Secure Compilation of Safe Erasure
(work in progress)

Frédéric Besson with Thomas Jensen and Alexandre Dang

Inria Rennes

Dagstuhl seminar - May 2018

Erasure of Sensitive Data

(see Kedar's Talk)

Alice fetches a secret data s
"s is stored in memory in clear text for some time"

Bob may be scanning the memory
$\Rightarrow$ Alice is erasing s ASAP

```
sign(d){
  byte[keysz] s;
  fetchKey(&s);
  sd = signWithKey(d,s);
  memset(s,0,keysz);
  return sd;
}
```

Informal security

- Secret s may be vulnerable during `signWithKey`
- Secret s is not vulnerable after `memset`

CERT MSC06-C: Beware of compiler optimisations

An actual implementation need not evaluate [...] an expression if it can deduce that its value is not used [...]

Compliant C99 code

```
errno_t memset_s(void *v, rsize_t smax, int c, rsize_t n) {
  if (v == NULL) return EINVAL;
  if (smax > RSIZE_MAX) return EINVAL;
  if (n > smax) return EINVAL;
  volatile unsigned char *p = v;
  while (smax-- && n--)
    *p++ = c;
  return 0;
}
```

Check compiler documentation and the assembly output from the compiler.

Compliant C11 code

[...] any call to the memset_s function shall be evaluated strictly [...]. That is, [we] shall assume that the memory indicated by s and n may be accessible in the future [...].

Problem Solved?

memset_s is rarely implemented
gcc/clang std=c11
⇒ `implicit declaration of function memset_s`

Dead store (still) considered harmful. Usenix Sec'17

What you get is what you C:Controlling side-effect in mainstream C compilers. EuroS&P'18

Preservation of Erasure as an Information Flow Property

Attacker model: arbitrary memory access
(at specific time)

Secure compiler:
A low-level attacker learns no more than a high-level attacker
$\Rightarrow$ low-level programs leak less

What kind of secure compiler?

Please do not break my code, please.
$\Rightarrow$ Compiler should do less...

High-level attacker $\equiv$ low-level attacker
Same observation power

Testing the Definition

Is following transformation safe?

```
sign(d){
  byte[keysz] s;
  fetchKey(&s);
  sd = signWithKey(d,s);
  memset(s,0,keysz);
  return sd;
}
```

```
sign(d){
  byte[keysz] s;
  fetchKey(&s);
  sd = signWithKey(d,s);
  // memset(s,0,keysz);
  return sd;
}
```

Expected answer: no!

Testing the Definition

What about the following program?

```
sign(d){
  byte[keysz] s;
  fetchKey(&s);
  sd = d xor s;
  memset(s,0,keysz);
  return sd;
}
```

```
sign(d){
  byte[keysz] s;
  fetchKey(&s);
  sd = d xor s;
  // memset(s,0,keysz);
  return sd;
}
```

1. The source-level attacker can reconstruct the secret. . .
2. The low-level attacker can do the same

$\Rightarrow$ zeroing the secret does not increase security):

Weaker attacker, stronger property

Weaker attacker: can only pick a choosen bit of information

```
sign(d){
  byte[keysz] s;
  fetchKey(&s);
  sd = d xor s;
  memset(s,0,keysz);
  return sd;
}
```

```
sign(d){
  byte[keysz] s;
  fetchKey(&s);
  sd = d xor s;
  // memset(s,0,keysz);
  return sd;
}
```

1. The source attacker cannot get the secret
2. The low-level attacker can

$\Rightarrow$ Transformation is not secure :)

Not convinced? Let's test again.

```
foo(x,y,a){
  a = 0;
}
```

```
foo(x,y,a){
  if(a)
   a = x;
  else
   a = y
}
```

1. 1-bit attacker cannot learn initial value of a
2. 2-bit attacker correlate a with (x,y)

Transformation is 1-secure but not 2-secure

Let quantify over the number of observed bits.
Secure $\equiv \forall n, n{-}secure$

Secure optimisations

Constant folding/propagation, Common Sub-expression Elimination

Sufficient condition: same memory

Low-level attacker = Source attacker.

Dead Store Elimination

See motivating example

Solution: enforce computation of same memory
$\Rightarrow$ At function exit, ALL variables are live

What is the impact of efficiency?

Peephole optimisations

Use the same exact registers
$x * 2 \rightsquigarrow x << 2$
$\Rightarrow$ No (more) information leak

Need dead registers for intermediate results

1. Dead but eventually live over all paths
2. Explicitly erase ; run (safe) dead store

Register allocation

Spilling duplicates information
register $\rightsquigarrow$ stack slot

A stack slot may outlive its register

Solution: ensure a mapping:
(stack slot $+$ register) $\rightarrow$ pseudo-register ($+$ cst)

Impact on register allocation?

Work in progress

Formalisation in terms of Attacker Knowledge
(limited to terminating, deterministic programs)

Formal proof of (modified) compiler passes
Eventually on CompCert, proof technique:
relational refinement / 2-simulation

Open questions:

- Is duplicating information ok?
- Are optimisations easy to fix?
- Are they still effective?
- What happens at the bottom?