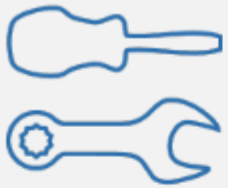
The SECURE Project and GCC

Security Enhancing Compilers for Use in Real-world Environments

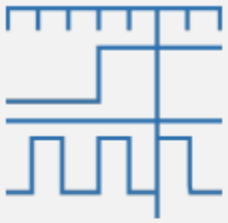
Speaker: Graham Markall – graham.markall@embecosm.com

Contributors: Jeremy Bennett, Craig Blackmore, Simon Cook, Ed Jones, Samuel Leonard, Paolo Savini

About Embecosm



Toolchain Porting >



Hardware Modeling >

Open Source Support >

Machine Learning Optimization >

Energy Efficient Compilation >

Superoptimization >

The SECURE Project

EMBECOSM®

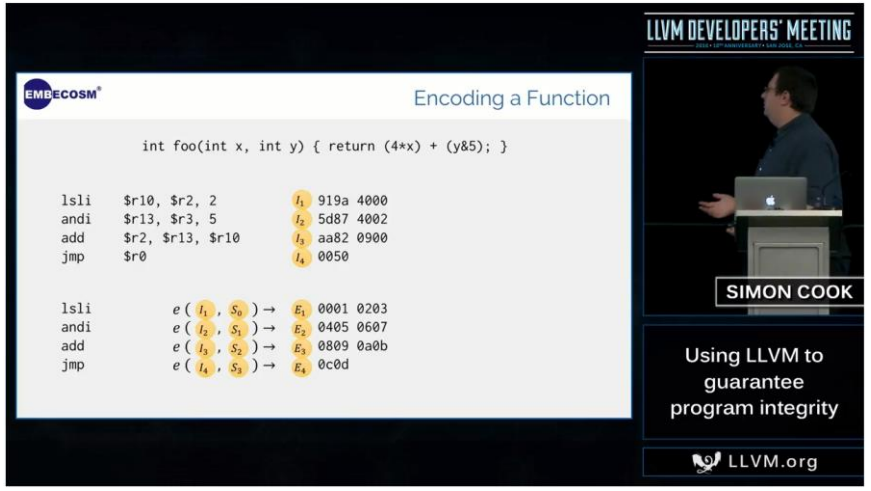
Innovate UK

Automation: *Reduce programmer effort*

Warnings: *Reduce programmer error*

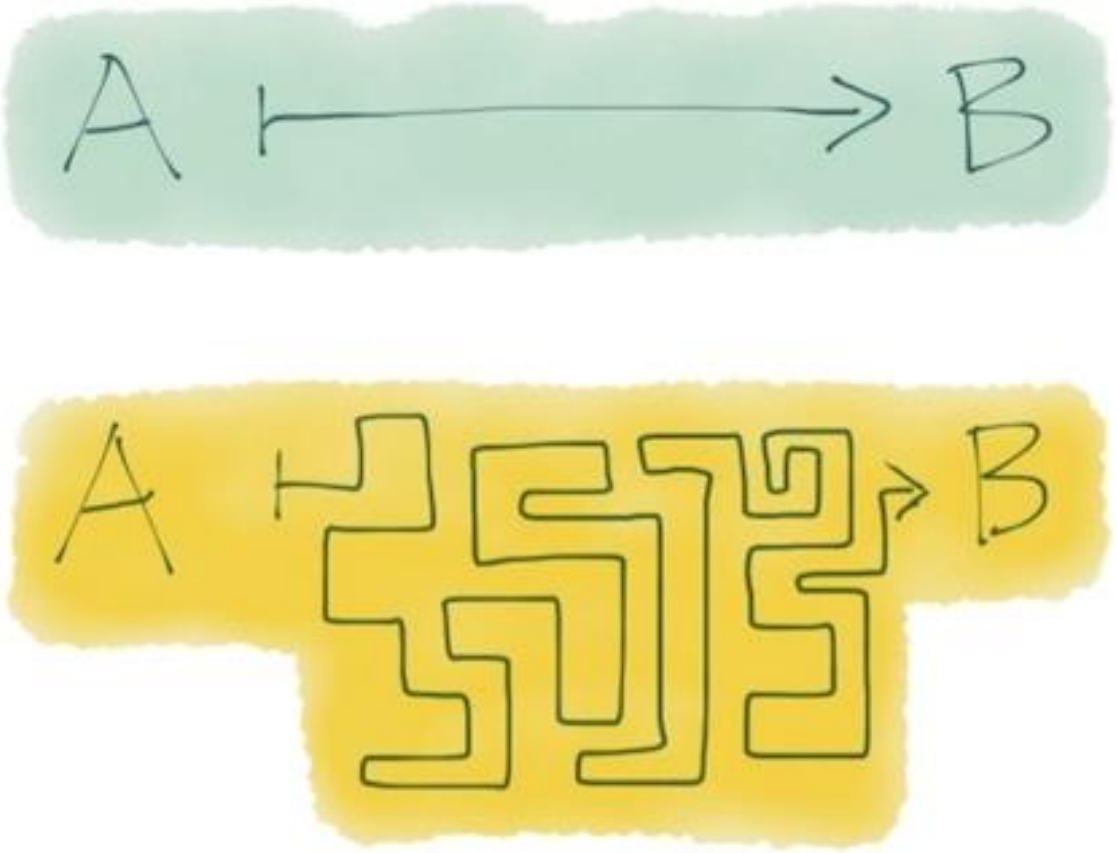
Academic / Industrial Contexts

- **LADA:** Leakage Aware Design Automation
  - Elisabeth Oswald
  - Dan Page
- Customers' Secure Processors
  - [Using LLVM to Guarantee Program Integrity]



