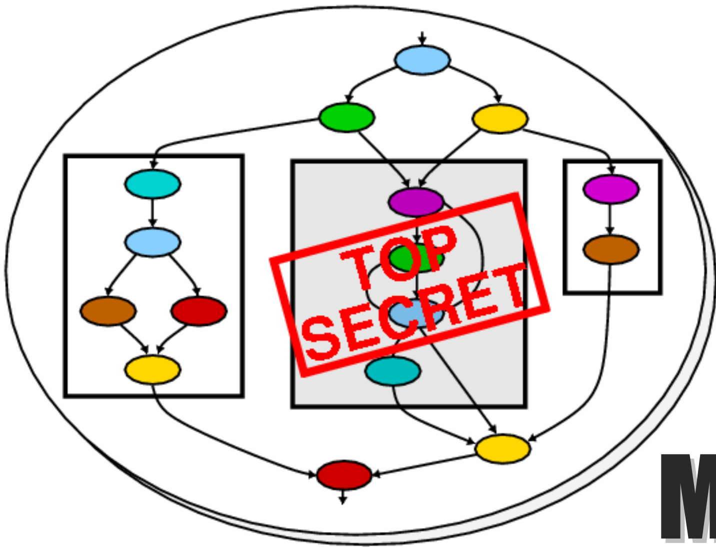


Memoization Attacks and Copy Protection in Partitioned Applications

Charles W. O'Donnell[1], G. Edward Suh[2]
Marten van Dijk[1], Srinivas Devadas[1]

[1]Massachusetts Institute of Technology
[2]Cornell University

IEEE Workshop on Information Assurance
June 22, 2007

CSAIL
MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY

Motivation

- ***Central concern: Intellectual Property (IP) Protection*** of applications
  - Prevent piracy, hide sensitive algorithms, etc

- Stop attacker from ***reproducing functionality*** of "protected" software code
  - Only some small regions of application may need protection

- ***Operational functionality:*** ultimate test of security
  - ***Unimportant:*** contents of protected code
  - ***Important:*** How protected code is used,
    How attacker can bypass code and still get "useful" results

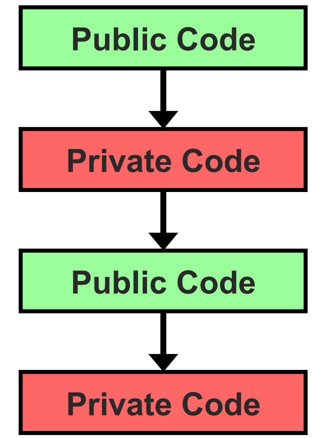
- One solution: Fully encrypt application
  - Requires: Secure CPU/Co-Processor, remote servers
  - Prevents piracy by requiring a key to execute
  - Speed/power/etc ***overheads***

```
addi    r3,r4,16
lw      r5,0(r15)
sub     r6,r5,r3
sw      4(r15),r6
addi    r11,r6,r5
```

Partitioned Applications

- ***Partitioned Application:*** only encrypt portions of application
  - May provide same security
  - Tradeoff ***security vs. speed***

```
addi    r3,r4,16
lw      r5,0(r15)
sub     r6,r5,r3
sw      4(r15),r6
addi    r11,r6,r5
```



- Architecture guarantees secret execution of encrypted code
  - Only memory accesses in and out of encrypted code region are visible
  - More details later

- ***Central Question:*** Deciding which regions of an application to encrypt

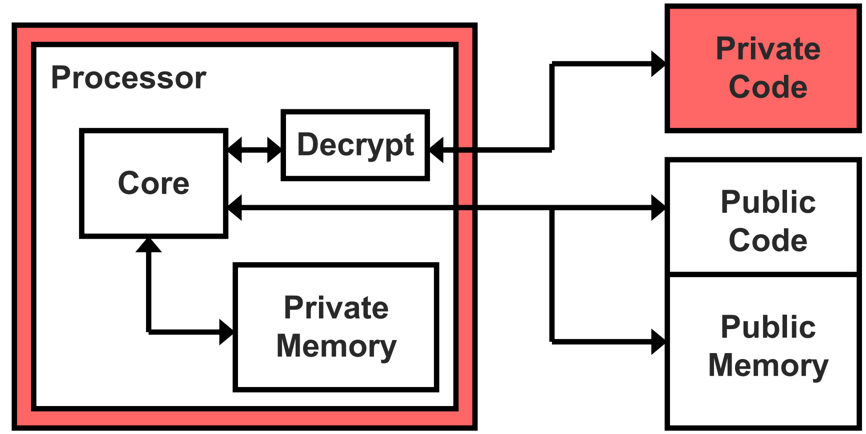
- ***Key Point:*** Naïve separation insecure
  - Designers must make a balanced decision based on how encrypted region will be used in the application at large

Presentation Outline

- Model
  - Define partitioned application and a very limited adversary
- Memoization Attacks
  - Describe problem and method of attack
- Implementing a Memoization Attack
  - Practical issues when performing attack
  - Attack results on real applications
- Indicators of Insecurity
  - Simple omens for when a Memoization Attack will succeed
  - Indicator accuracy results on real applications
- Related Work
  - Long standing research problem

Partitioned Applications Details

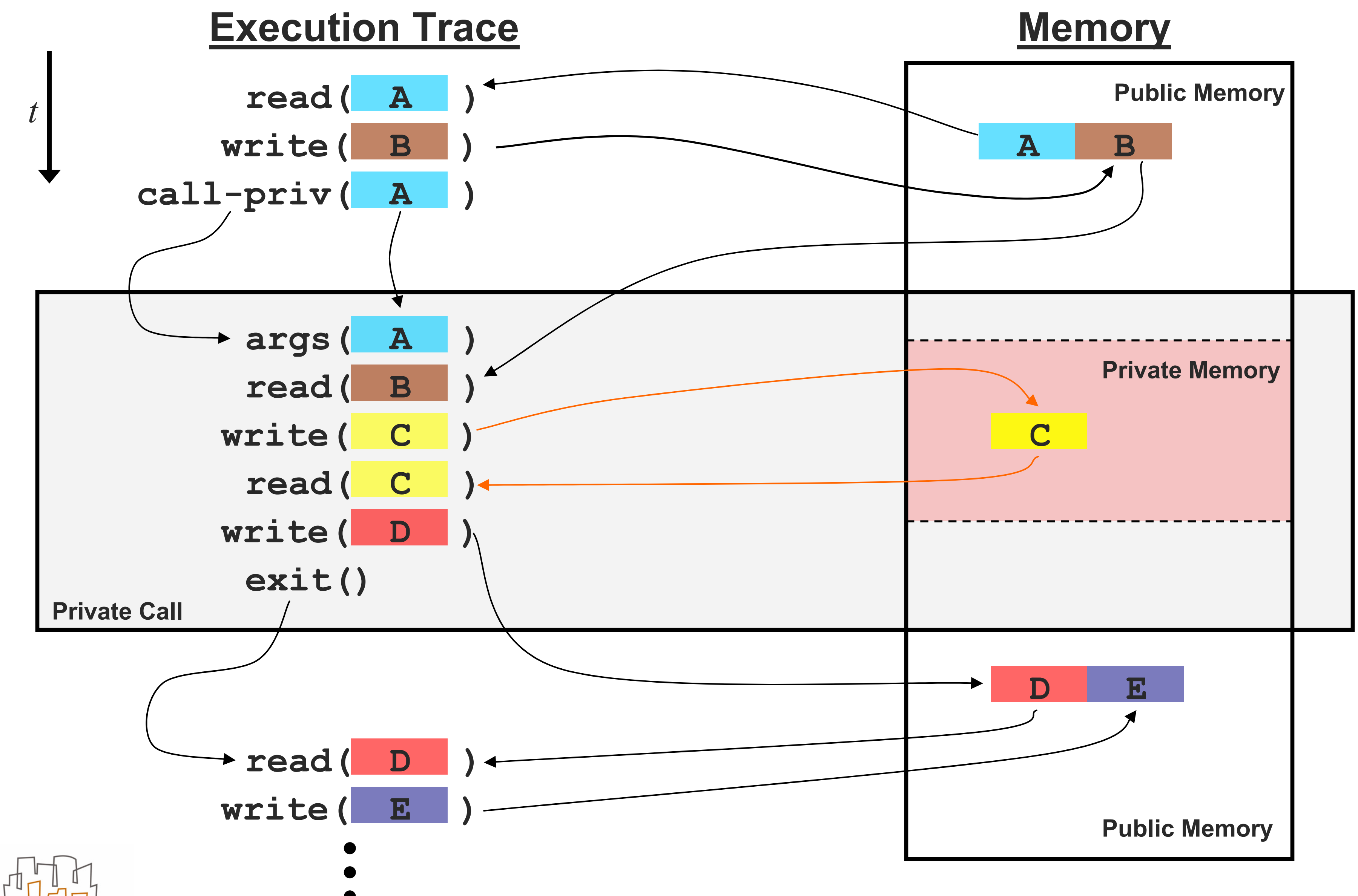
- Application code
  - encrypted ***private*** regions
  - unencrypted ***public*** regions



