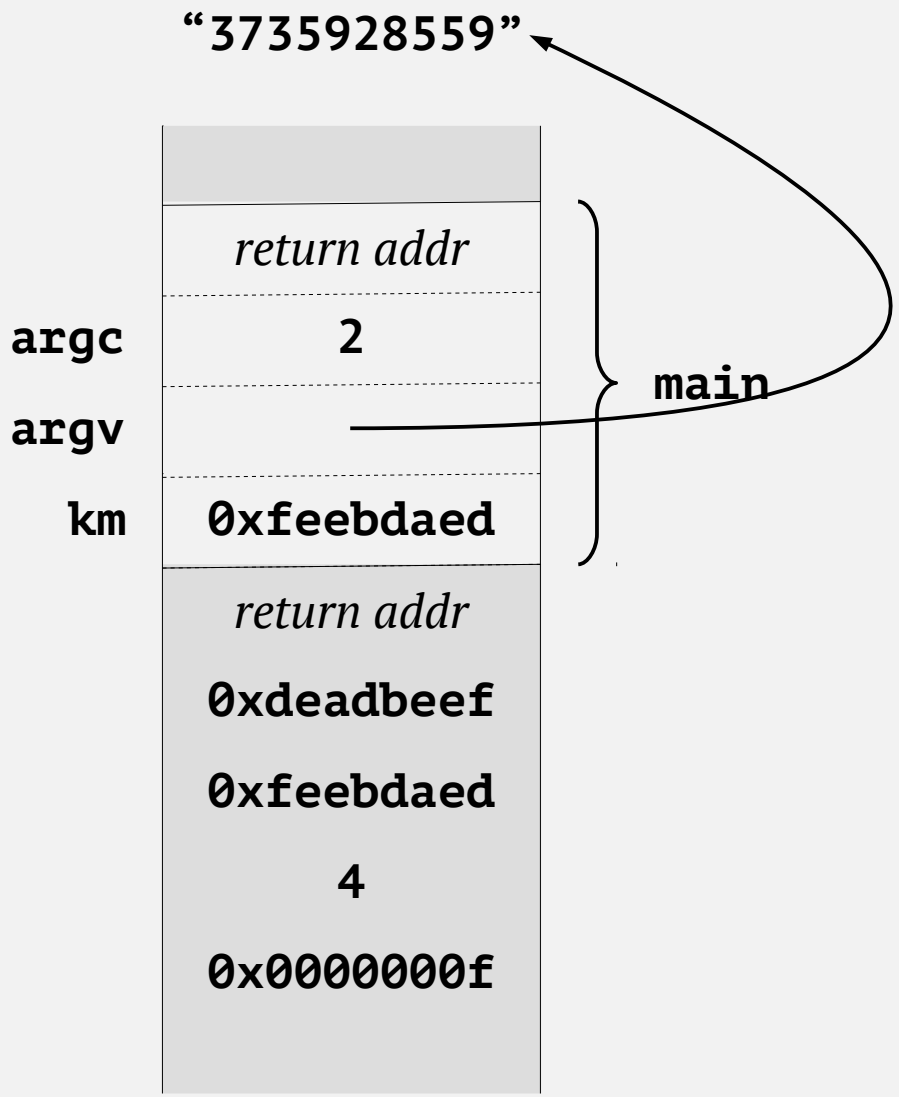
Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Simple Things

```
int mangle (uint32_t k)
{
  uint32_t  res = 0;
  int i;

  for (i = 0; i < 8; i++)
   {
      uint32_t b = k >> (i * 4) & 0xf;
      res |= b << ((7 - i) * 4);
   }
  return res;
}

int main (int argc,
          char *argv[])
{
  uint32_t km;
  km = mangle (atoi (argv[1]));
  return (&km)[2];
}
```



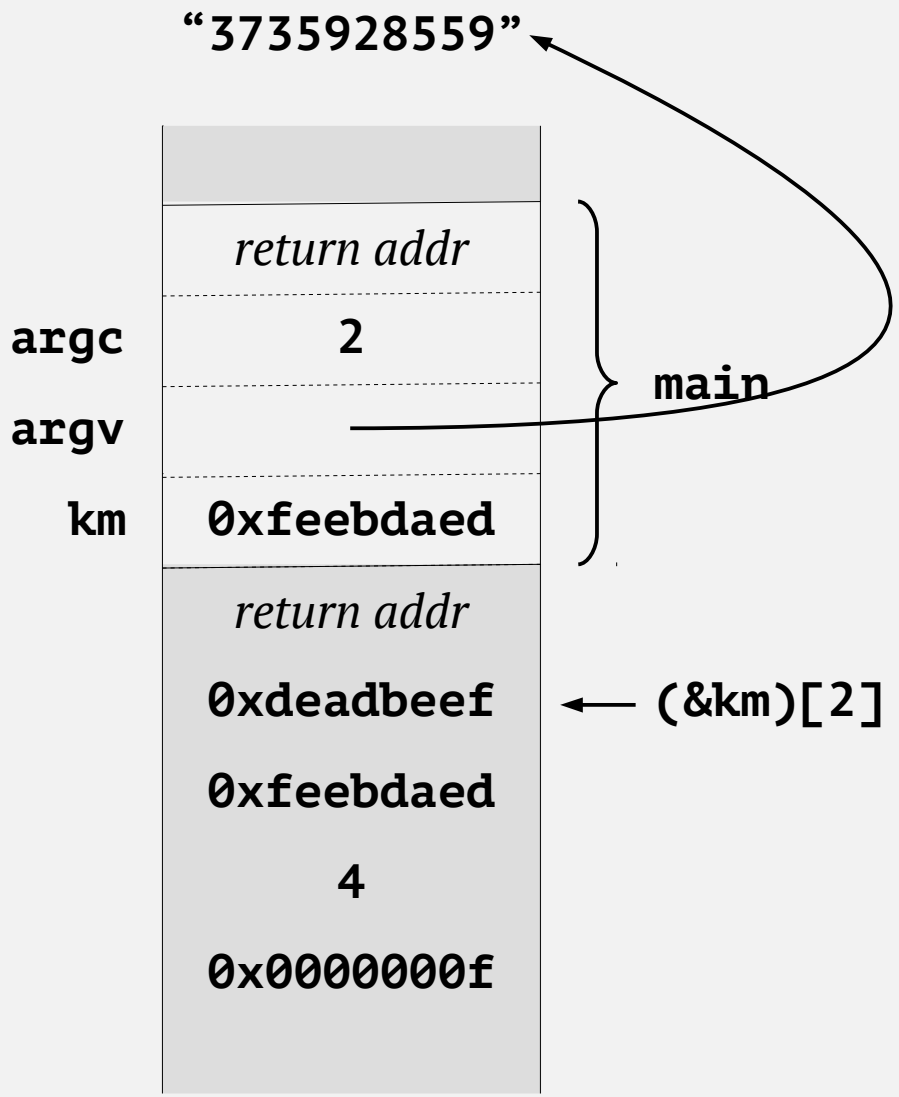
Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Simple Things

```
int mangle (uint32_t k)
{
  uint32_t  res = 0;
  int i;

  for (i = 0; i < 8; i++)
   {
      uint32_t b = k >> (i * 4) & 0xf;
      res |= b << ((7 - i) * 4);
   }
  return res;
}

int main (int argc,
          char *argv[])
{
  uint32_t km;
  km = mangle (atoi (argv[1]));
  return (&km)[2];
}
```