Techniques

- **Stack Erase**
- Register Erase
- Sensitive Control Flow
- Defensive Stores
- Bit Slicing
- Control Flow Balancing



Key takeaway: *Conceptual simplicity* vs *practical complexity*

Stack Erase

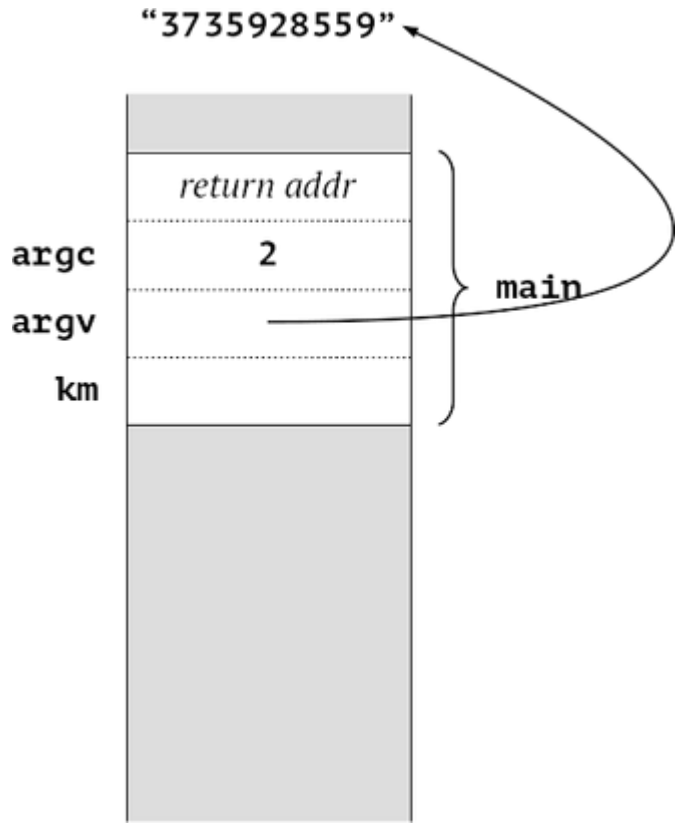
Stack Erase – problem example

```
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];
}
```

- Sensitive / secret variable **k** in **mangle**

Stack Erase – problem example

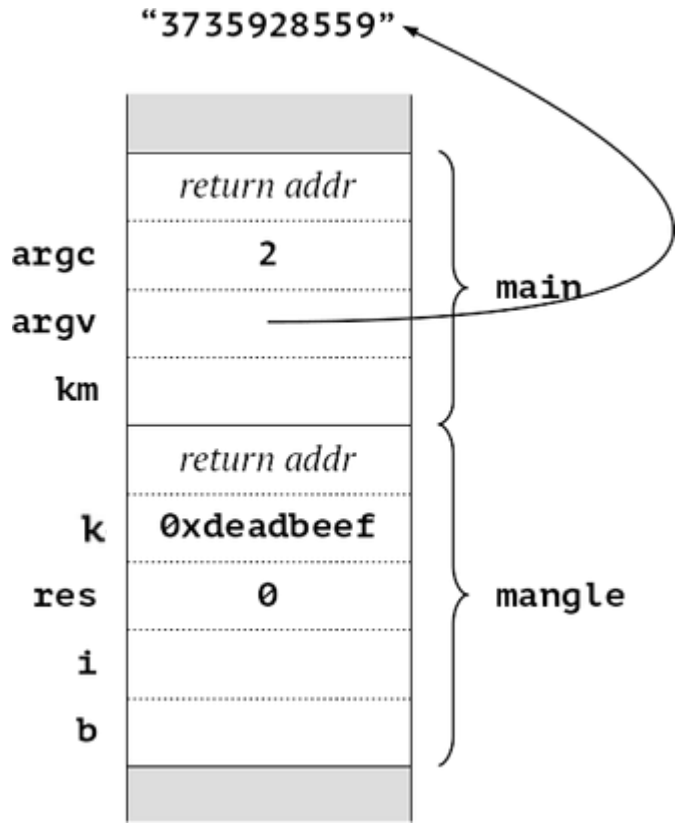
```
int mangle (uint32_t k)
  __attribute__ ((erase_stack))
{
  uint32_t  res = 0;
  int i;

  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];
}
```