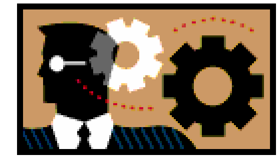
Example Secure Architecture

- Private regions
  - Executes ***secretly***
  - Access special private memory ***secretly***
  - Can access regular public memory
- Simplifying assumptions:
  - ***Procedures*** are fundamental region units
  - ***No private state between calls*** (Common case)
  - For experiments: in-order memory, no cache
- Adversary observes memory bus to attack

Observing a Partitioned Application



What an Adversary Knows

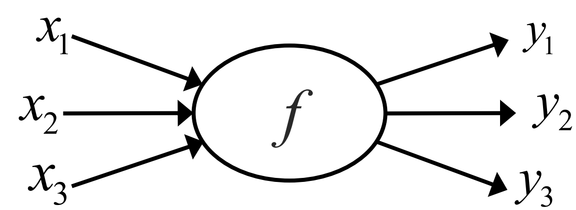


- Adversary can observe memory accesses
  - But what does he "*know*" about secret region?
- Unlimited possible models…
  - We analyze ***weakest*** form of adversary, ***no priors***
  - This still enough to perform a successful attack
- Our adversary:
  - Can only observe application execution for ***reasonable*** (polynomial) amount of time
  - Has only limited (polynomial) storage space
  - Has only limited (polynomial) computational power

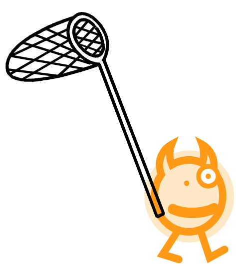
  - Our experiments used one standard x86 server (no farm jobs, etc)

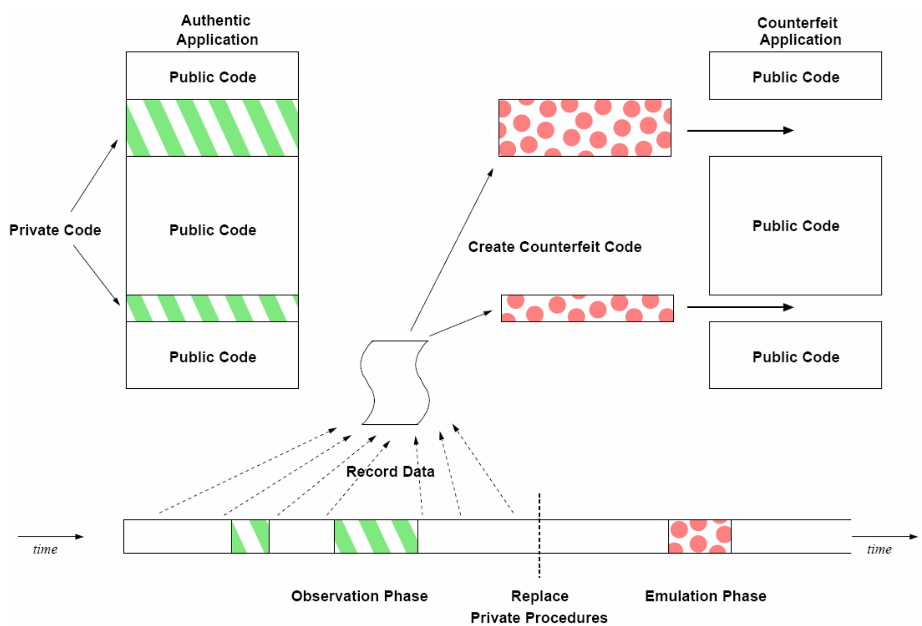
Memoization Attacks

- ***Procedures only a set of input-output mappings***



- ***Observe*** application, remembering inputs and outputs in table
  - Then replace private code and ***emulate***





- However, such a simple table is not enough. . .

Implementing a Memoization Attack

- Two main problems
  - Input self-determination
  - Keeping the "*Interaction Table*" small

- ***Input self-determination***

Private procedure

```
F(a) :
   if (a):
      b ← [Z]
   else:
      b ← [Y]
   return (2*b)
```

Two possible input sets

```
{a = ?, [Z] = ?}
{a = ?, [Y] = ?}
```

Naïve solution too costly

```
{a = ?, [Y] = ?, [Z] = ?}
```

- Emulating procedure requires ***order*** information
  - Temporal Memoization

CSAIL

Temporal Memoization

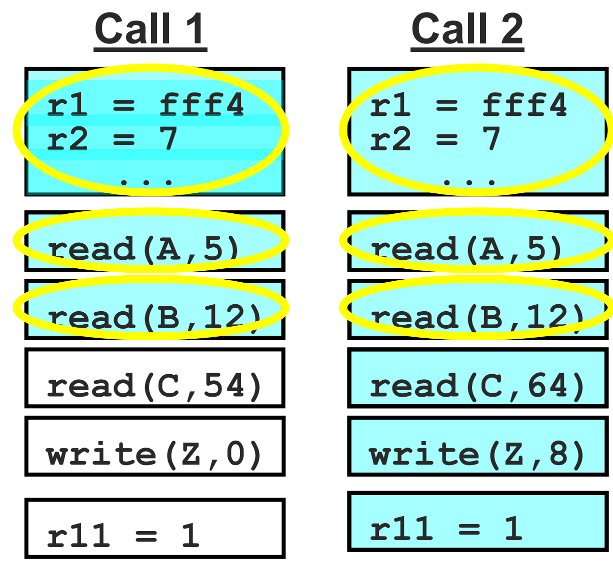
Call 1	Call 2	Call 3	Call 4
r1 = fff4 r2 = 7 ...	r1 = fff4 r2 = 7 ...	r1 = fff4 r2 = 3 ...	r1 = fff4 r2 = 7 ...
read[A]=5	read[A]=5	read[D]=1	read[A]=6
read[B]=12	read[B]=12	read[E]=24	read[B]=30
read[C]=54	read[C]=64	read[F]=20	read[G]=50
write[Z]=0	write[Z]=8	set r11 = 8	write[X]=0
set r11 = 1	set r11 = 1		set r11 = 4

Emulation:

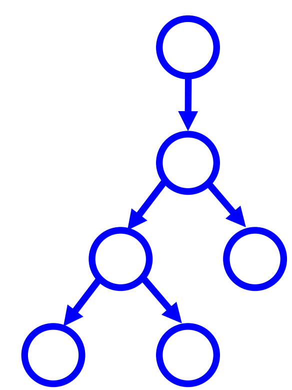
step	1	2	3	4
reads	r1 = fff4 r2 = 7	A = 5	B = 12	C = 64
writes	-	-	-	Z = 8 , r11 = 1

Interaction Table Compression

- ***Keeping the Interaction Table small***
  - Table can become *huge*
  - Contains many redundancies



- Instead of table columns, think of execution trace ***tree***
  - Branches in tree occur on ***reads***
    since they solely determine control flow