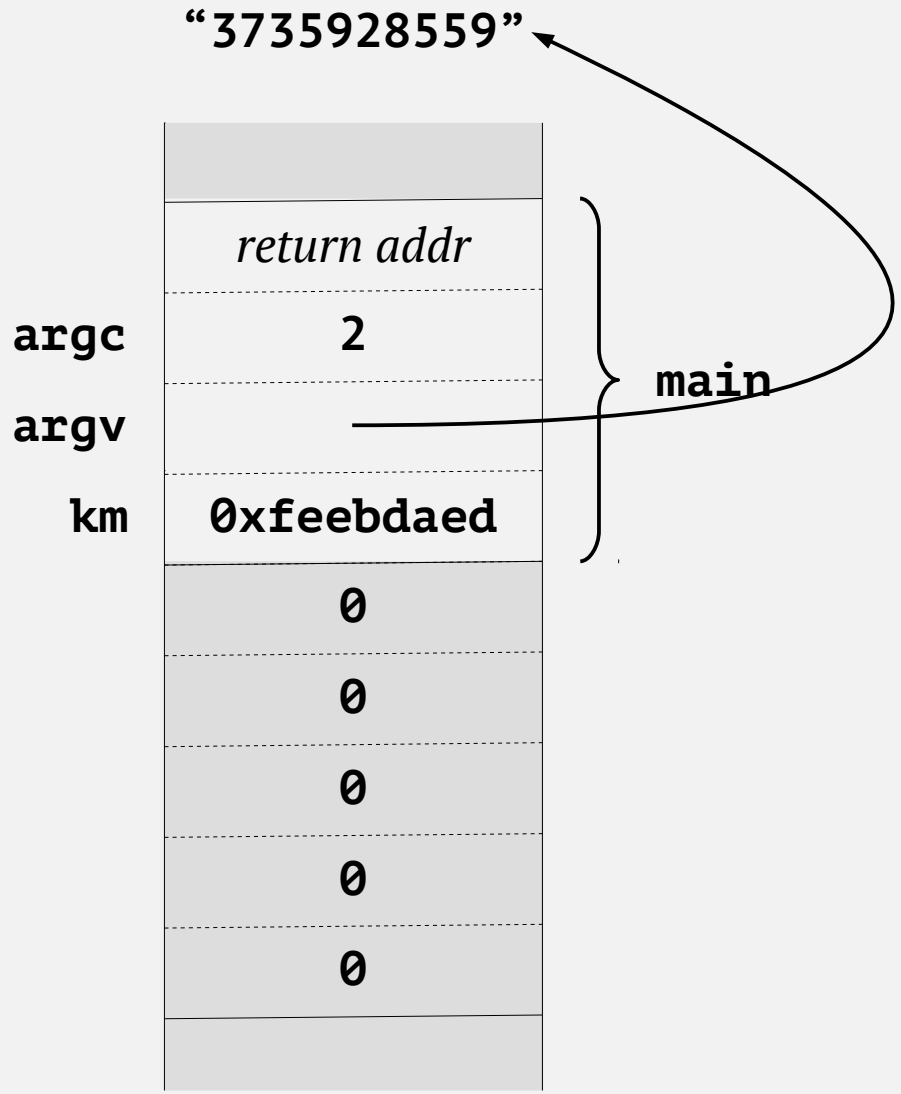
Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Simple Fix

```
int mangle (uint32_t k)
  __attribute__ ((erase_stack))
{
  uint32_t  res = 0;
  int i;

  for (i = 0; i < 8; i++)
   {
      uint32_t b = k >> (i * 4) & 0xf;
      res |= b << ((7 - i) * 4);
   }
  return res;
}

int main (int argc,
          char *argv[])
{
  uint32_t km;
  km = mangle (atoi (argv[1]));
  return (&km)[2];
}
```



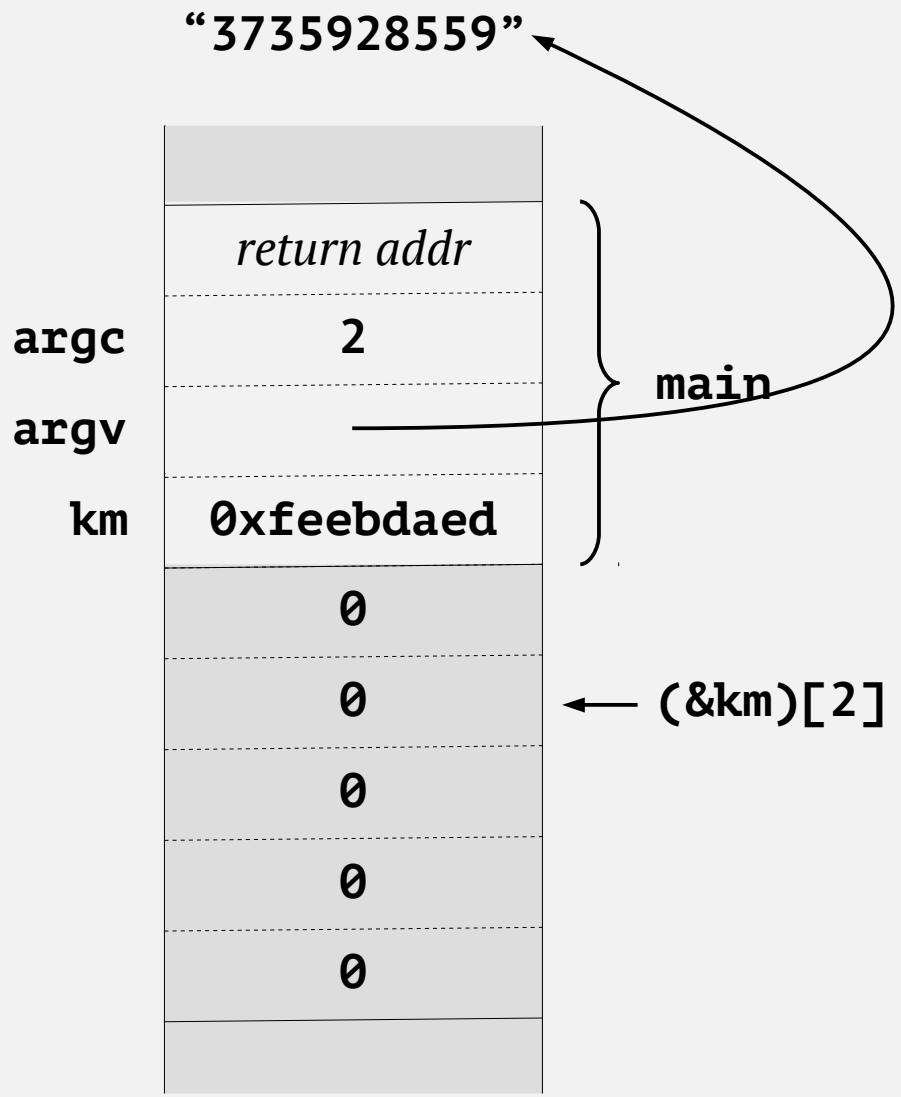
Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Simple Fix

```
int mangle (uint32_t k)
  __attribute__ ((erase_stack))
{
  uint32_t  res = 0;
  int i;

  for (i = 0; i < 8; i++)
   {
      uint32_t b = k >> (i * 4) & 0xf;
      res |= b << ((7 - i) * 4);
   }
  return res;
}

int main (int argc,
          char *argv[])
{
  uint32_t km;
  km = mangle (atoi (argv[1]));
  return (&km)[2];
}
```



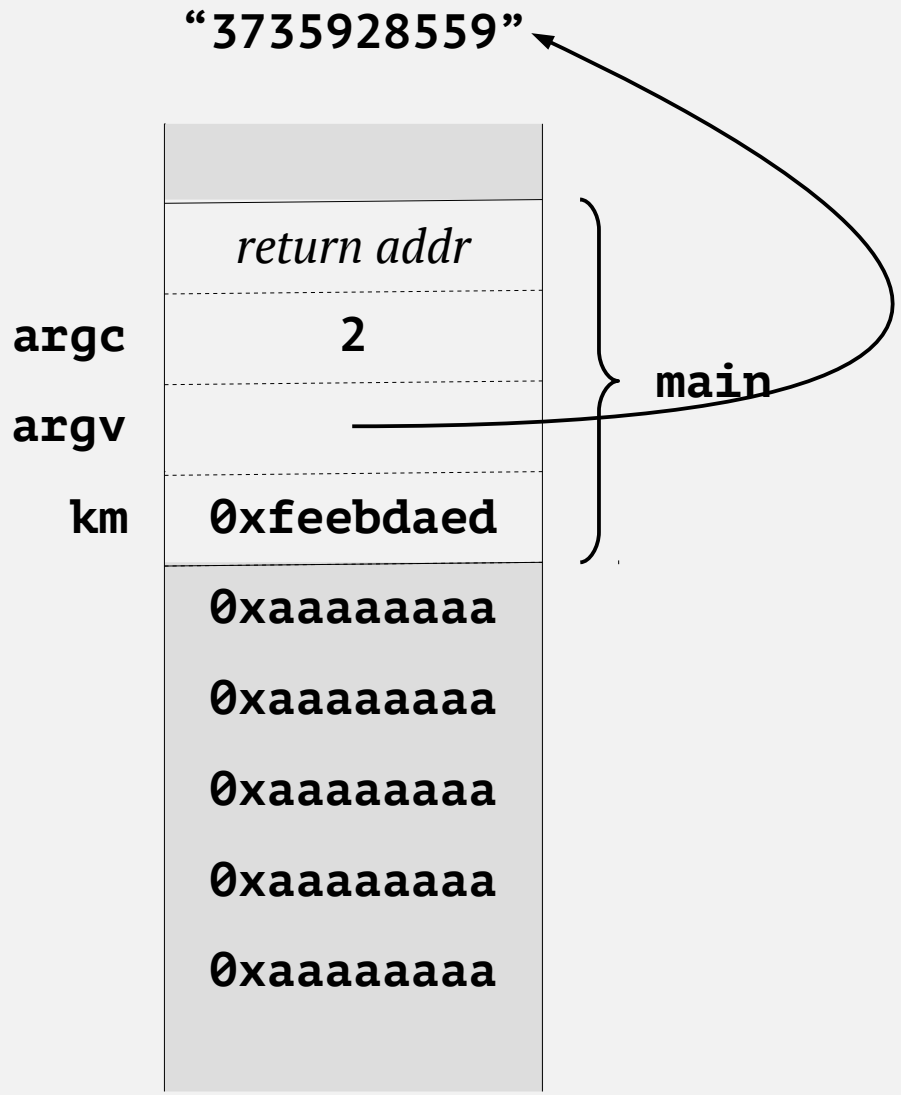
Copyright © 2017 Embecosm. Freely available under a Creative Commons license