- Sensitive / secret variable **k** in **mangle**

Stack Erase – problem example

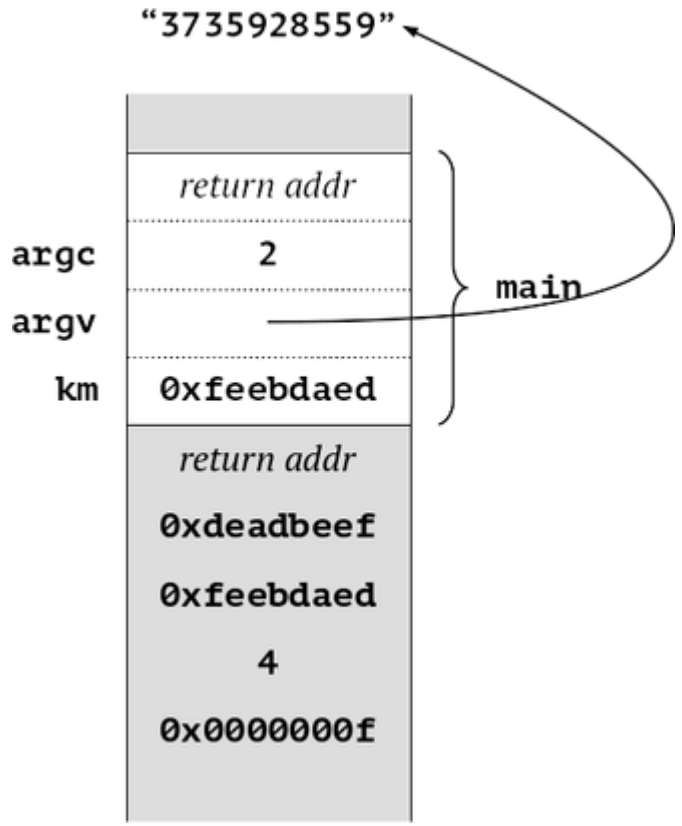
```
int mangle (uint32_t k)
  __attribute__ ((erase_stack))
{
  uint32_t  res = 0;
  int i;

  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];
}
```



- Sensitive / secret variable **k** in **mangle**

Stack Erase – problem example

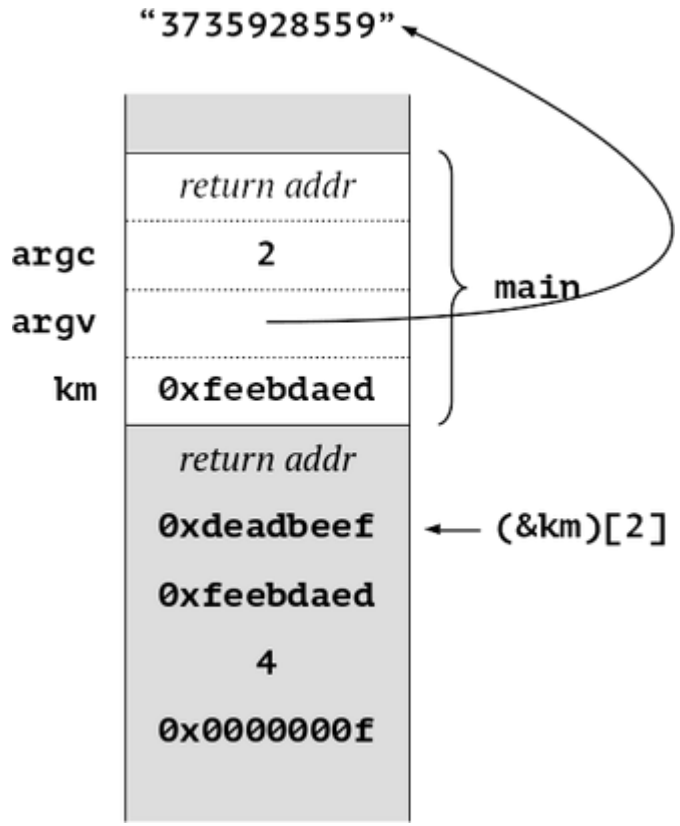
```
int mangle (uint32_t k)
  __attribute__ ((erase_stack))
{
  uint32_t  res = 0;
  int i;

  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];
}
```



- Sensitive / secret variable **k** in **mangle**
- Dead value on old stack frame still in memory

Stack Erase – problem example

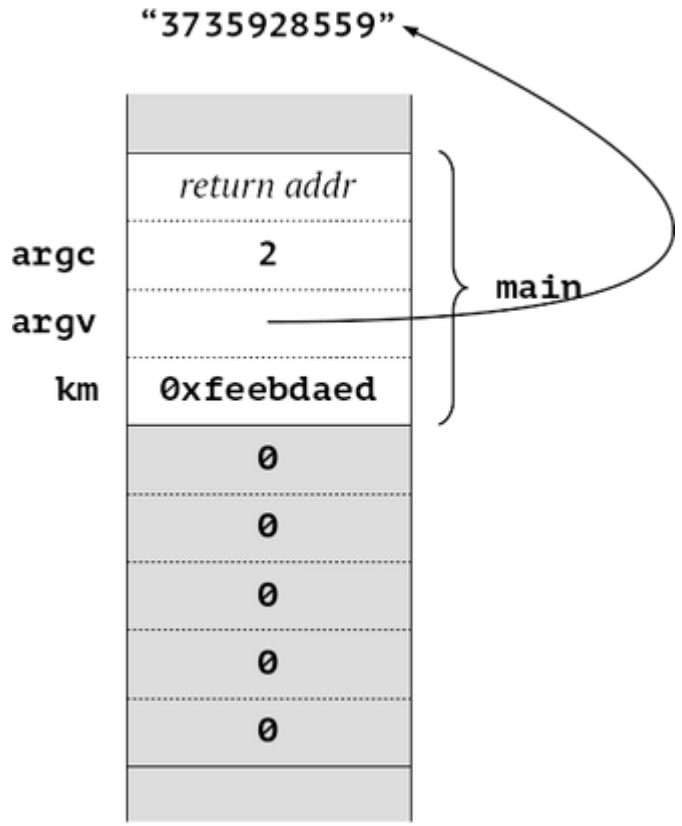
```
int mangle (uint32_t k)
  __attribute__ ((erase_stack))
{
  uint32_t  res = 0;
  int i;

  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];
}
```



