(Un)Encrypted Computing and Indistinguishability Obfuscation

Peter T. Breuer

ptb@hecusys.com

Hecusys LLC, USA

Jonathan P. Bowen

London South Bank University, UK

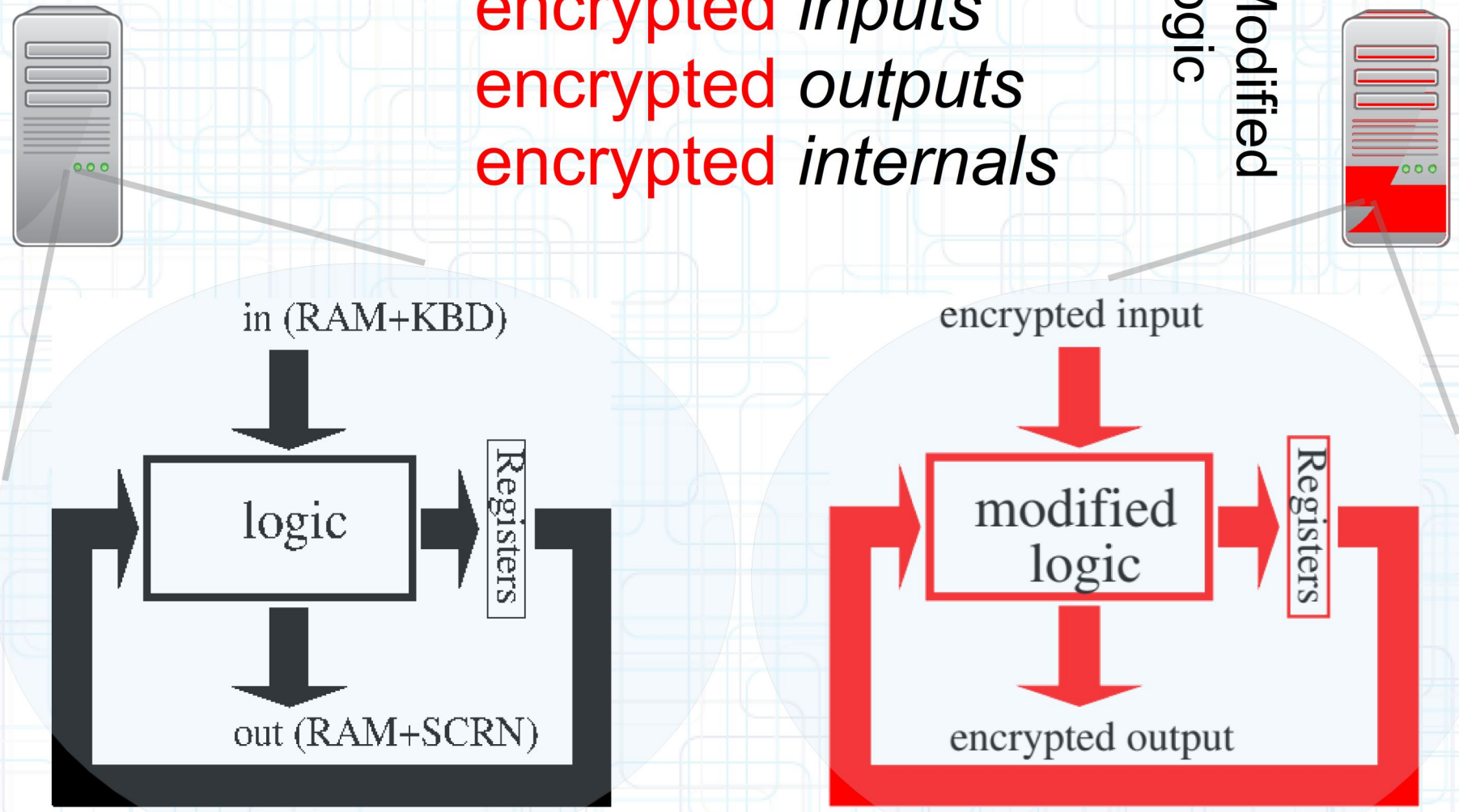
To look but not to see

- In this talk I will relate recent successes
  - in compilation for Encrypted Computing
  - to code obfuscation