A Better Simple Fix

```
int mangle (uint32_t k)
  __attribute__ ((erase_stack (0xaa)))
{
  uint32_t  res = 0;
  int i;

  for (i = 0; i < 8; i++)
   {
      uint32_t b = k >> (i * 4) & 0xf;
      res |= b << ((7 - i) * 4);
   }
  return res;
}

int main (int argc,
          char *argv[])
{
  uint32_t km;
  km = mangle (atoi (argv[1]));
  return (&km)[2];
}
```



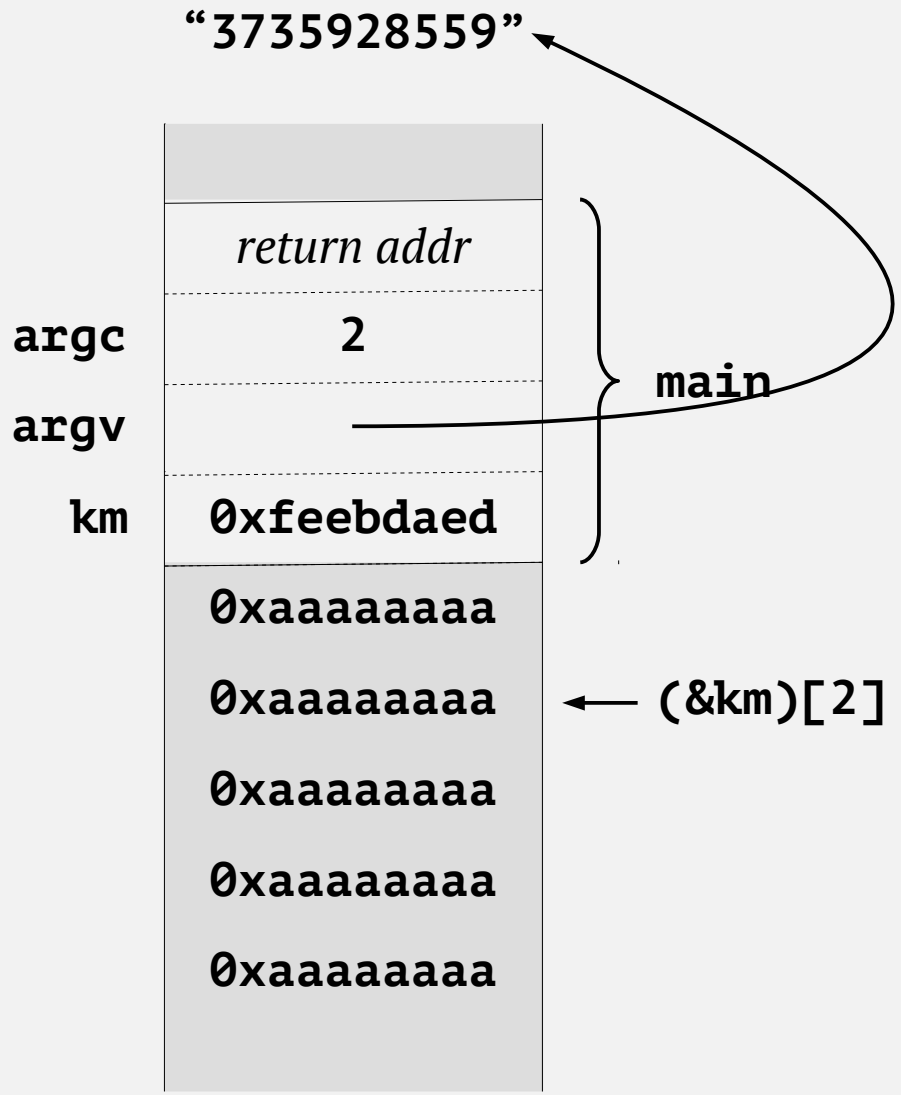
Copyright © 2017 Embecosm. Freely available under a Creative Commons license

A Better Simple Fix

```
int mangle (uint32_t k)
  __attribute__ ((erase_stack (0xaa)))
{
  uint32_t  res = 0;
  int i;

  for (i = 0; i < 8; i++)
   {
      uint32_t b = k >> (i * 4) & 0xf;
      res |= b << ((7 - i) * 4);
   }
  return res;
}

int main (int argc,
          char *argv[])
{
  uint32_t km;
  km = mangle (atoi (argv[1]));
  return (&km)[2];
}
```



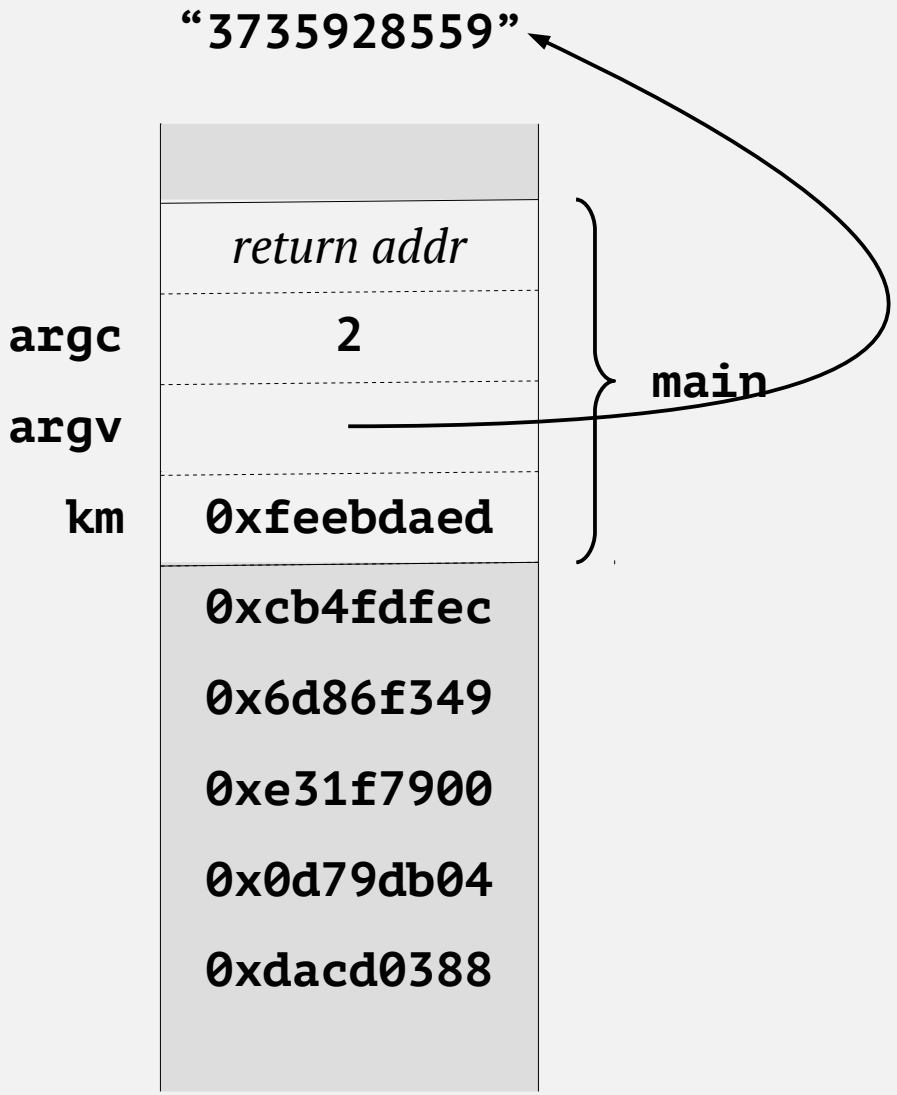
Copyright © 2017 Embecosm. Freely available under a Creative Commons license

The Best Simple Fix?

```
int mangle (uint32_t k)
  __attribute__ ((randomize_stack))
{
  uint32_t  res = 0;
  int i;

  for (i = 0; i < 8; i++)
   {
      uint32_t b = k >> (i * 4) & 0xf;
      res |= b << ((7 - i) * 4);
   }
  return res;
}

int main (int argc,
          char *argv[])
{
  uint32_t km;
  km = mangle (atoi (argv[1]));
  return (&km)[2];
}
```