- Sensitive / secret variable **k** in **mangle**
- Dead value on old stack frame still in memory
- Arbitrary read can leak value

Stack Erase – potential solutions

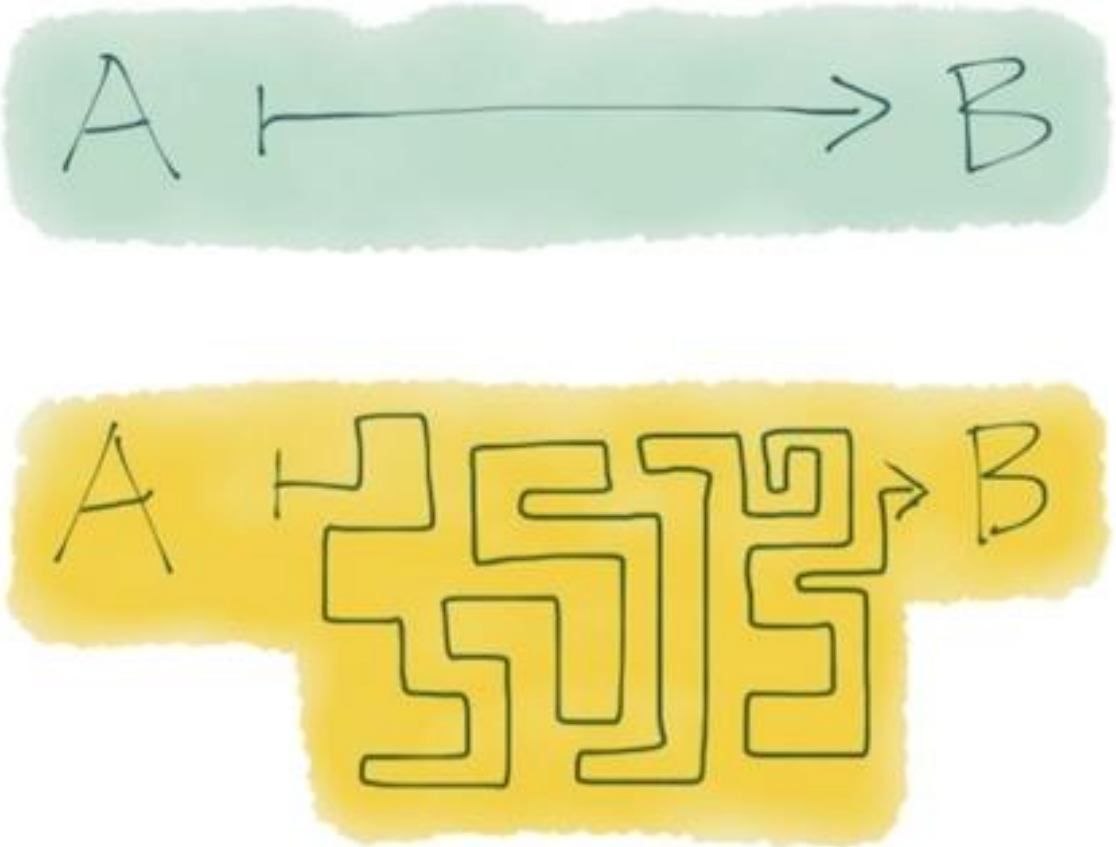
```
int mangle (uint32_t k)
  __attribute__ ((erase_stack))
{
  uint32_t  res = 0;
  int i;

  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];
}
```



- Cleverness required to work around compiler
- Ada:
  - Limited private types,
  - Inspection Points,
  - (Chapman, 2017)
- Compiler-supported stack-erase

Theory vs. practice

EMBECOSM®



Wishlist

- Widely applicable / usable implementation for most environments
- Few restrictions on its use (i.e. compatible with most language features)
- Easy to use (i.e. no gotchas that lull user into false sense of security)
- Robust (well-tested, again no false sense of security)

What you get is what you C: Controlling side effects in mainstream C compilers

Laurent Simon
Samsung Research America
University of Cambridge
lmrs2@cl.cam.ac.uk

David Chisnall
University of Cambridge
David.Chisnall@cl.cam.ac.uk

Ross Anderson
University of Cambridge
Ross.Anderson@cl.cam.ac.uk

Abstract

Security engineers have been fighting with C compilers for years. A careful programmer would test for null pointer dereferencing or division by zero; but the compiler would fail to understand, and optimize the test away. Modern compilers now have dedicated options to mitigate this.

But when a programmer tries to control side effects of code, such as to make a cryptographic algorithm execute in constant time, the problem remains. Programmers devise complex tricks to obscure their intentions, but compiler writers find ever smarter ways to optimize code. A compiler upgrade can suddenly and without warning open a timing channel in previously secure code. This arms race is pointless and has to stop.

and which lead to subtle failures. Implicit assumptions try to control side effects of source code; but the C standard is concerned only with effects that are visible within the C abstract machine, not about how those effects are generated. Providing explicit control for implicit assumptions is challenging.