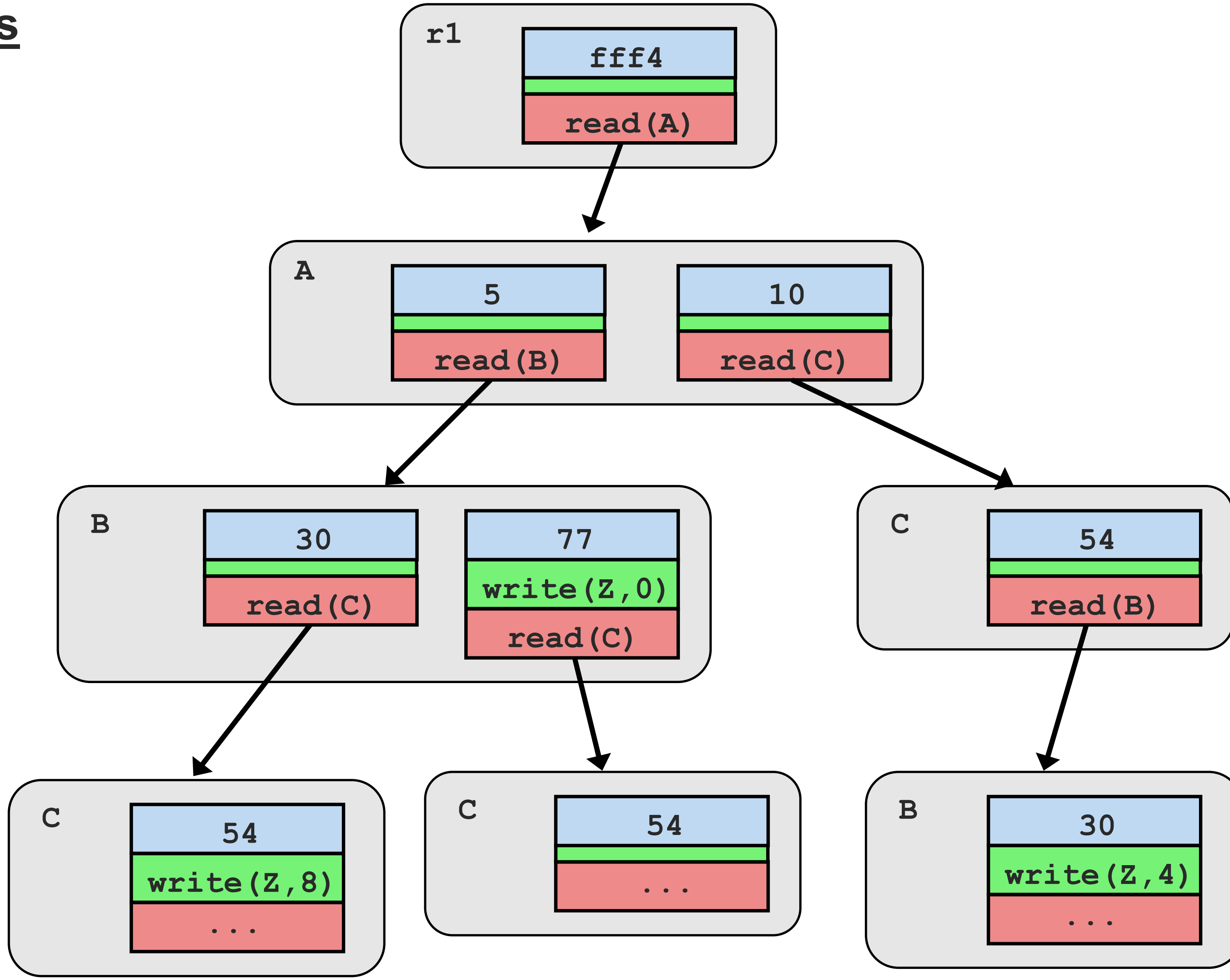
Interaction Tree Construction

Observed Calls

1
```
r1 = fff4
read( A, 5 )
read( B, 30)
read( C, 54)
write( Z, 8)
...
```

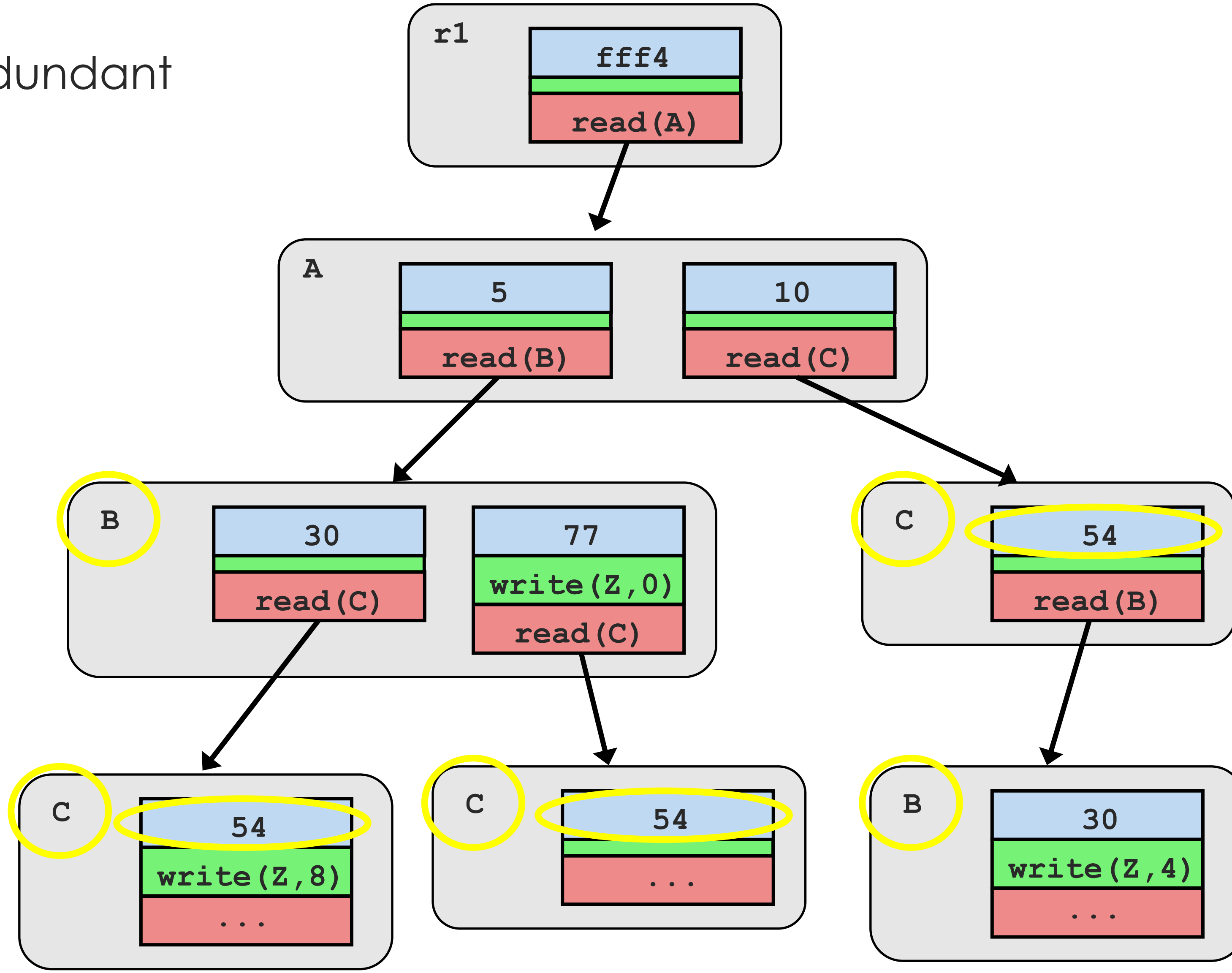
2
```
r1 = fff4
read( A, 10)
read( C, 54)
read( B, 30)
write( Z, 4)
...
```