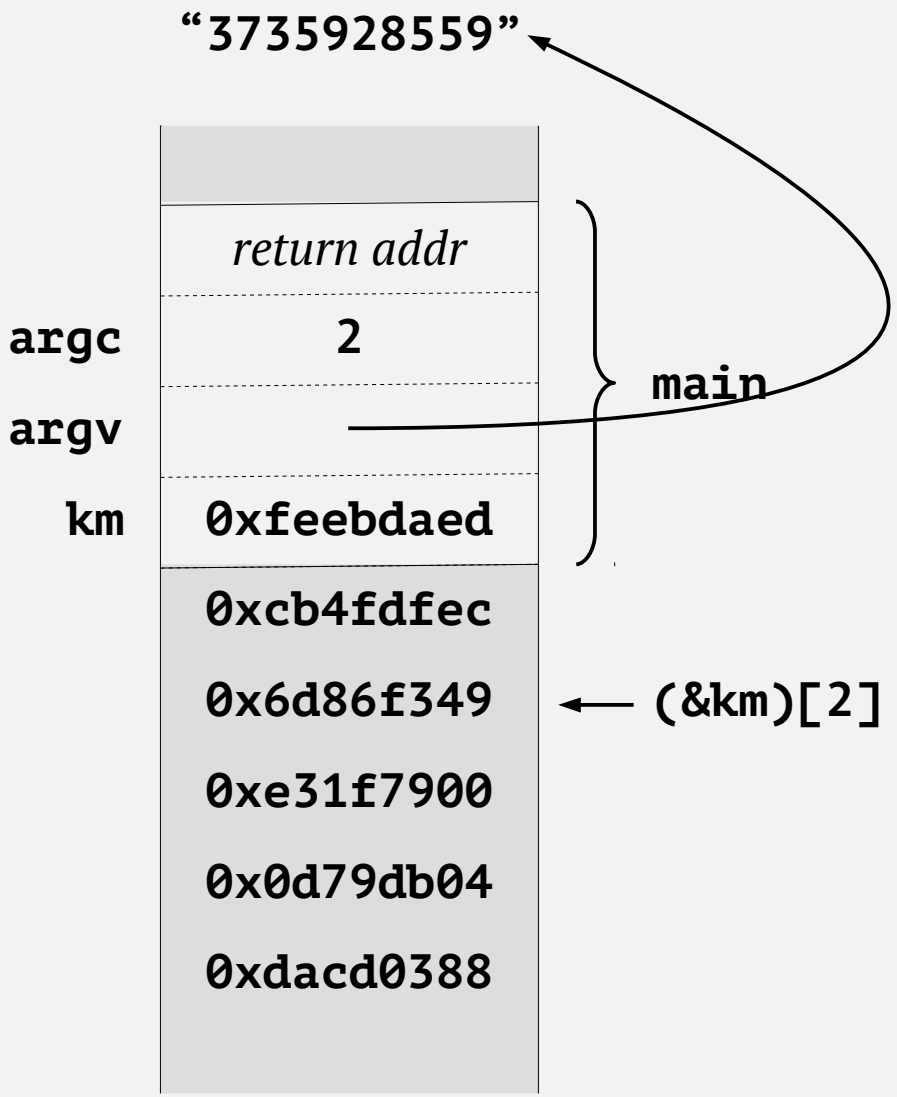
Copyright © 2017 Embecosm. Freely available under a Creative Commons license

The Best Simple Fix?

```
int mangle (uint32_t k)
  __attribute__ ((randomize_stack))
{
  uint32_t  res = 0;
  int i;

  for (i = 0; i < 8; i++)
   {
      uint32_t b = k >> (i * 4) & 0xf;
      res |= b << ((7 - i) * 4);
   }
  return res;
}

int main (int argc,
          char *argv[])
{
  uint32_t km;
  km = mangle (atoi (argv[1]));
  return (&km)[2];
}
```



Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Fixing setjmp/longjmp

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Fixing setjmp/longjmp

```
int top_func ()
{
  ...
  setjmp (buf)
  ...
}

int midfunc (uint32_t  k,
             jmp_buf *buf)
{
  ...
}

int farfunc (jmp_buf *buf)
{
  ...
  longjmp (*buf, 42);
}
```

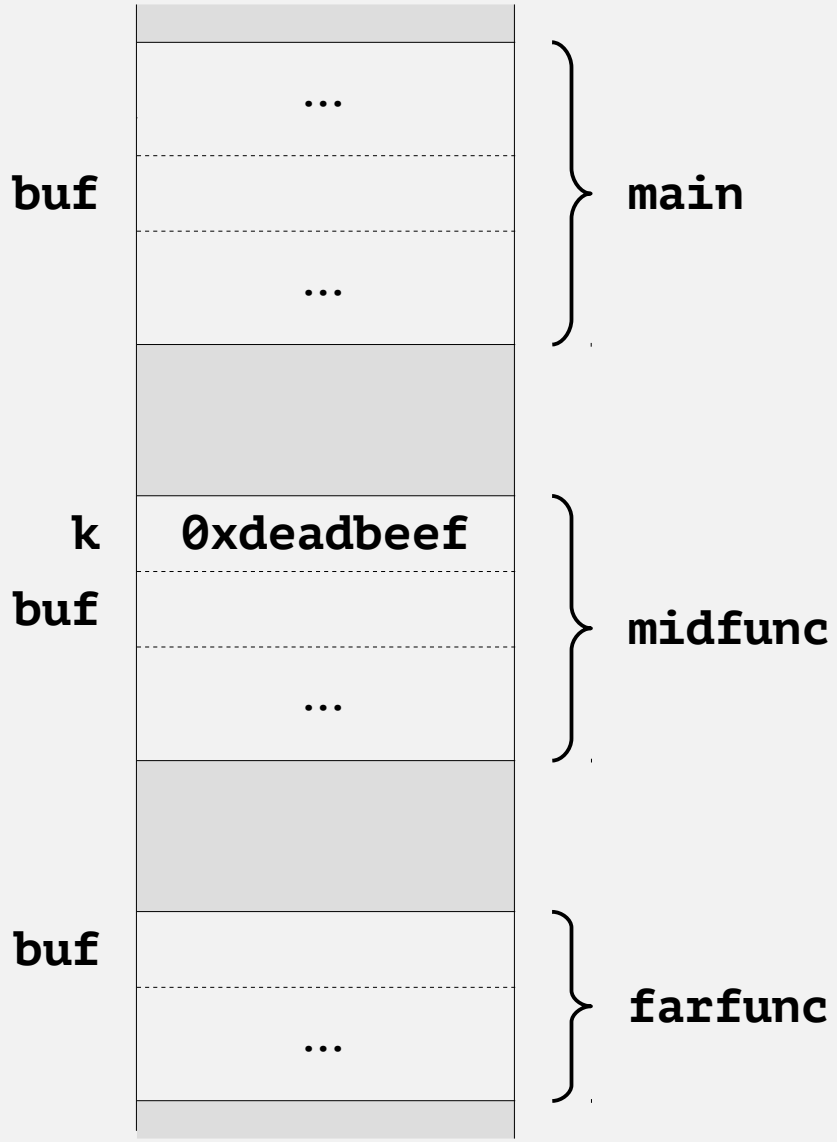
Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Fixing setjmp/longjmp

```
int top_func ()
{
   ...
   setjmp (buf)
   ...
}

int midfunc (uint32_t  k,
             jmp_buf *buf)
{
   ...
}

int farfunc (jmp_buf *buf)
{
   ...
   longjmp (*buf, 42);
}
```



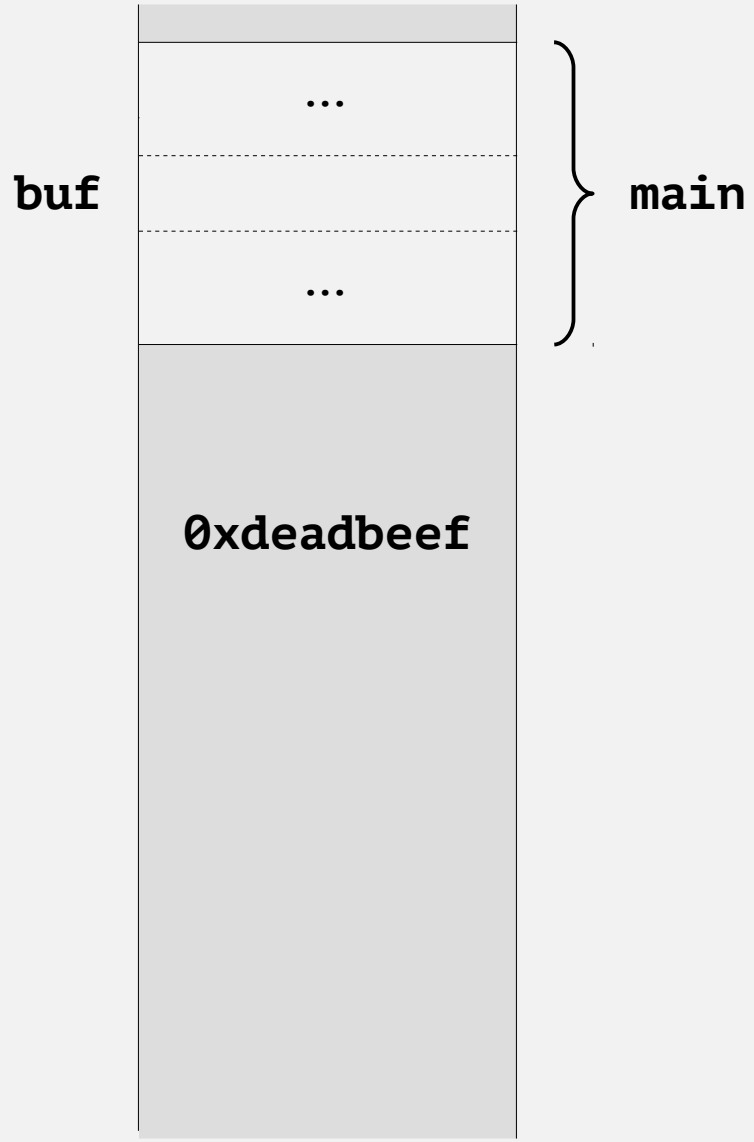
Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Fixing setjmp/longjmp

```
int top_func ()
{
  ...
  setjmp (buf)
  ...
}

int midfunc (uint32_t  k,
             jmp_buf *buf)
{
  ...
}

int farfunc (jmp_buf *buf)
{
  ...
  longjmp (*buf, 42);
}
```