There is a long list of implicit properties that would benefit from explicit controls (Section 3 on page 5). For example, cryptographic code has to run in constant time, to forestall timing attacks [2–7]. Unfortunately, there is no way to express this in C, so cryptographers resort to coding tricks instead – which a compiler can frustrate by optimizing away code that seems to do no "useful" work (Section 2 on the following page). And while a clever coder may outwit today's compiler, tomorrow's can quietly open

Function-based

- Annotated functions erase their stack:

```
__attribute__((stack_erase))
uint32_t mangle(uint32_t k) {
  …
  callee(…)
  …
  return x;
}
```

Overheads (Simon et al., 2018)

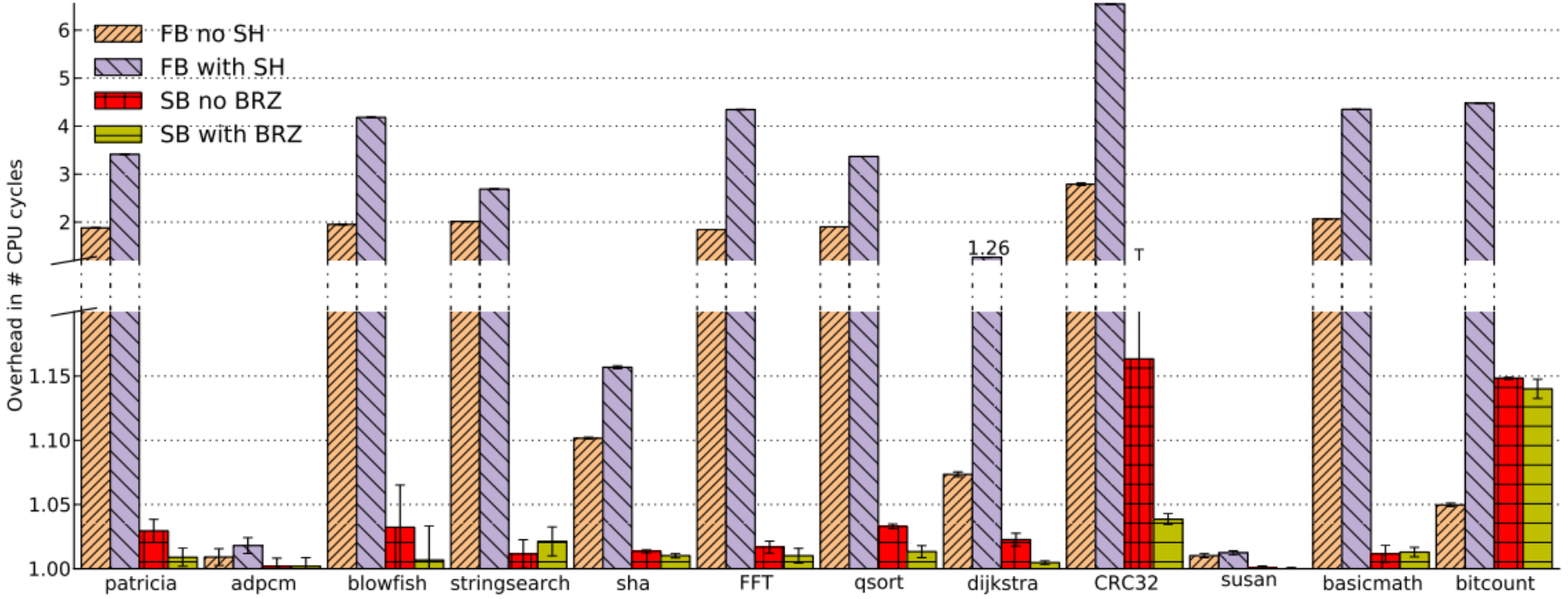


Figure 3: Runtime overhead for MiBench programs. The call graph solution (**CGB**) is omitted because there is no instrumentation (i.e. no additional overhead) besides the actual zeroing of the stack and registers.

- Function-Based (no SH) overhead is **1.86x execution time**

Stack-based

- Global variable in C library:

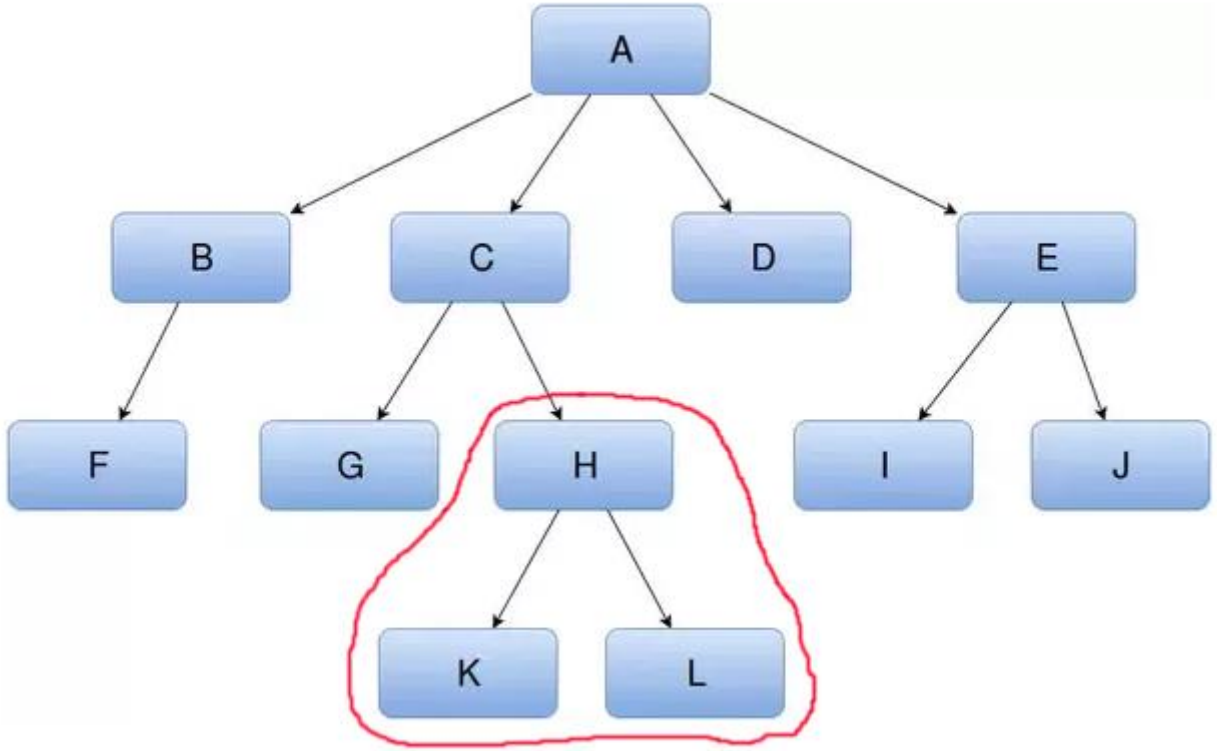
```
uintptr_t __StackPoint = 0;
```

- Function exit:

```
__StackPoint += max(__StackPoint, this_fn_stack_usage);
```

- Annotated function exit:
  - Erase between SP and (SP - __StackPoint)

Call graph-based



Comparison - observations

Approach	Advantage	Disadvantage
Function-based	• Straightforward implementation	• Slow
Stack-based	• Higher performance	• Requires instrumented libraries + global tracking
Call-graph based	• Highest performance	• Requires visibility of entire callgraph at compilation time

Comparison – from (Simon et al., 2018)

TABLE 3: Comparisons of Implementations. The first symbol indicates if a solution theoretically supports a feature; the second symbol indicates if our current implementation implements it. ✓indicates support; ✗no support.

	lazy binding	signal handling	call tail optimization	call graph cycles	asynchronous APIs	variable-size stack onjects	function pointers	PLT stubs
FB	✓– ✓	✓– ✓	✓– ✗	✓– ✓	✓– ✓	✗– ✗	✓– ✓	✓– ✗
SB	✓– ✓	✓– ✓	✓– ✗	✓– ✓	✓– ✓	✗– ✗	✓– ✓	✓– ✗
CGB	✓– ✗	✓– ✓*	✓– ✓**	✗– ✗+	✗– ✗	✗– ✗	✓– ✓++	✓– ✗

Implementation in GCC – choices!

- Function-based
  - Others depend on environment or need whole program source
  - Only dependency on environment is longjmp
  - Enforce consistency between prototype and definition
  - Also erase stack when restoring stack pointer
  - No inlining of stack-erase functions
- RISC-V and x86

Implementation in GCC – choices! (2)

- No register erase
  - Arbitrary memory read a larger class than arbitrary register read
  - Memory sensitive values longer-lived than register values
- No signal handling consideration
  - Defer to environment

Implementation in GCC – choices! (3)

- Always erase stack with zeroes:
  - Alternative: use random values
    - Zeroing can leak stack values via EM side channels
  - Problem: good randomness is hard to find
    - especially in embedded systems
  - Constraint: We don’t consider side channels for this work

Consistency requirement


```
int function(int a, int b);

__attribute__((stack_erase))
int function(int a, int b) { … }

$ riscv32-unknown-elf-gcc impl.c -c

impl.c:4:5: error: 'stack_erase' attribute present on 'function'
 int function(int a, int b)
     ^~~~~~~~
In file included from impl.c:1:
header.h:3:5: error: but not here
 int function(int a, int b);
     ^~~~~~~~
```

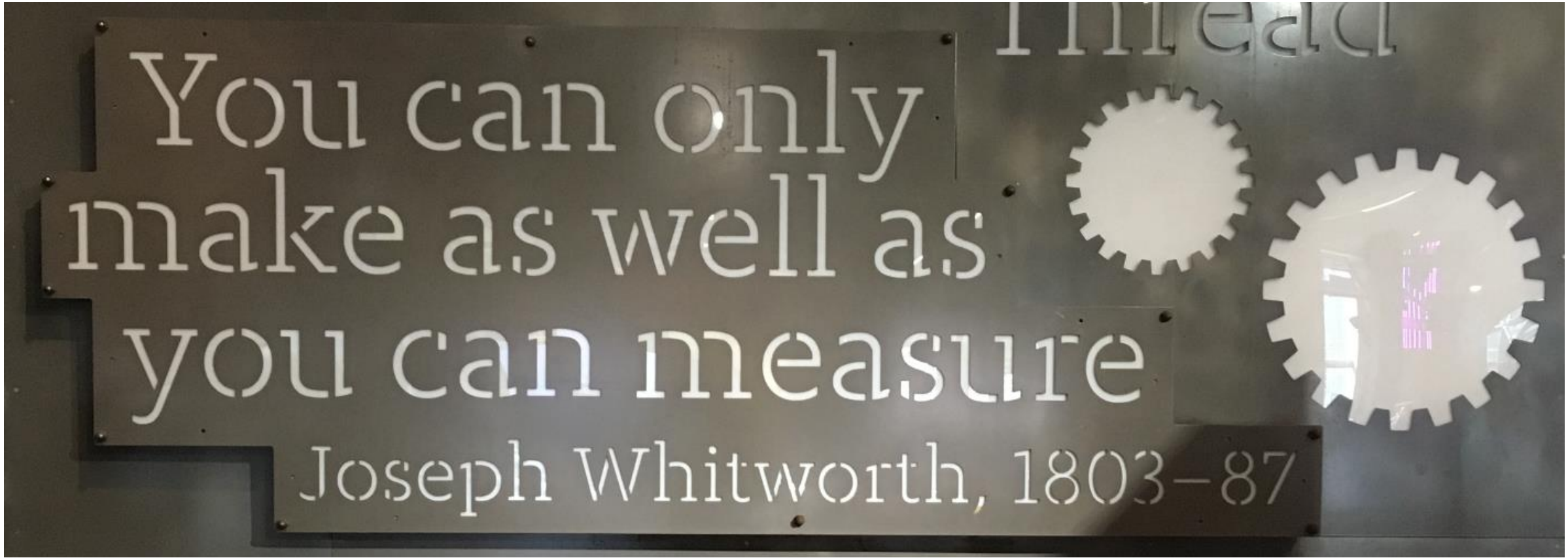
Function pointer calls


```
__attribute__((stack_erase))
int function(int (*myfunc)(int, int), int a, int b) {
  myfunc(a, b);
}
```

```
$ riscv32-unknown-elf-gcc impl.c -c
```

```
impl.c:4:3: error: cannot call using function pointer
'myfunc' from stack-erase
 myfunc(a, b);
 ^~~~~~
```

Testing


You can only
make as well as
you can measure
Joseph Whitworth, 1803–87

Testing (1) – testsuite tests


```
int test(int (*func) (int, int))
{
  // Zero 2kb of stack space and record SP
  void *initial_sp;
  __asm__ volatile (
    "  addi    t1, sp, -2048\n …"
   : "=r" (initial_sp) : : );

  // Call test function
  func(7, 5);

  // Check stack pointer is unchanged after return and that stack is zeroed.
  int stack_check;
  __asm__ volatile (
  "  bne     sp, %1, 3f\n"
  …
```

Testing (2) – default stack erase

- Configure GCC with `--enable-default-stack-erase`

- Modify _start:
  - Before calling main:
    - Save stack pointer
    - Zero out X KB beneath SP
  - Call main()
  - After main:
    - Check SP == Saved stack pointer
    - Ensure X KB beneath SP are still zero

- Modify `longjmp` – erase between old SP and new SP

Test results - RISCV

- GCC:

	Base	SE Patch	Δ
# of expected passes	91545	91577	32
# of unexpected failures	7	7	0
# of unexpected successes	2	2	0
# of expected failures	205	205	0
# of unsupported tests	2414	2438	24

- G++:

	Base	SE Patch	Δ
# of expected passes	99877	99907	30
# of unexpected failures	22	22	0
# of expected failures	442	442	0
# of unsupported tests	5180	5190	10

- New passes: stack-erase tests
- New unsupported: atomics (?)

Test results – x86

- GCC:

	Base	SE Patch	Δ
# of expected passes	132721	132709	-12
# of unexpected failures	258	331	73
# of expected failures	411	411	0
# of unresolved testcases	12	12	0
# of unsupported tests	2153	2153	0

- G++:

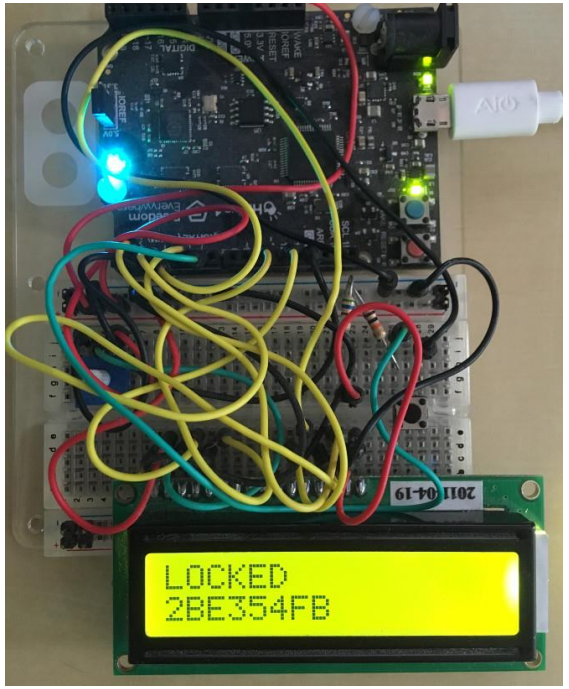
	Base	SE Patch	Δ
# of expected passes	124834	124821	-13
# of unexpected failures	231	243	12
# of expected failures	523	523	0
# of unsupported tests	5019	5019	0

Causes for new fails:

- Scan-assembler-not, scan-assembler-times, etc…
- Varargs?
- Assumption that dead stack values don’t change!