3
```
r1 = fff4
read( A, 5 )
read( B, 77)
write( Z, 0)
read( C, 54)
...
```



Compressing the Interaction Tree

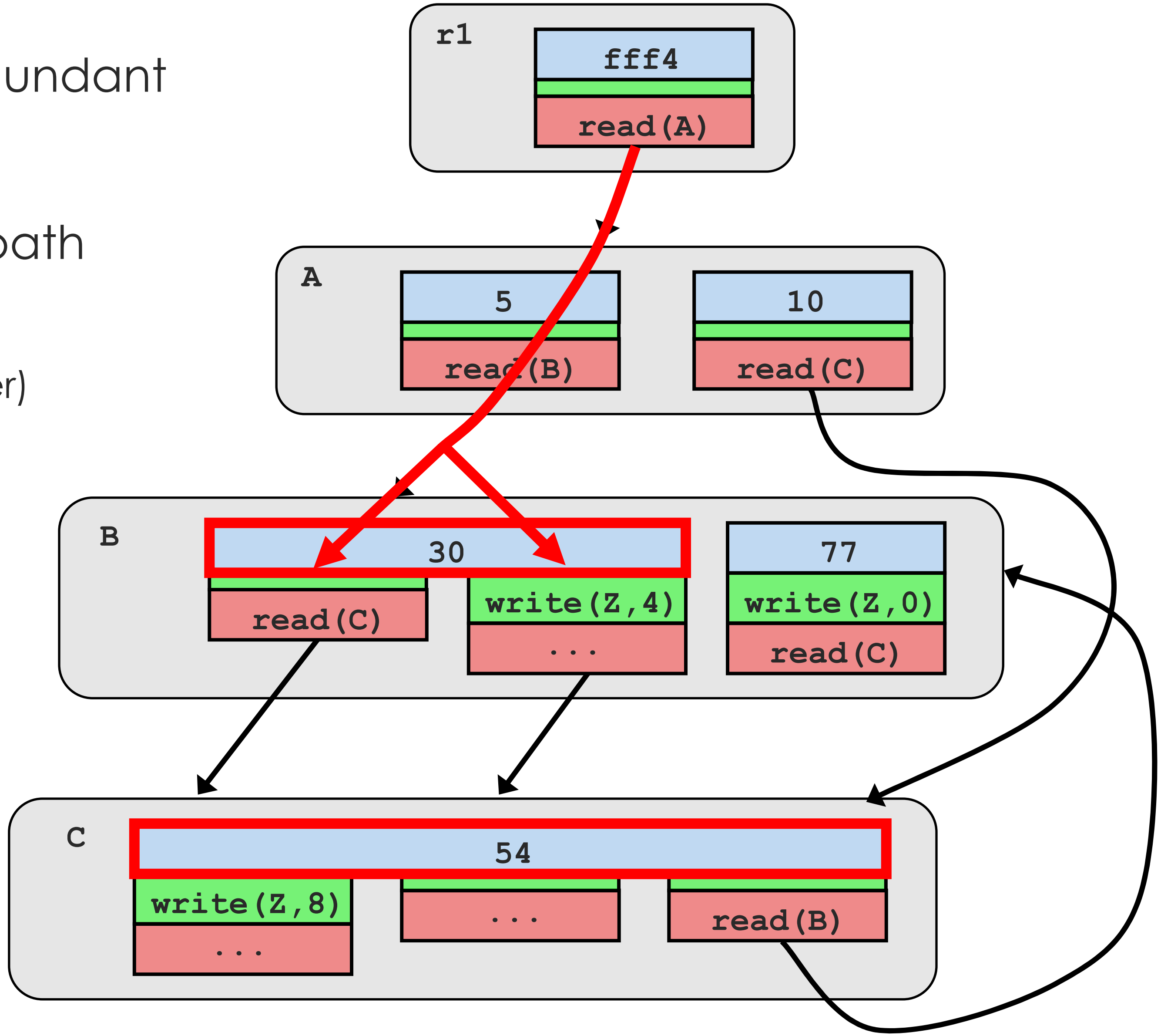
- Tree still redundant



Compressing the Interaction Tree

- Tree still redundant
- Introduce path numbers

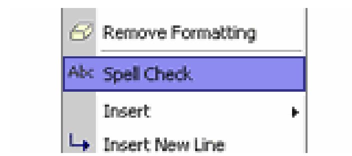
  (more in paper)



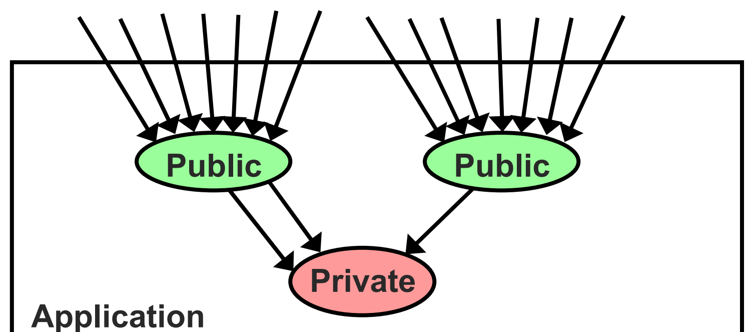
Results of Memoization Attacks

- Memoization Attacks ***can*** work on ***some***, but not ***all*** applications.

- Two "types" effected most (***defined by context***):
  - ***Partially repeated input sets*** (external *workloads*)
    - Repeats ***functionality*** or ***input workload***



  - ***Compositing input sets*** (external *workloads*)
    - If a few input sets to application ***cover*** the input space of single procedure, bounded set of possible inputs
    - If application inputs filtered before reaching private call
    - More ***dangerous*** since ***non-intuitive***



Effectiveness on Repeated Workloads

- SPEC CPU2000 Parser:
  - ***special_command()*** - Memoization Attack always succeeds
    - Repeats same functionality, changes internal settings
  - ***is_equal()*** – Memoization Attack always succeeds
    - Only run over dictionary data (checks for special tokens)
- Size of structures manageable:

Size Metric	**Parser: `special_command()`**	**Parser: `is_equal()`**
Number of tree nodes (compressed)	283	5
Size on disk	26,972 Bytes	2,042,968 Bytes
Maximum depth of expanded tree	743	5

Effectiveness on Composite Workloads

- SPEC CPU2000 Gzip ***bi_reverse()***
  - Called when working on entire dataset (bit manipulation)
  - Memoization Attack successful on 97% of calls
- SPEC CPU2000 Parser ***contains_one()***
  - Called for every new input
  - Memoization Attack successful on 33% of calls

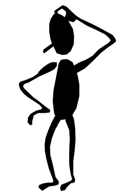
Gzip: `bi_reverse()`
Emulating: `ref.log`

Observed Inputs	Emulatable Calls
`random`	681 / 1797 38%
`random, graphic`	1362 / 1797 76%
`random, graphic, program`	1518 / 1797 84%
`random, graphic, program, source`	1741 / 1797 97%

Parser: `contains_one()`	
Workload: `lgred.in` Emulating: `smred.in`	0 / 71 0%
Workload: `lgred.in` Emulating: `mdred.in`	1136 / 3485 33%

Indicators of Insecurity

- Memoization Attack ***feasible***
  - But can't prove exactly when it will work…



- Which procedures will it work for?
  - Running attack to determine is computationally intensive
  - Instead, use ***indicators*** that give ***suggestion of success***
    - We give two, but many more possible
- Tests show negative results

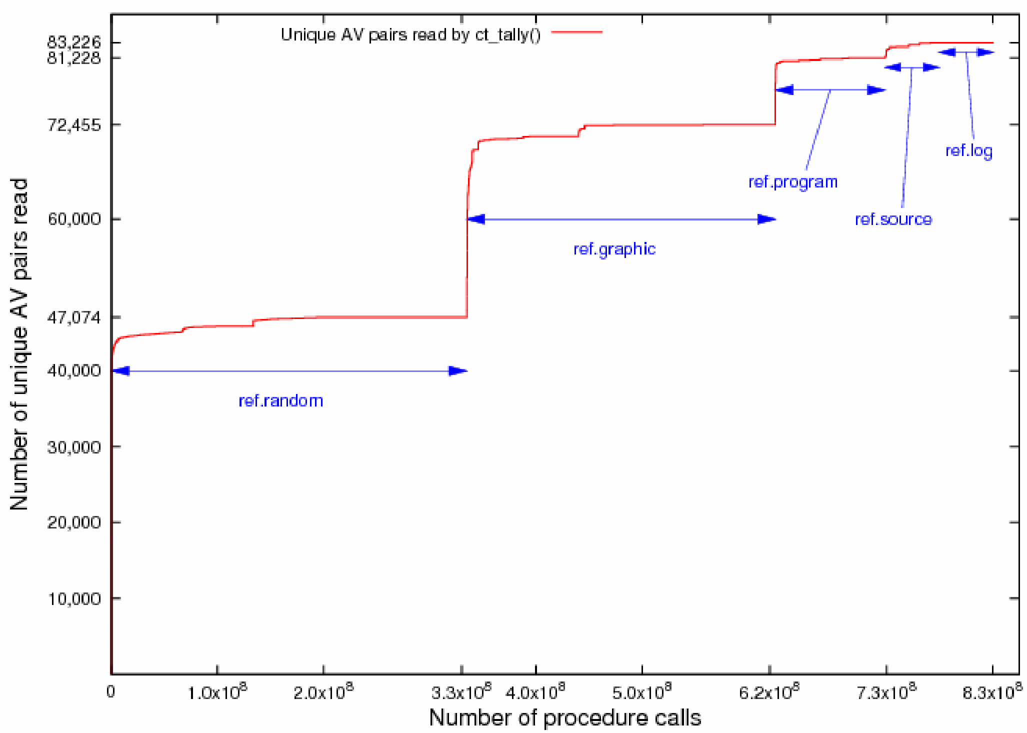
  - Cannot show positive security (especially given heuristics)
- Tests should be
  - computationally ***simple***
  - ***numerous*** and self-supporting