Copyright © 2017 Embecosm. Freely available under a Creative Commons license

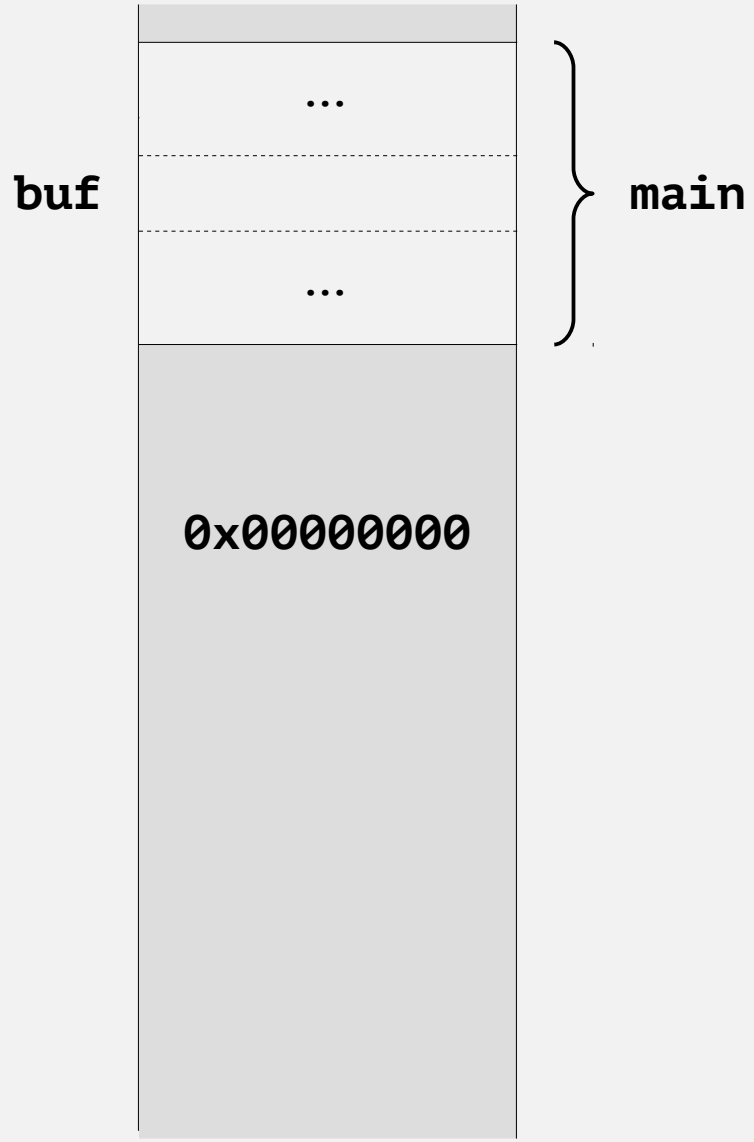
Fixing setjmp/longjmp

```
int top_func ()
{
  ...
  setjmp (buf)
  ...
}

int midfunc (uint32_t  k,
             jmp_buf *buf)
{
  ...
}

int farfunc (jmp_buf *buf)
{
  ...
  longjmp (*buf, 42);
}
```

clang -ferase-stack ...



Copyright © 2017 Embecosm. Freely available under a Creative Commons license

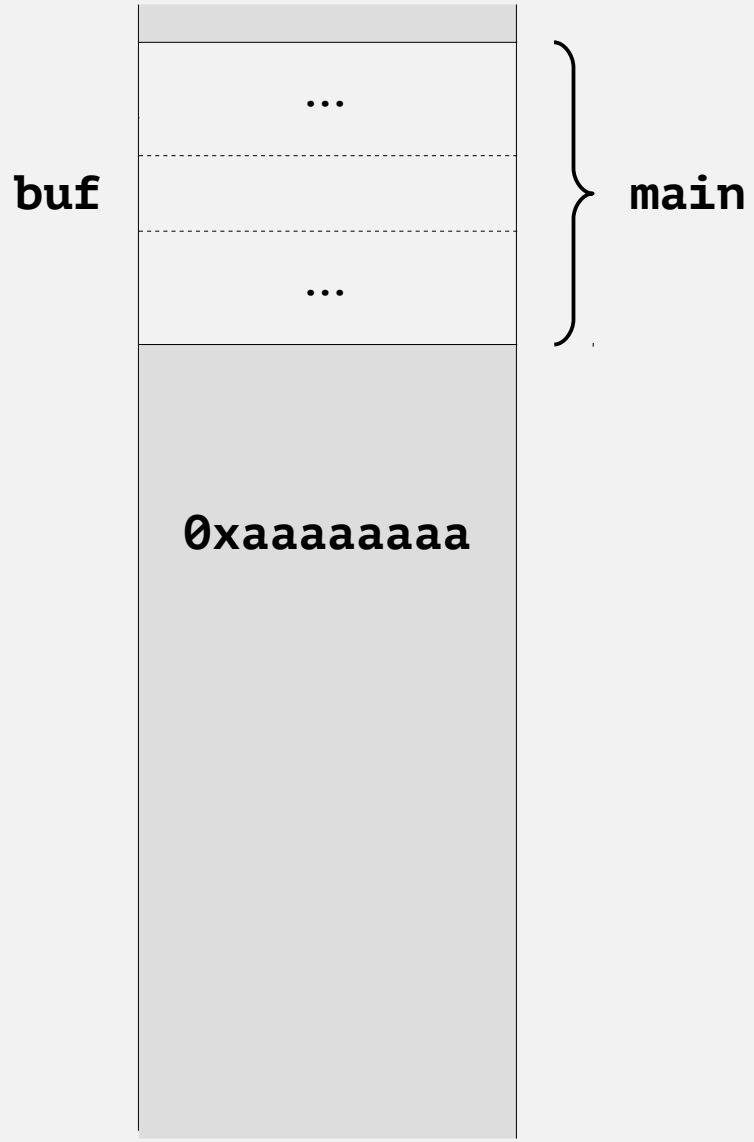
Fixing setjmp/longjmp

```
int top_func ()
{
  ...
  setjmp (buf)
  ...
}

int midfunc (uint32_t  k,
             jmp_buf *buf)
{
  ...
}

int farfunc (jmp_buf *buf)
{
  ...
  longjmp (*buf, 42);
}
```

clang -ferase-stack=0xaa ...



Copyright © 2017 Embecosm. Freely available under a Creative Commons license

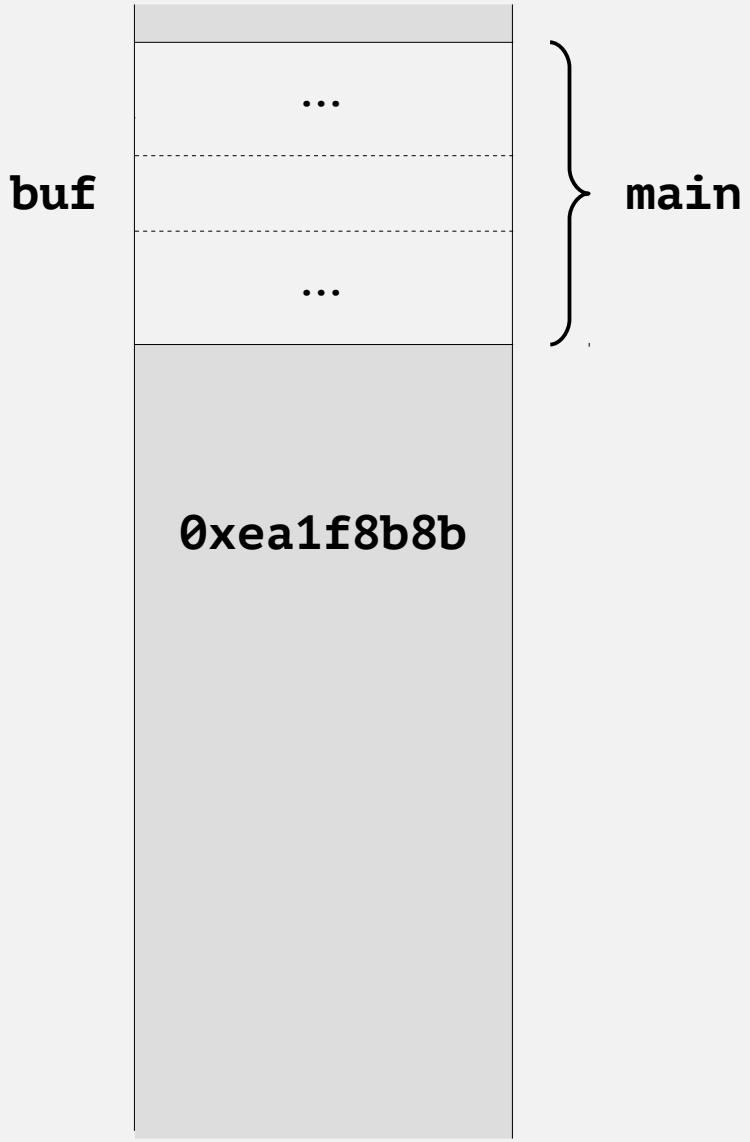
Fixing setjmp/longjmp

```
int top_func ()
{
  ...
  setjmp (buf)
  ...
}

int midfunc (uint32_t  k,
             jmp_buf *buf)
{
  ...
}

int farfunc (jmp_buf *buf)
{
  ...
  longjmp (*buf, 42);
}
```