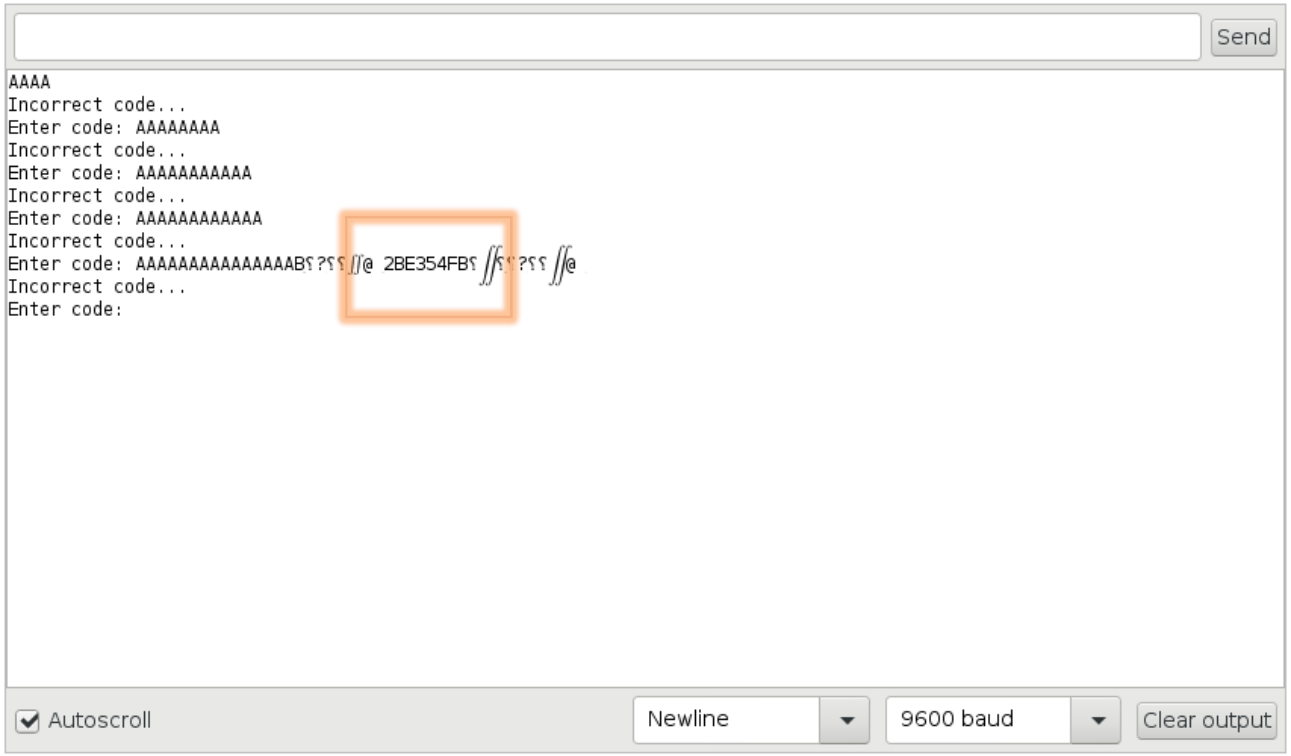
Practical usage – demo application

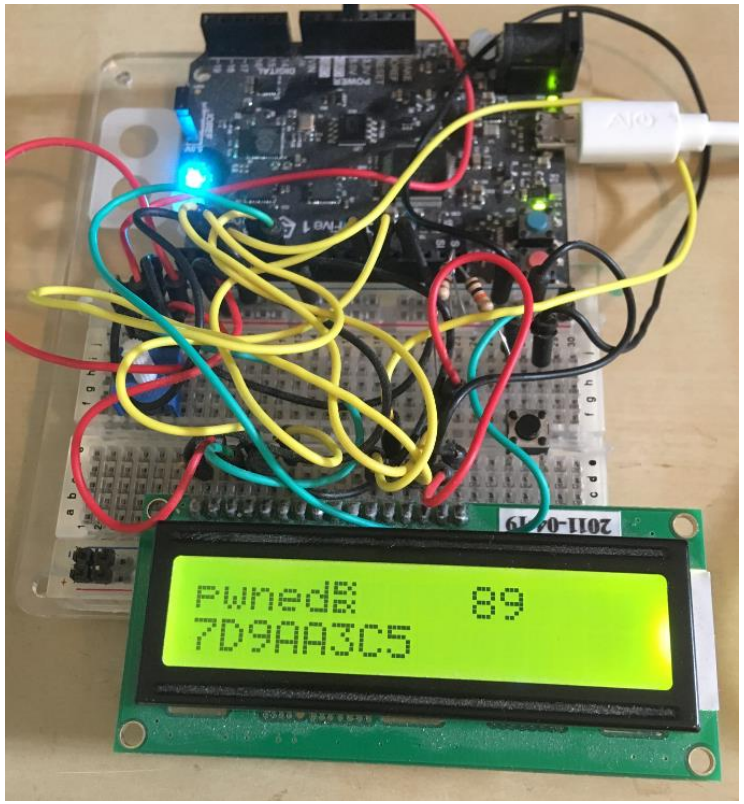
https://www.embecosm.com/2018/08/10/protecting-secret-data-with-stack-erase/




```
// A simplified version of the readSerialBuf function
char* readSerialBuf() {
  char buf[8];
  uint32_t bufLoc = 0;
  while (true) {
    if (Serial.available()) {
      buf[bufLoc] = Serial.read();
      if (buf[bufLoc] == '\n') {
        Serial.write(buf, bufLoc+1); return;
      }
      bufLoc++;
    }
  }
}
```

Exploitation - memory read / RCE





Protecting the secret

- Header file:

```
class Print {
private:
  int write_error;
  __attribute__((stack_erase))
  size_t printNumber(unsigned long, uint8_t);
  // ... lots more functions ...
```

- Source file:

```
__attribute__((stack_erase)) size_t
Print::printNumber(unsigned long n, uint8_t base) {
  char buf[8 * sizeof(long) + 1]; // Assumes 8-bit chars plus zero byte.
  char *str = &buf[sizeof(buf) - 1];

  *str = '\0';

  …
```

Stack Erase – summary

Towards an implementation that is:

- Widely-usable
- Few restrictions
- Easy to use
- Robust

- Patch to be submitted ASAP!