Input Saturation

- ***Count*** unique input values seen by procedure
  - Indicates cost/size of Interaction Tree
- Many ways to estimate input values
  - Our experiment simply counted on few executions
- ***Plot*** or "***Saturation Weight***" describes count

$$SW = \frac{1}{N\omega(N)} \int_0^N \omega(c)dc$$

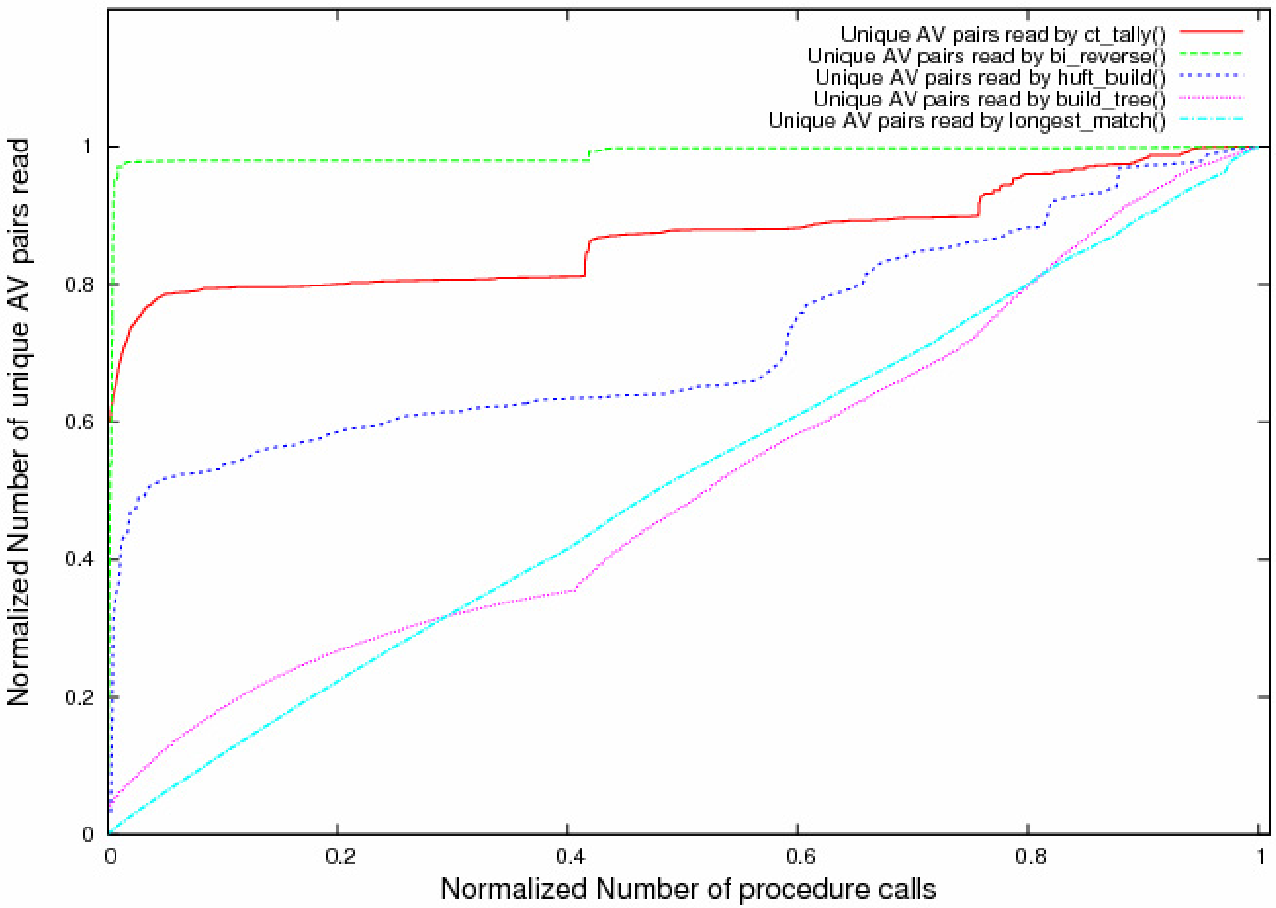
- ***Saturating*** when *SW=1.0*



Results of Input Saturation on Gzip

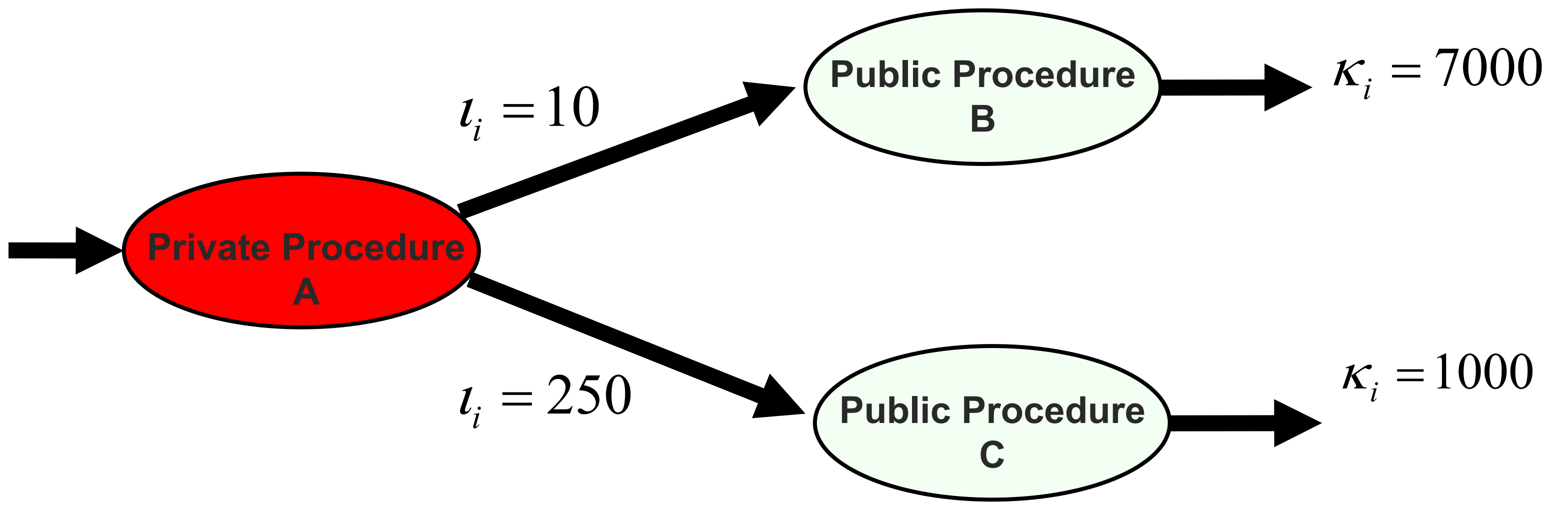
- Some clearly saturate, others clearly do not
  - Some ambiguous ➡ needs more testing

Procedure	SW
bi_reverse	0.99
ct_tally	0.87
huft_build	0.72
build_tree	0.51
longest_match	0.51