clang -frandomize-stack ...



Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Summary of one Simple Problem

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Summary of one Simple Problem

- Clean up the stack for an individual function
  - `__attribute__ ((erase_stack))`

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Summary of one Simple Problem

- Clean up the stack for an individual function
  - `__attribute__ ((erase_stack))`
  - `__attribute__ ((erase_stack (`*n*`)))`

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Summary of one Simple Problem

- Clean up the stack for an individual function
  - `__attribute__ ((erase_stack))`
  - `__attribute__ ((erase_stack (`*n*`)))`
  - `__attribute__ ((randomize_stack))`

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Summary of one Simple Problem

- Clean up the stack for an individual function
  - `__attribute__ ((erase_stack))`
  - `__attribute__ ((erase_stack (`*n*`)))`
  - `__attribute__ ((randomize_stack))`
- Always clean up the stack, including longjmp

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Summary of one Simple Problem

- Clean up the stack for an individual function
  - `__attribute__ ((erase_stack))`
  - `__attribute__ ((erase_stack (`*n*`)))`
  - `__attribute__ ((randomize_stack))`
- Always clean up the stack, including longjmp
  - clang -ferase-stack …

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Summary of one Simple Problem

- Clean up the stack for an individual function
  - `__attribute__ ((erase_stack))`
  - `__attribute__ ((erase_stack (`*n*`)))`
  - `__attribute__ ((randomize_stack))`
- Always clean up the stack, including longjmp
  - clang -ferase-stack …
  - clang -ferase-stack=*n* …

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Summary of one Simple Problem

- Clean up the stack for an individual function
  - `__attribute__ ((erase_stack))`
  - `__attribute__ ((erase_stack (`*n*`)))`
  - `__attribute__ ((randomize_stack))`
- Always clean up the stack, including longjmp
  - clang -ferase-stack …
  - clang -ferase-stack=*n* …
  - clang -frandomize-stack …

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

A Harder Problem

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

A Harder Problem

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int globvar;

int main (int argc,
          char *argv[])
{
  uint32_t k, km;

  k = atoi (argv[1]);
  globvar = 1729;
  km = (k * k) % globvar;

  if (globvar != 1729)
    printf ("Globvar wrong\n");

  return km;
}
```

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

A Harder Problem
Unoptimized Code

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int globvar;

int main (int argc,
          char *argv[])
{
  uint32_t k, km;

  k = atoi (argv[1]);
  globvar = 1729;
  km = (k * k) % globvar;

  if (globvar != 1729)
    printf ("Globvar wrong\n");

  return km;
}
```

```
...
        callq   atoi
        movl    %eax, -20(%rbp)
        movl    $1729, globvar
        movl    -20(%rbp), %eax
        imull   -20(%rbp), %eax
        xorl    %edx, %edx
        divl    globvar
        movl    %edx, -24(%rbp)
        cmpl    $1729, globvar
        je      .LBB0_2
        movabsq $.L.str, %rdi
        movb    $0, %al
        callq   printf
        movl    %eax, -28(%rbp)
.LBB0_2:
        movl    -24(%rbp), %eax
        addq    $32, %rsp
        popq    %rbp
        retq
```

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

A Harder Problem
Optimized Code

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int globvar;

int main (int argc,
          char *argv[])
{
  uint32_t k, km;

  k = atoi (argv[1]);
  globvar = 1729;
  km = (k * k) % globvar;

  if (globvar != 1729)
    printf ("Globvar wrong\n");

  return km;
}
```

```
...
callq   strtol
movl    $1729, globvar(%rip)
imull   %eax, %eax
imulq   $792420225, %rax, %rcx
shrq    $32, %rcx
movl    %eax, %edx
subl    %ecx, %edx
shrl    %edx
addl    %ecx, %edx
shrl    $10, %edx
imull   $1729, %edx, %ecx
subl    %ecx, %eax
popq    %rcx
retq
```

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

A Harder Problem
Optimized Code Fixed - Sort Of

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

volatile int globvar;

int main (int argc,
          char *argv[])
{
  uint32_t k, km;

  k = atoi (argv[1]);
  globvar = 1729;
  km = (k * k) % globvar;

  if (globvar != 1729)
    printf ("Globvar wrong\n");

  return km;
}
```

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

A Harder Problem
Optimized Code Fixed - Sort Of

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

volatile int globvar;

int main (int argc,
          char *argv[])
{
  uint32_t k, km;

  k = atoi (argv[1]);
  globvar = 1729;
  km = (k * k) % globvar;

  if (globvar != 1729)
    printf ("Globvar wrong\n");

  return km;
}
```

```
...
callq   strtol
movl    $1729, globvar(%rip)
imull   %eax, %eax
xorl    %edx, %edx
divl    globvar(%rip)
movl    %edx, %ebx
movl    globvar(%rip), %eax
cmpl    $1729, %eax
je      .LBB0_2
movl    $.Lstr, %edi
callq   puts
.LBB0_2:
movl    %ebx, %eax
popq    %rbx
retq
```

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

A Harder Problem
Optimized Code Fixed - Sort Of

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

volatile int globvar;

int main (int argc,
          char *argv[])
{
  uint32_t k, km;

  k = atoi (argv[1]);
  globvar = 1729;
  km = (k * k) % globvar;

  if (globvar != 1729)
    printf ("Globvar wrong\n");

  return km;
}
```

```
...
callq   strtol
movl    $1729, globvar(%rip)
imull   %eax, %eax
xorl    %edx, %edx
divl    globvar(%rip)
movl    %edx, %ebx
movl    globvar(%rip), %eax
cmpl    $1729, %eax
je      .LBB0_2
movl    $.Lstr, %edi
callq   puts
.LBB0_2:
movl    %ebx, %eax
popq    %rbx
retq
```

How do we disable optimization for just one block?

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Leakage Aware Design Automation
The LADA Project

Prof Elisabeth Oswald
University of Bristol

Dr Dan Page
University of Bristol

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

EMBECOSM®

Leakage Aware Design Automation
The LADA Project

Prof Elisabeth Oswald
University of Bristol

Dr Dan Page
University of Bristol

- EPSRC funded 4 year academic research project
  - supported by a team of RAs and PhD students

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Leakage Aware Design Automation
The LADA Project

Prof Elisabeth Oswald
University of Bristol

Dr Dan Page
University of Bristol

- EPSRC funded 4 year academic research project
  - supported by a team of RAs and PhD students
- Embecosm is the "industrial supporter"
  - providing summer PhD internships
  - writing open source implementations for GCC & LLVM

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

What is Information Leakage

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

What is Information Leakage

"Information leakage happens whenever a system that is designed to be closed to an eavesdropper reveals some information to unauthorized parties nonetheless."

Wikipedia

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

What is Information Leakage

"Information leakage happens whenever a system that is designed to be closed to an eavesdropper reveals some information to unauthorized parties nonetheless."