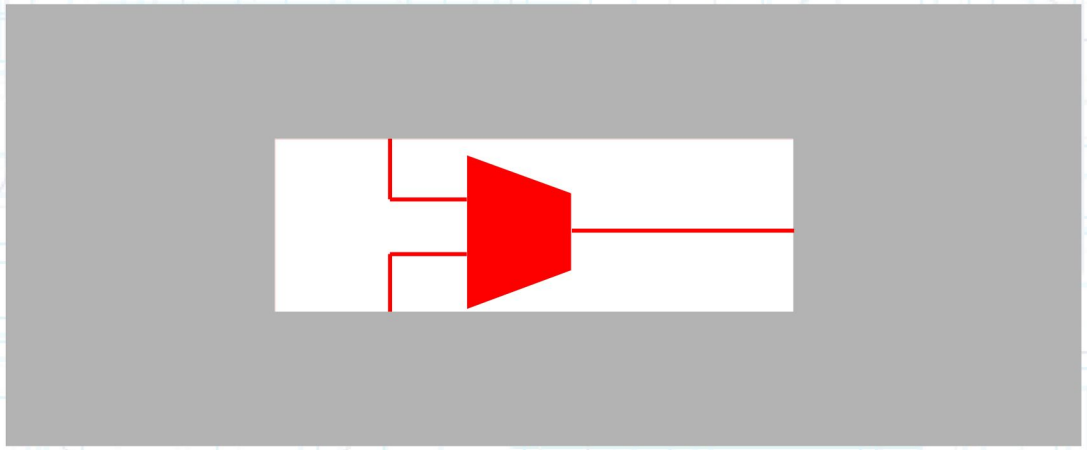
Crash course: Encrypted Computing

Modified Processor

encrypted *inputs*
encrypted *outputs*
encrypted *internals*



To answer an unasked question

- “Modified logic” is really only a
  - Modified arithmetic unit
  - Within standard processor logic
- 
- 2013 paper shows that is enough for encrypted (1-to-many) computing

Of importance in Encrypted Computing is a proof ...

Adversary has no method to decipher run-time data

code

```
0: if x[<]E[15] goto 8
4: x ← x[+]E[1]
8: ...
```

trace

```
x=E[16]

x=E[17]
```

Notation: $E[u]\ \mathbf{[o]}\ E[v] =_D E[u\ o\ v]$

$E[u]\ \mathbf{[R]}\ E[v] \Leftrightarrow u\ R\ v$

$E[u] =_D E[v] \Leftrightarrow u = v \Leftrightarrow E[u]\ [=]\ E[v]$

To adversary (who cannot read the encryption) it looks like …

```
0: if x[<]E[ ] goto 8
4: x ← x[+]E[ ]
8: …
```

```
x=E[ ]
x=E[ ]
```

Thought experiment …

Add 7 to run-time data and adjust program constants

code

```
0: if x[<]E[22] goto 8
4: x ← x[+]E[1]
8: …
```

trace

```
x=E[23]

x=E[24]
```

That looks same as before to adversary

```
0: if x[<]E[ ] goto 8
4: x ← x[+]E[ ]
8: …
```

```
x=E[ ]
```

```
x=E[ ]
```

Unchanged code structure, trace!

Deduce ...

- Different data underneath, but codes and traces look the same to adversary
- The *method* for deciphering data must give different answer from same observations
- So adversary's *method* cannot exist. QED

Further examination of proof ...

1. Shows *runtime* data *offset* “+7” can be varied
   - per each instruction input, output, per instruction
   - per memory location, per instruction
   - Independently, with uniform flat distribution
   - Then No stochastic decryption attack can be right more than chance (Goldwasser&Micali *cryptographic semantic security*)
2. Requires
   - `Obfuscating’ compiler varies arithmetic instructions
     - maximally
   - Encryption is *independently secure*

`Obfuscating' Compiler

- HAVOC for ansi C
  - `obfuscates' by offsetting ("+7") code constants
    - with maximal entropy
  - Produces maximal entropy offsets throughout traces
    - Trace structures all the same
    - Code structures all the same
- Zen: "One obfuscation looks just as random as any"
  - Basic compiler constructs already obfuscatory
    - E.g. to write one array entry it writes all in a "write storm"
      - In randomly generated order

Example – Ackermann(3,1)

```
...
176     add  t0  v0  zer E[1155534589]  t0 = E[1155534602]
196     lw   c9  E[0](sp)               c9 = E[0]
216     add  t1  t0  zer E[-262567953]  t1 = E[892966649]
236     add  t2  sp  zer E[1351769372]  t2 = E[892966636]
256     sw   E[-892966636](t2) t1       [E[0]] = E[892966649]
276     add  t0  sp  zer E[1351769372]  t0 = E[892966636]
296     lw   t0  E[-892966636](t0)      t0 = E[892966649]
316     add  t0  t0  zer E[-937042374]  t0 = E[-44075725]
336     add  t0  t0  zer E[-486720728]  t0 = E[-530796453]
356     add  v0  t0  zer E[530796466]   v0 = E[13]
376     lw   ra  E[-2](fp)              ra = E[0]
396     cmov sp  fp  fp                 sp = E[0]
400     lw   fp  E[-1](sp)              fp = E[0]
420     jr   ra
STOP
```