Data Egress

- Output possibly more indicative of complexity than input
- Count unique data created by procedure ***and*** data's ***importance*** to rest of program (use for both control & final value)



- Egress Weight: $$\Phi(\eta) = \sum_{\forall(\iota_i,\kappa_i)\in\eta} \frac{\kappa_i}{\iota_i}$$
  - ***higher*** = ***harder*** to attack (compared against other procedures in single app)

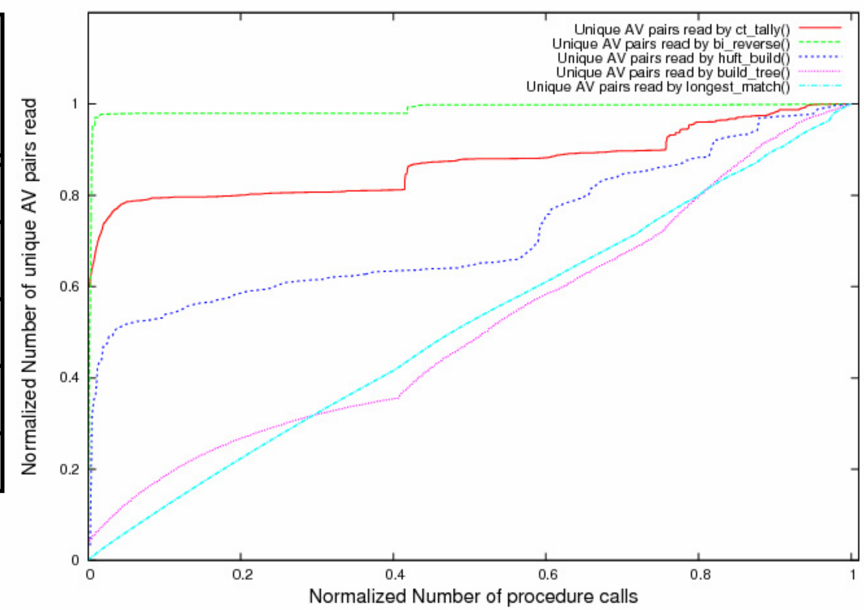
Results of Data Egress on Gzip

- Both high and low Egress Weights
- Inconsistencies and similarities when compared with Saturation Weight
  - ***Lesson:*** Must use multiple metrics
- Real attack: *bi_reverse* almost 100%, *ct_tally* tiny success

Egress Weight

Procedure	Total Unique Writes	Public Readers	Φ weight
bi_reverse	259	2	93
ct_tally	4,214,758	4	1,343,144
huft_build	59,224	4	96
build_tree	21,000	4	2
longest_match	515	1	13,010

Input Saturation



Related Work – Secrecy & Piracy

- Four major areas – By far, incomplete list, showing most related

- Software Secrecy
  - Gosler – Defined problem, deconstructing [1986]
  - Collberg, et al – Obfuscation Transforms [1997,2002]
  - Barak, et al – Obfuscation infeasibility [2001-2005]
  - Kent – Encrypted processor [1981]
  - Lie, Suh, et al – Physical security [2000-2005]

- Software Piracy
  - Collberg, et al – Watermarking [2001-2002]
  - Jakobsson, et al – Renewability [2002]
  - Microsoft, others – Online verification [recent]
  - Lie, TCG, NGSCB – Tie code to physical CPU [2000-present]

Related Work – Partitioning & Complexity

- Program Partitioning
  - Yee – Partitioning for secure coprocessors [1994]
  - White, et al – ABYSS, separations for security [1990]
  - Zhang, et al – Program slicing for piracy [2003]
  - Brumley, et al – Privtrans, monitor/slave separation [2004]
  - Zdancewic, et al – For end-to-end information flow [2002]
  - Ori Dvir, et al – Remote memory allocation [2005]

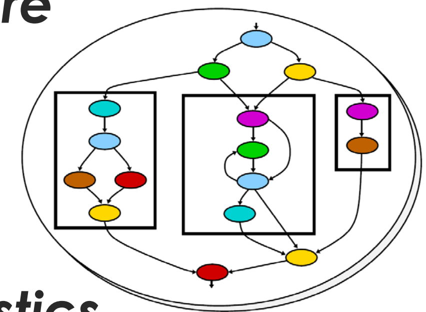
- Application Complexity
  - McCabe –
  - Kent –
  - Harrison, et al –
  - Henry, et al –
  - Munson, et al –

  } Software engineering metrics [1976-1994]

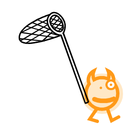
  - Yang, et al – Metrics for difficulty to deconstruct [1997]

Conclusions

- ***Partitioned Applications are not automatically "secure"***
  - Secret code can be reconstructed



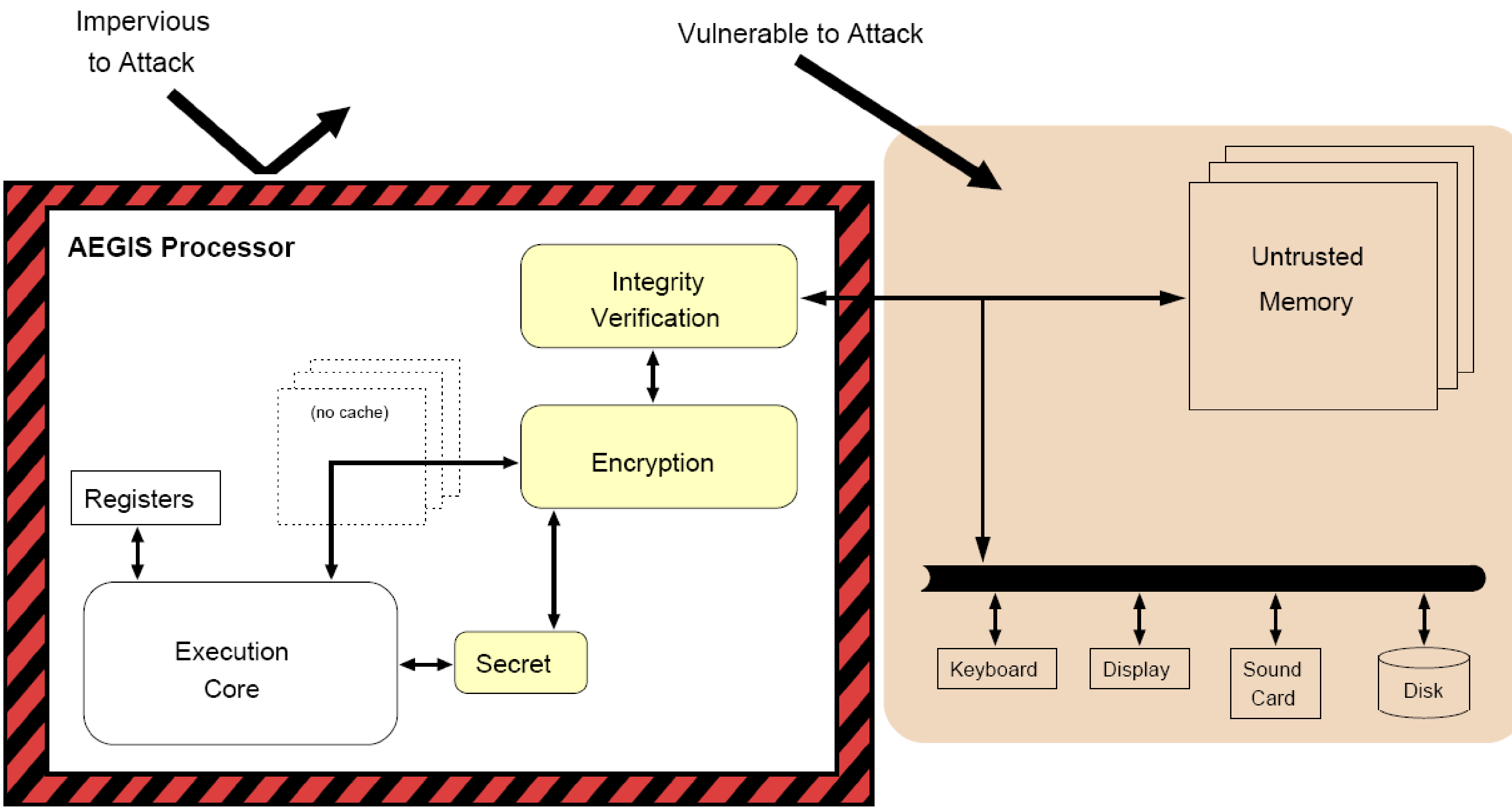
- ***Memoization Attacks*** are feasible and non-trivial
  - Even when using a weak adversary with ***no heuristics***
    - Although they cannot ***always*** succeed
  - Can be implemented and performed on a regular computer
  - Repeated Workloads very easily emulated
  - Composite Workloads also can be emulated