Wikipedia

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Differential Power Analysis

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Differential Power Analysis

```
int func (uint32_t  k)
{
  int  i, res = 0;

  for (i = 0; i < 10000000; i++)
    if (1 == (k & 1))
      res += k - 1;
    else
      {
        double r;
        r = sqrt ((double) k);
        res += (int) r;
      }
  return res;
}

int main (int argc,
          char *argv[])
{
  return func (atoi (argv[1]));
}
```

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Differential Power Analysis

```
int func (uint32_t  k)
{
  int  i, res = 0;

  for (i = 0; i < 10000000; i++)
    if (1 == (k & 1))
      res += k - 1;
    else
      {
        double r;
        r = sqrt ((double) k);
        res += (int) r;
      }
  return res;
}

int main (int argc,
          char *argv[])
{
  return func (atoi (argv[1]));
}
```

```
$ time ./dpa 7

real    0m0.025s
user    0m0.024s
sys     0m0.000s
```

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Differential Power Analysis

```
int func (uint32_t  k)
{
  int  i, res = 0;

  for (i = 0; i < 10000000; i++)
    if (1 == (k & 1))
      res += k - 1;
    else
      {
        double r;
        r = sqrt ((double) k);
        res += (int) r;
      }
  return res;
}

int main (int argc,
          char *argv[])
{
  return func (atoi (argv[1]));
}
```

```
$ time ./dpa 7

real    0m0.025s
user    0m0.024s
sys     0m0.000s

$ time ./dpa 6

real    0m0.086s
user    0m0.084s
sys     0m0.000s
```

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Differential Power Analysis

```
int func (uint32_t  k)
{
  int  i, res = 0;

  for (i = 0; i < 10000000; i++)
    if (1 == (k & 1))
      res += k - 1;
    else
      {
        double r;
        r = sqrt ((double) k);
        res += (int) r;
      }
  return res;
}

int main (int argc,
          char *argv[])
{
  return func (atoi (argv[1]));
}
```

```
$ time ./dpa 7

real    0m0.025s
user    0m0.024s
sys     0m0.000s

$ time ./dpa 6

real    0m0.086s
user    0m0.084s
sys     0m0.000s
```

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Differential Power Analysis

```
int func (uint32_t  k)
{
  int  i, res = 0;

  for (i = 0; i < 10000000; i++)
    if (1 == (k & 1))
      res += k - 1;
    else
      {
        double r;
        r = sqrt ((double) k);
        res += (int) r;
      }
  return res;
}

int main (int argc,
          char *argv[])
{
  return func (atoi (argv[1]));
}
```

```
$ time ./dpa 7

real    0m0.025s
user    0m0.024s
sys     0m0.000s

$ time ./dpa 6

real    0m0.086s
user    0m0.084s
sys     0m0.000s
```

We need to worry about
- data dependent control flow
- data dependent instruction timing
- data dependent memory access

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Warning about Critical Variables Simple Cases (1)

```
int func (uint32_t  k)

{
  int  i, res = 0;

  if (1 == (k & 1))
    res += k - 1;
  else
    {
      double r;
      r = sqrt ((double) k);
      res += (int) r;
    }
  return res;
}
```

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Warning about Critical Variables Simple Cases (1)

```
int func (uint32_t  k
  __attribute__ ((critvar)))
{
  int  i, res = 0;

  if (1 == (k & 1))
    res += k - 1;
  else
    {
      double r;
      r = sqrt ((double) k);
      res += (int) r;
    }
  return res;
}
```

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Warning about Critical Variables Simple Cases (1)

```
int func (uint32_t  k
  __attribute__ ((critvar)))
{
  int  i, res = 0;

  if (1 == (k & 1))
    res += k - 1;
  else
    {
      double r;
      r = sqrt ((double) k);
      res += (int) r;
    }
  return res;
}
```

```
$ clang -c gv.c
gv.c:6:12: warning: critical
variable controlling flow 'k' [-
Wcritical-variable]
  if (1 == (k & 1))
            ^
```

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Warning about Critical Variables Simple Cases (2)

myinc.h:

```
extern uint32_t k
  __attribute__ ((critvar));
```

gv.c:

```
#include "myinc.h"
int func ()
{
  int  i, res = 0;


  if (1 == (k & 1))
    res += k - 1;
  else
    {
      double r;
      r = sqrt ((double) k);
      res += (int) r;
    }
  return res;
}
```

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Warning about Critical Variables Simple Cases (2)

myinc.h:

```
extern uint32_t k
  __attribute__ ((critvar));
```

gv.c:

```
#include "myinc.h"
int func ()
{
  int  i, res = 0;


  if (1 == (k & 1))
    res += k - 1;
  else
    {
      double r;
      r = sqrt ((double) k);
      res += (int) r;
    }
  return res;
}
```

```
$ clang -c gv.c
gv.c:6:12: warning: critical
variable controlling flow 'k' [-
Wcritical-variable]
  if (1 == (k & 1))
            ^
```

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Warning about Critical Variables Simple Cases (2)

myinc.h:

```
extern uint32_t k
  __attribute__ ((critvar));
```

gv.c:

```
#include "myinc.h"
int func ()
{
  int  i, res = 0;


  if (1 == (k & 1))
    res += k - 1;
  else
    {
      double r;
      r = sqrt ((double) k);
      res += (int) r;
    }
  return res;
}
```

```
$ clang -c gv.c
gv.c:6:12: warning: critical
variable controlling flow 'k' [-
Wcritical-variable]
  if (1 == (k & 1))
            ^
```

Critical variable usage is obvious

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Warning about Critical Variables
Not Quite So Simple Case

```
int func (uint32_t  k
  __attribute__ ((critvar)))
{
  int  i, res = 0;
  uint8_t b = k & 0xff;

  if (1 == (b & 1))
    res += k - 1;
  else
    {
      double r;
      r = sqrt ((double) k);
      res += (int) r;
    }
  return res;
}
```

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Warning about Critical Variables
Not Quite So Simple Case

```
int func (uint32_t  k
  __attribute__ ((critvar)))
{
  int  i, res = 0;
  uint8_t b = k & 0xff;

  if (1 == (b & 1))
    res += k - 1;
  else
    {
      double r;
      r = sqrt ((double) k);
      res += (int) r;
    }
  return res;
}
```

```
$ clang -c gv.c
gv.c:6:12: warning: critical
variable controlling flow 'b' [-
Wcritical-variable]
  if (1 == (b & 1))
            ^
```

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Warning about Critical Variables
Not Quite So Simple Case

```
int func (uint32_t  k
  __attribute__ ((critvar)))
{
  int  i, res = 0;
  uint8_t b = k & 0xff;

  if (1 == (b & 1))
    res += k - 1;
  else
    {
      double r;
      r = sqrt ((double) k);
      res += (int) r;
    }
  return res;
}
```

```
$ clang -c gv.c
gv.c:6:12: warning: critical
variable controlling flow 'b' [-
Wcritical-variable]
  if (1 == (b & 1))
            ^
```

Critical variable usage needs to understand local data flow

- can we always get this right?

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Warning about Critical Variables Harder Case

```
gv1.c:
int func1 (uint32_t  k
  __attribute__ ((critvar)))
{
  return func2 (k);
}

gv2.c:
int func2 (uint32_t  x)
{
  int  i, res = 0;

  if (1 == (x & 1))
    res += x - 1;
  else
    {
      double r;
      r = sqrt ((double) x);
      res += (int) r;
    }
  return res;
}
```

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Warning about Critical Variables Harder Case

gv1.c:

```
int func1 (uint32_t  k
  __attribute__ ((critvar)))
{
  return func2 (k);
}
```

gv2.c:

```
int func2 (uint32_t  x)
{
  int  i, res = 0;

  if (1 == (x & 1))
    res += x - 1;
  else
    {
      double r;
      r = sqrt ((double) x);
      res += (int) r;
    }
  return res;
}
```

```
$ clang -c gv1.c
$ clang -c gv2.c
$ clang gv1.o gv2.o
$
```

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Warning about Critical Variables Harder Case

```
gv1.c:
int func1 (uint32_t  k
  __attribute__ ((critvar)))
{
  return func2 (k);
}

gv2.c:
int func2 (uint32_t  x)
{
  int  i, res = 0;

  if (1 == (x & 1))
    res += x - 1;
  else
    {
      double r;
      r = sqrt ((double) x);
      res += (int) r;
    }
  return res;
}
```

```
$ clang -c gv1.c
$ clang -c gv2.c
$ clang gv1.o gv2.o
$
```

No warning! Individually neither function can see a critical variable is controlling flow.

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Warning about Critical Variables Harder Case

```
gv1.c:
int func1 (uint32_t  k
  __attribute__ ((critvar)))
{
  return func2 (k);
}

gv2.c:
int func2 (uint32_t  x)
{
  int  i, res = 0;

  if (1 == (x & 1))
    res += x - 1;
  else
    {
      double r;
      r = sqrt ((double) x);
      res += (int) r;
    }
  return res;
}
```

```
$ clang -flto -c gv1.c
$ clang -flto -c gv2.c
$ clang -flto gv1.o gv2.o
gv2.c:5:12: warning: critical
variable controlling flow 'x' [-
Wcritical-variable]
  if (1 == (x & 1))
            ^
```

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Warning about Critical Variables Harder Case

```
gv1.c:
int func1 (uint32_t  k
  __attribute__ ((critvar)))
{
  return func2 (k);
}

gv2.c:
int func2 (uint32_t  x)
{
  int  i, res = 0;

  if (1 == (x & 1))
    res += x - 1;
  else
    {
      double r;
      r = sqrt ((double) x);
      res += (int) r;
    }
  return res;
}
```

```
$ clang -flto -c gv1.c
$ clang -flto -c gv2.c
$ clang -flto gv1.o gv2.o
gv2.c:5:12: warning: critical
variable controlling flow 'x' [-
Wcritical-variable]
  if (1 == (x & 1))
            ^
```

Critical variable usage needs to understand *global* data flow

- can we always get this right?

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Warning about Critical Variables Summary

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Warning about Critical Variables Summary

- Simple cases are easy

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Warning about Critical Variables Summary

- Simple cases are easy
- Most cases need dataflow analysis
  - LTO for programs of any size

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Warning about Critical Variables Summary

- Simple cases are easy
- Most cases need dataflow analysis
  - LTO for programs of any size
- What if we get it wrong
  - false positives mean lots of fruitless debugging
  - false negatives mean “bad” code gets through

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Warning about Critical Variables Summary

- Simple cases are easy
- Most cases need dataflow analysis
  - LTO for programs of any size
- What if we get it wrong
  - false positives mean lots of fruitless debugging
  - false negatives mean “bad” code gets through
- It is not just control flow that leaks
  - what about variation in memory access?
  - what about variation in instruction timing?
  - what about energy?

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

8-Bit Processor Multiply Instruction Heat Map

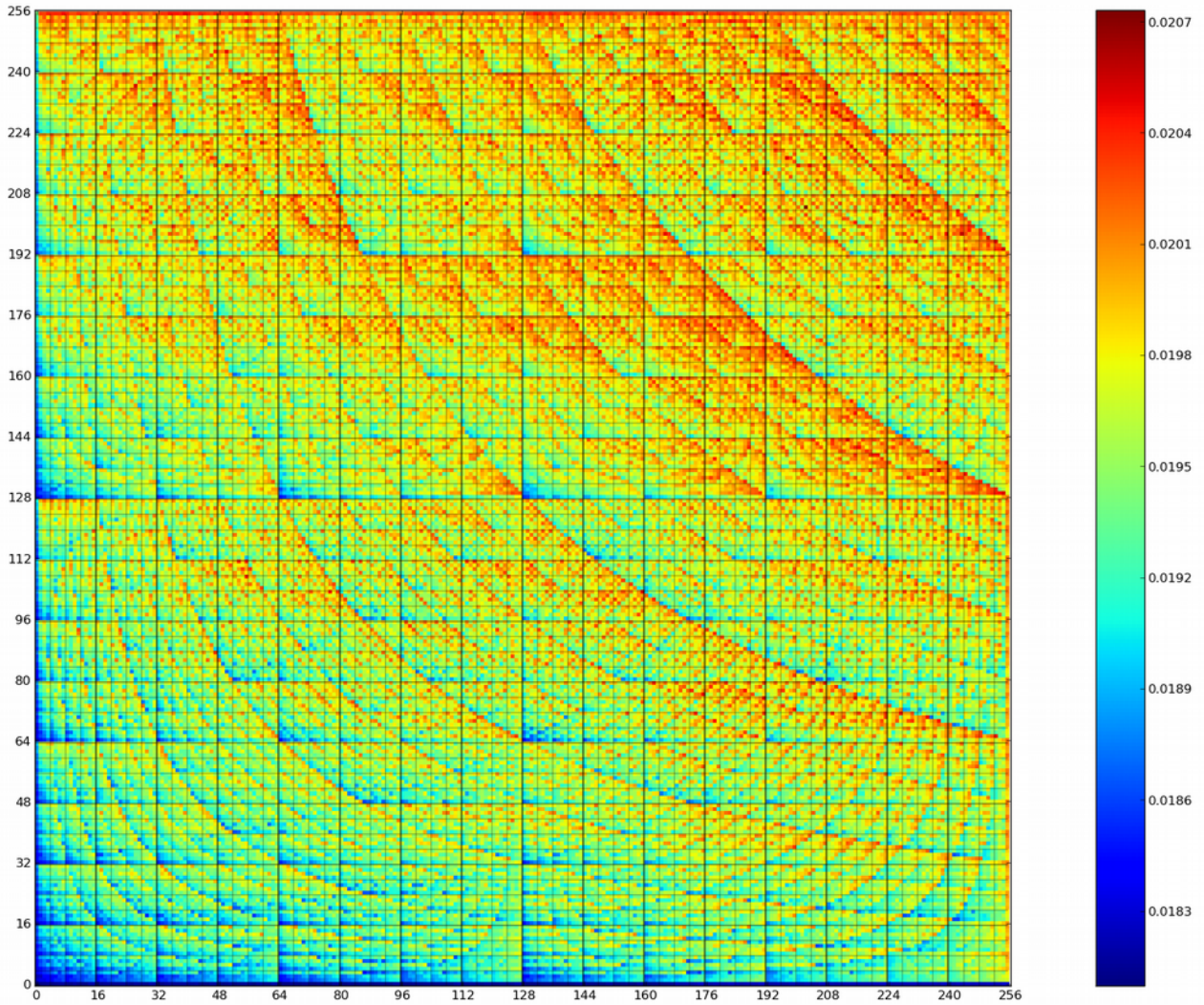


Image: Dr James Pallister, University of Bristol

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Bit Splitting By Hand

```
#include <stdint.h>

uint8_t k0, k1, k2, k3;

void inc_key32 ()
{
  k3++;
  if (0 == k3)
    {
      k2++;
      if (0 == k2)
        {
          k1++;
          if (0 == k1)
            k0++;
        }
    }
}
```

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Bit Splitting By Hand

```
#include <stdint.h>

uint8_t k0, k1, k2, k3;

void inc_key32 ()
{
  k3++;
  if (0 == k3)
    {
      k2++;
      if (0 == k2)
        {
          k1++;
          if (0 == k1)
            k0++;
        }
    }
}
```

- Spread key through memory
  - can't be scanned for

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Bit Splitting By Hand

```
#include <stdint.h>

uint8_t k0, k1, k2, k3;

void inc_key32 ()
{
  k3++;
  if (0 == k3)
    {
      k2++;
      if (0 == k2)
        {
          k1++;
          if (0 == k1)
            k0++;
        }
    }
}
```

- Spread key through memory
  - can't be scanned for
- This *ought* to work

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Bit Splitting By Hand

```
#include <stdint.h>

uint8_t k0, k1, k2, k3;

void inc_key32 ()
{
  k3++;
  if (0 == k3)
    {
      k2++;
      if (0 == k2)
        {
          k1++;
          if (0 == k1)
            k0++;
        }
    }
}
```

- Spread key through memory
  - can't be scanned for
- This *ought* to work
- But it is hard work
  - just byte splitting here!

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

EMBECOSM®

Bit Splitting By Hand

```
#include <stdint.h>

uint8_t k0, k1, k2, k3;

void inc_key32 ()
{
  k3++;
  if (0 == k3)
    {
      k2++;
      if (0 == k2)
        {
          k1++;
          if (0 == k1)
            k0++;
        }
    }
}
```

- Spread key through memory
  - can't be scanned for
- This *ought* to work
- But it is hard work
  - just byte splitting here!
- Optimization
  - might combine k0,k1,k2,k3
  - benefits wiped out

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Bit Splitting Automated

```
#include <stdint.h>

uint32_t k
  __attribute__ ((bitsplit));

void inc_key32 ()
{
  k++;
}
```

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Bit Splitting Automated

```
#include <stdint.h>

uint32_t k
  __attribute__ ((bitsplit));

void inc_key32 ()
{
  k++;
}
```

- Trivial for the programmer

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Bit Splitting Automated

```
#include <stdint.h>

uint32_t k
  __attribute__ ((bitsplit));

void inc_key32 ()
{
  k++;
}
```

- Trivial for the programmer
- Compiler can ensure operations are optimal

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Bit Splitting Automated

```
#include <stdint.h>

uint32_t k
  __attribute__ ((bitsplit));

void inc_key32 ()
{
  k++;
}
```

- Trivial for the programmer
- Compiler can ensure operations are optimal
- Compiler will never optimize

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Bit Splitting Automated

```
#include <stdint.h>

uint32_t k
  __attribute__ ((bitsplit));

void inc_key32 ()
{
  k++;
}
```

- Trivial for the programmer
- Compiler can ensure operations are optimal
- Compiler will never optimize
- Same dataflow issues as critical variables

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Other Areas To Investigate

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Other Areas To Investigate

- Atomicity – balancing control flow paths

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Other Areas To Investigate

- Atomicity – balancing control flow paths
- Superoptimizing for minimal leakage

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Other Areas To Investigate

- Atomicity – balancing control flow paths
- Superoptimizing for minimal leakage
- Algorithmic choice

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Other Areas To Investigate

- Atomicity – balancing control flow paths
- Superoptimizing for minimal leakage
- Algorithmic choice
- Instruction Set Extensions for minimizing leakage

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

EMBECOSM®

Other Areas To Investigate

- Atomicity – balancing control flow paths
- Superoptimizing for minimal leakage
- Algorithmic choice
- Instruction Set Extensions for minimizing leakage
- Instruction shuffling

Copyright © 2017 Embecosm. Freely available under a Creative Commons license

Thank You

www.embecosm.com
jeremy.bennett@embecosm.com