Result = 13

Random program constants

Random trace entries

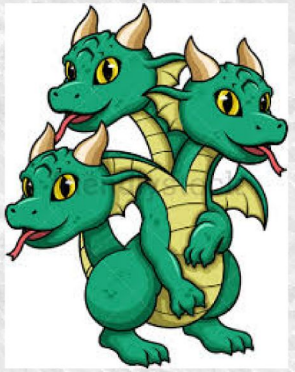
Doing it without encryption

Invent *secret offsets* for program variables x,y,z

- Say 8 for x when you mean 1 (+7)
- Say 13 for y when you mean 1 (+12)
- Say 25 for z when you mean 2 (+23)
- z ←x+y+4
- Z+23 ←X+7+Y+12+4
- Z ← X+Y+0
- See 25 = 8+13 (+4) but understand 2=1+1 (+0)

- Zen: program and trace is understandable many ways

Worked example: One program is many



- Loop offsets at start and end must be equal
- introduce ‘trailer’ x← x+k instructions to even up offsets

- while x < y+z+1
  x ← x +2; x← x+3;
- while X+4 < Y+5+Z+6+1
  X+7←X+4+2; X+4←X+7+3;
- while X < Y+Z+8
  X←X-1; X←X+6;
- Choice of **4**,**5**,**6**,**7** was arbitrary
- One code, three variables X,Y, Z, 2^{4x32} ways to read it

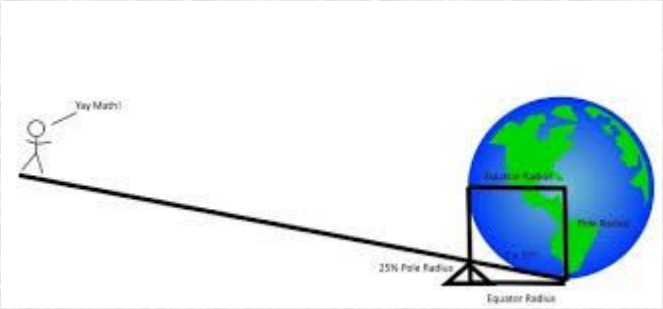
Pause for summary

- Each program can be read many ways by changing mind on what the variables stand for
- There are many ways to write the same program by changing the actual offset of the data at each point
- I can RE-compile each program many ways
- Re-compiler iO is indistinguishability obfuscator

Limited Indistinguishability

- Indistinguishability obfuscator preserves function
  - fun[iO(C)] = fun[C]
- Possible recompilations C1,C2 of one program
  - There is no way to tell iO(C1) from iO(C2)
  - Runtime traces have the same structure
  - Program codes have the same structure
  - Runtime trace *data* and program *constants* differ

Can this be leveraged?



- Indistinguishability obfuscation should camouflage any (within reason) programs with same function
- Not just variations f(x+A)+B of f(x) !

Crazy Idea

- Any finite state transformation
  - Is representable as a constant vector (f(0),f(1),…)
- To get f(x) just *lookup* the xth entry
  - Differs by constant B from any other constant
- Then any two state transformations are the *same program* with *different embedded constant*

- iO recompiler equates any such state transforms

What's wrong with that idea?

- Takes exponentially many operations to find the constant vector that makes any state transform program into a lookup function
- But …

  There may be classes of programs that take only polynomial time to transform to a common programmatic form, differing only in constants
- *Those* would be equated by the iO recompiler
  - E.g. variants of AES encryption

The prime sieve (array/memory ex.)

```
...
22504   add  t0  sp  zer E[-208591572]  t0 = E[-667394321]
22524   lw   t0  E[667394320](t0)       t0 = E[-667394313]
22544   add  t0  t0  zer E[218457017]   t0 = E[-448937296]
22564   add  t0  t0  zer E[493589813]   t0 = E[44652517]
22584   add  v0  t0  zer E[-44652510]   v0 = E[7]
22604   lw   ra  E[-2](fp)              ra = E[0]
22624   cmov sp  fp  fp                 sp = E[0]
22628   lw   fp  E[-1](sp)              fp = E[0]
22648   jr   ra
STOP
```

Random memory address

Result = 7 (a prime)

Memory *addresses* are random, uniformly distributed
- Hardware must support that
- To maintain entropy, address of memory location is changed before every write

- Any obfuscatable class of programs must share same memory access patterns
  - W1 W2 R1 R2 R1 W3 …
  - Best to rewrite every memory location after read, avoiding read trains RRR

Conclusion

1. Obfuscation must happen at machine code level
   - Source code obfuscation useless with clever compiler
2. Processor hardware can help with obfuscation
   - E.g. by internally shuffling registers/offsetting them
     - Using pseudorandom sequence, seed known to compiler
   - My own processor helps by working encrypted
     - Full range random memory addressing requires special support
3. Have prototype obfuscating (re)compiler:

HAVOC