- Simple ***tests indicate*** when Memoization Attacks might succeed
  - Easier to perform than full attack
  - But, not a guarantee (use many tests)
  - Can aid software designer

Extra Slides

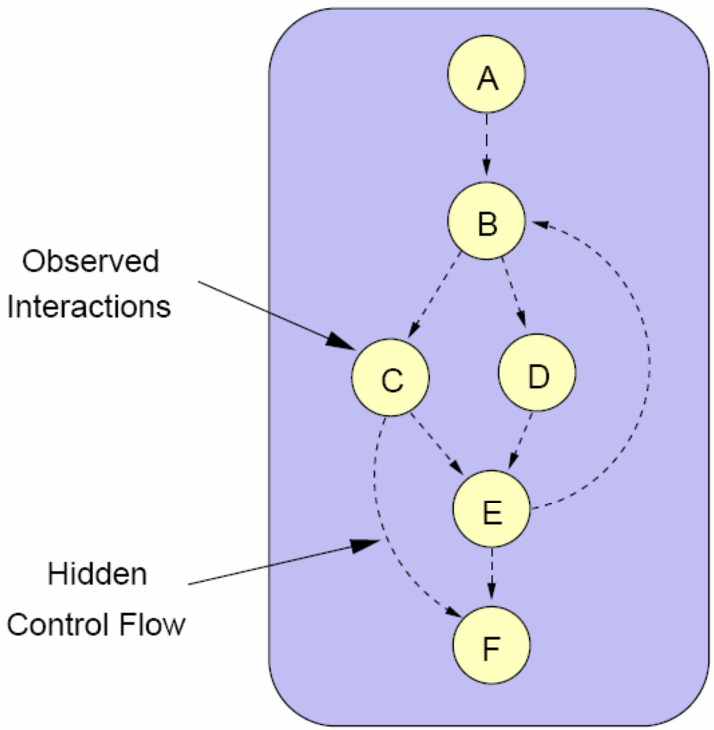
AEGIS Model



Impervious
to Attack
Vulnerable to Attack
AEGIS Processor
Integrity
Verification
(no cache)
Encryption
Registers
Execution
Core
Secret
Untrusted
Memory
Keyboard
Display
Sound
Card
Disk

Tree from Hidden Control Flow Graph

Private Procedure



Observed Sequences

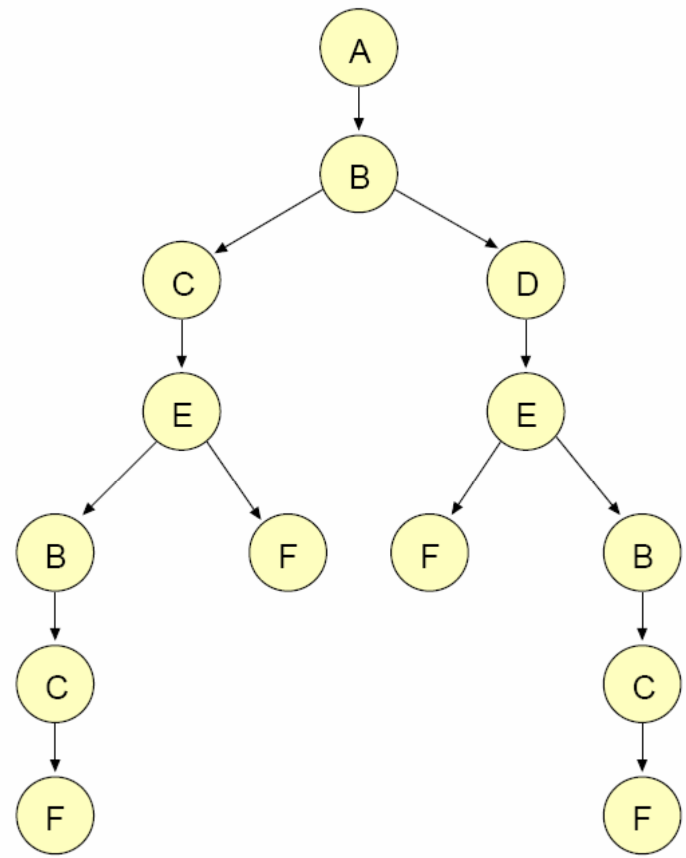
{ A, B, C, E, B, C, F }

{ A, B, D, E, F }

{ A, B, D, E, B, C, F }

{ A, B, C, E, F }

Interaction Tree



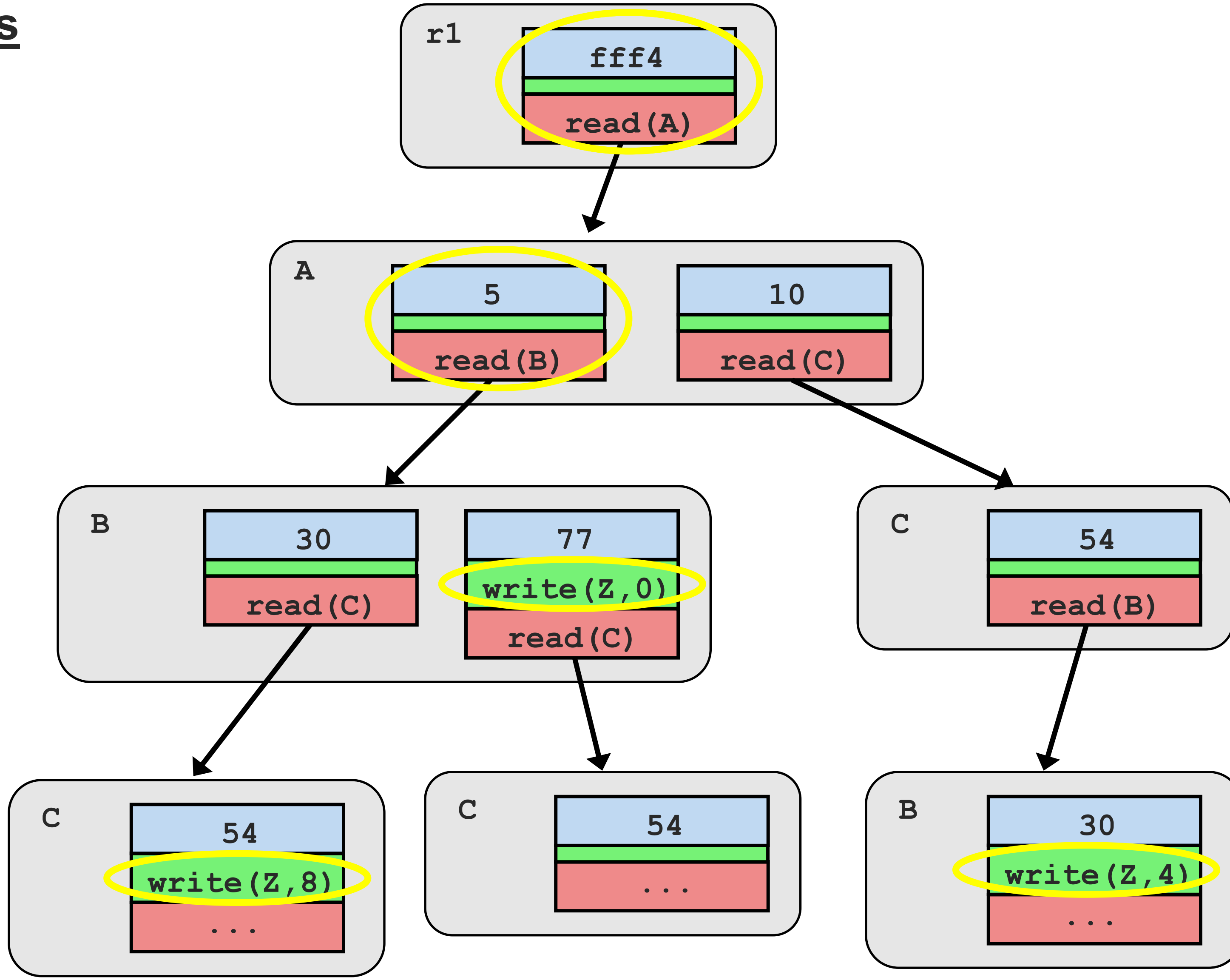
Interaction Tree Construction Steps

Observed Calls

1
```
r1 = fff4
read( A, 5 )
read( B, 30)
read( C, 54)
write( Z, 8)
...
```

2
```
r1 = fff4
read( A, 10)
read( C, 54)
read( B, 30)
write( Z, 4)
...
```

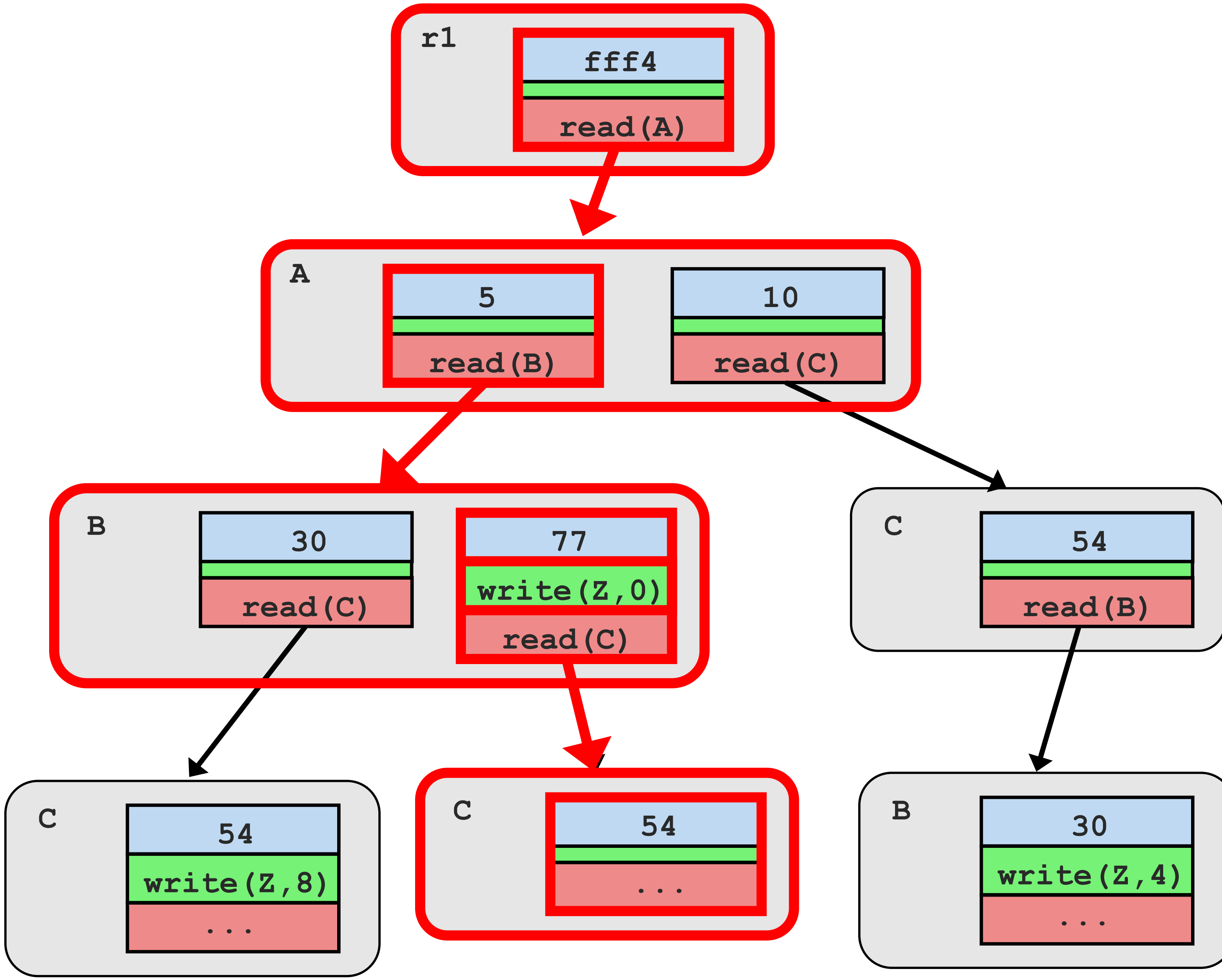
3
```
r1 = fff4
read( A, 5 )
read( B, 77)
write( Z, 0)
read( C, 54)
...
```



Emulating with Interaction Tree

Emulation:

```
r1 = fff4
 A = 5
 B = 77
write(Z, 0)
 C = 54
 ...
```



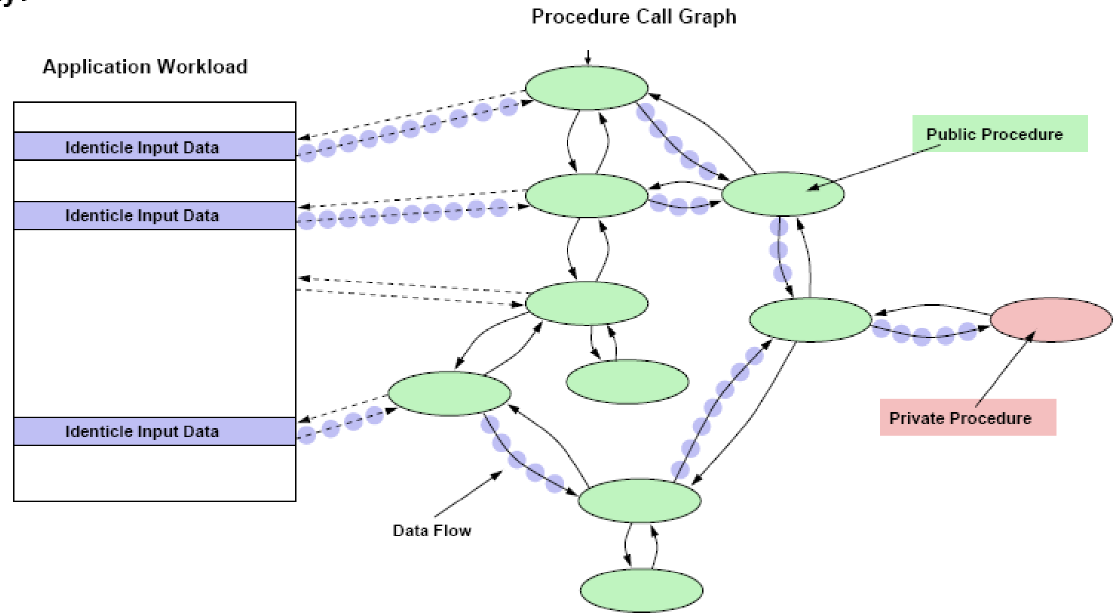
Interaction Table Path Numbers

- Path numbers enable joins and loops in Interaction Tree
- Each path number refers to unique branch of un-compressed tree
- Nodes in Interaction Table can contain multiple path numbers

Address	Read Value	Write AV Pairs	Path Numbers	Next Address
r1	0xfff4	-	$\{0 \rightarrow 1\}$	r3
	0xffc0	-	$\{0 \rightarrow 2\}$	r3
r3	0x7	(0x4410, 0x1e)	$\{1\}$	0x4072
	0x7	(0x4420, 0x60) (0x4424, 0x0)	$\{2\}$	0x4104
	0x3	-	$\{1 \rightarrow 4\}$	0x4100
	0x3	(0x4420, 0x5c)	$\{2 \rightarrow 5\}$	0x4100
0x4072	0x1	-	$\{1, \ldots\}$	0x4100
	0x2	-	$\{1 \rightarrow 3, \ldots\}$	0x4100
0x4100	0x20	-	$\{5, \ldots\}$	0x4088
⋮	⋮	⋮	⋮	⋮

Repeated/Composite Workloads

Repeated Functionality:



Multiple Workloads: