Testudo: Heavyweight Security Analysis via Statistical Sampling

Joseph L. Greathouse

Ilya Wagner | David A. Ramos | Gautam Bhatnagar

Todd Austin | Valeria Bertacco | Seth Pettie

Advanced Computer Architecture Laboratory

University of Michigan

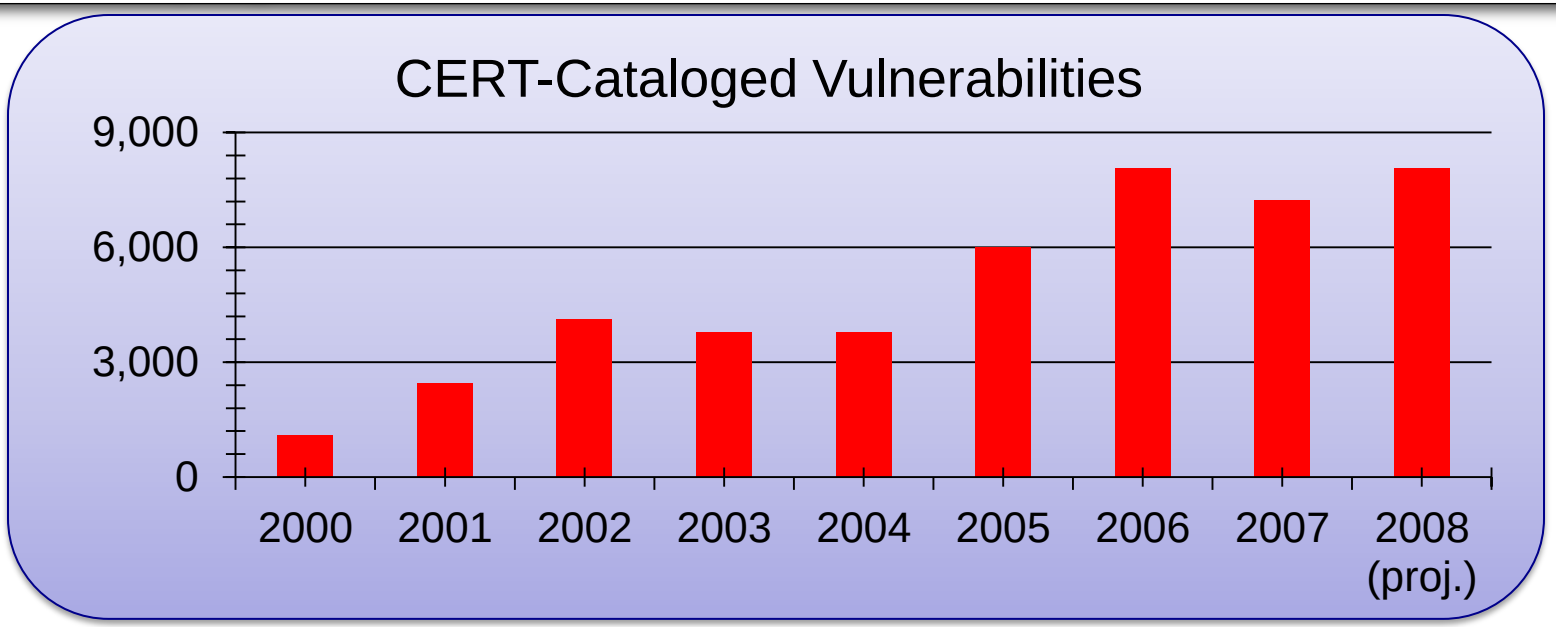
UNIVERSITY OF MICHIGAN

MICRO, Lake Como, Italy
November 11, 2008

Bad Software is Everywhere

- NIST: SW errors cost U.S. $60 billion/year as of 2002
- These errors include security vulnerabilities.

"Security bugs are out there, in fact in web apps they're pretty much a plague." - Zeev Suraski, co-creator of PHP



Bad Software is Everywhere

- NIST: SW errors cost U.S. $60 billion/year as of 2002
- These errors include security vulnerabilities.

"Security bugs are out there, in fact in web apps they're pretty much a plague." - Zeev Suraski, co-creator of PHP

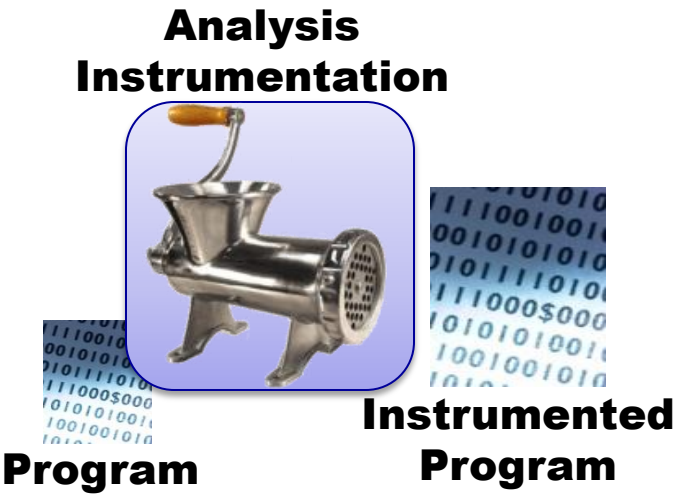
Software Dynamic Analysis for Security

- Valgrind, Rational Purify, DynInst
  - + Multiple types of tests, runtime protection
  - – Extremely high runtime overheads

Analysis Instrumentation

Software Dynamic Analysis for Security

- Valgrind, Rational Purify, DynInst
  - + Multiple types of tests, runtime protection
  - – Extremely high runtime overheads



Software Dynamic Analysis for Security

- Valgrind, Rational Purify, DynInst
  - + Multiple types of tests, runtime protection
  - – Extremely high runtime overheads

Developer

Analysis Instrumentation

LONG run time

In-House Test Server(s)

Software Dynamic Analysis for Security

- Valgrind, Rational Purify, DynInst
  - + Multiple types of tests, runtime protection
  - – Extremely high runtime overheads

Developer

Analysis Results

Analysis Instrumentation

In-House Test Server(s)

Hardware Dynamic Analysis for Security

- DIFT, Raksha, FlexiTaint, et al.
  - \+ Low/no runtime overhead, runtime protection
  - – Limited analysis types, complex HW overhead

Developer

In-House Test Server(s)

Hardware Dynamic Analysis for Security

- DIFT, Raksha, FlexiTaint, et al.
  - \+ Low/no runtime overhead, runtime protection
  - – Limited analysis types, complex HW overhead

Developer

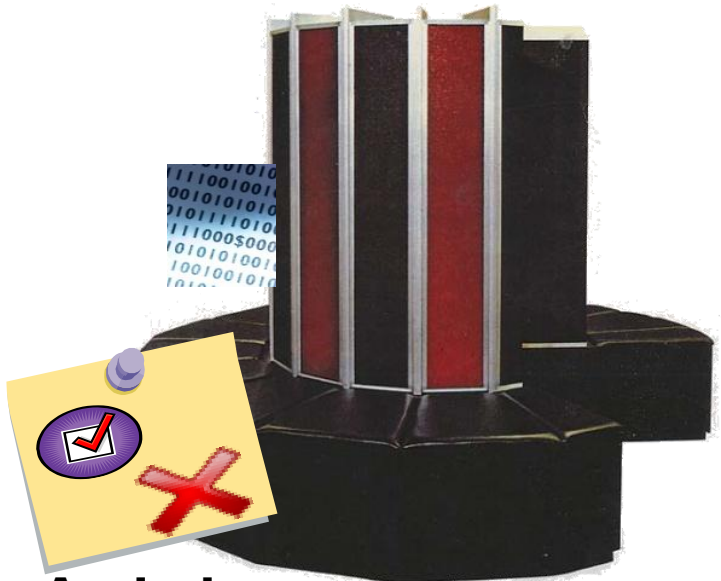
Program

Hardware-Instrumented Test Server(s)

Hardware Dynamic Analysis for Security

- DIFT, Raksha, FlexiTaint, et al.
  - \+ Low/no runtime overhead, runtime protection
  - – Limited analysis types, complex HW overhead

Developer



Analysis Results

Hardware-Instrumented Test Server(s)

Hardware Dynamic Analysis for Security

- DIFT, Raksha, FlexiTaint, et al.
  - \+ Low/no runtime overhead, runtime protection
  - – Limited analysis types, complex HW overhead

Developer

Analysis Results

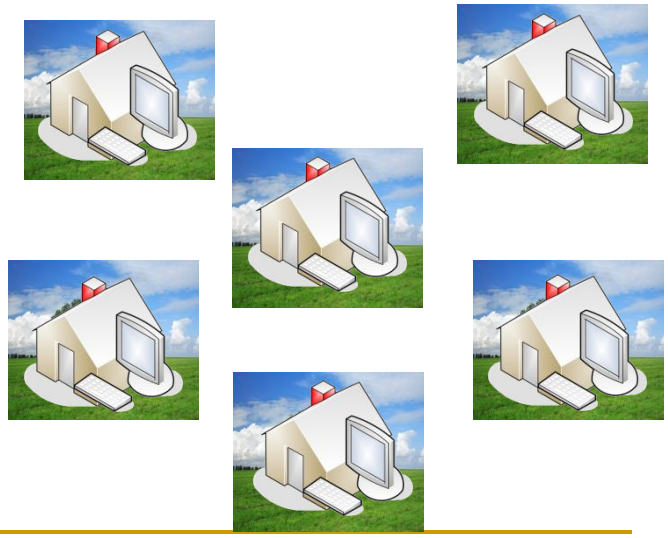
Hardware-Instrumented Test Server(s)

Testudo: Distributed Dynamic Analysis

- Split analysis across population of users
  - + Low HW cost, low runtime overhead, runtime information from the field
  - – Analysis only

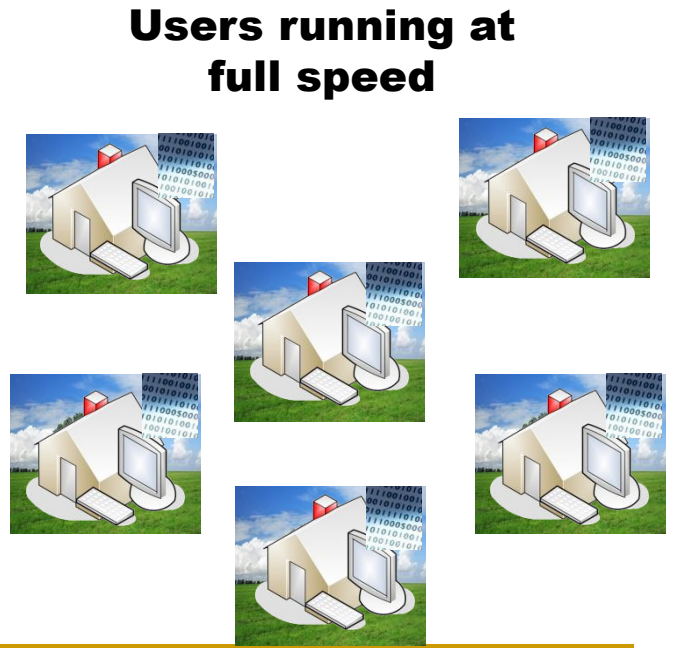
Developer

Instrumented Program



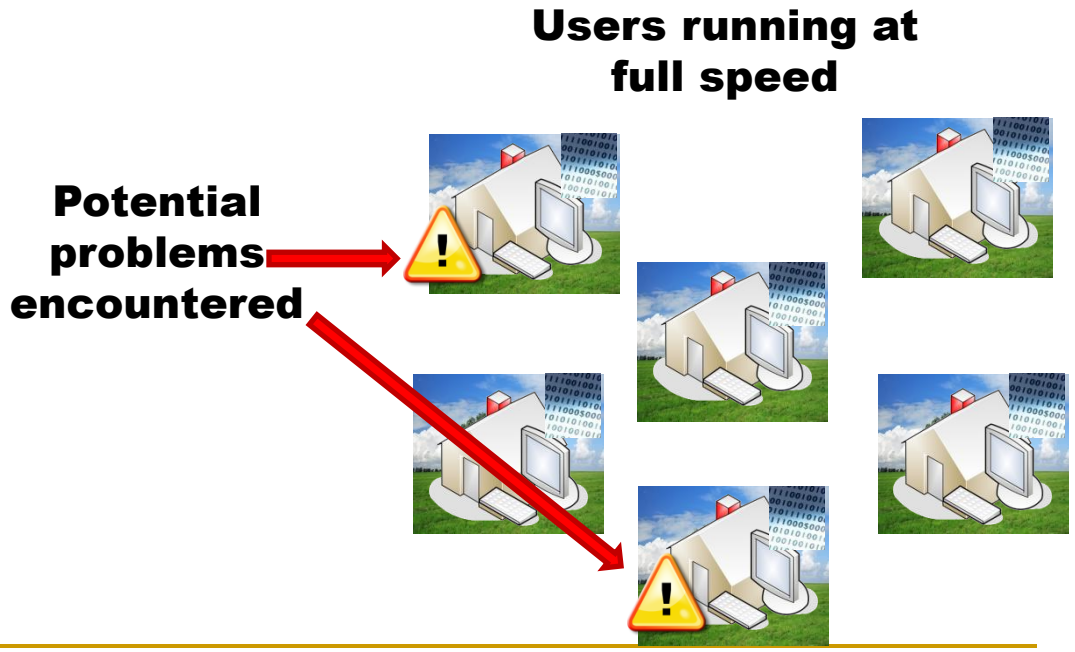
Testudo: Distributed Dynamic Analysis

- Split analysis across population of users
  - + Low HW cost, low runtime overhead, runtime information from the field
  - – Analysis only



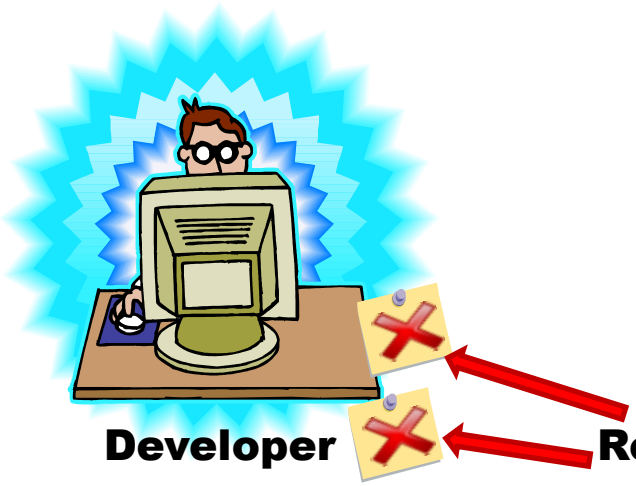
Testudo: Distributed Dynamic Analysis

- Split analysis across population of users
  - + Low HW cost, low runtime overhead, runtime information from the field
  - – Analysis only



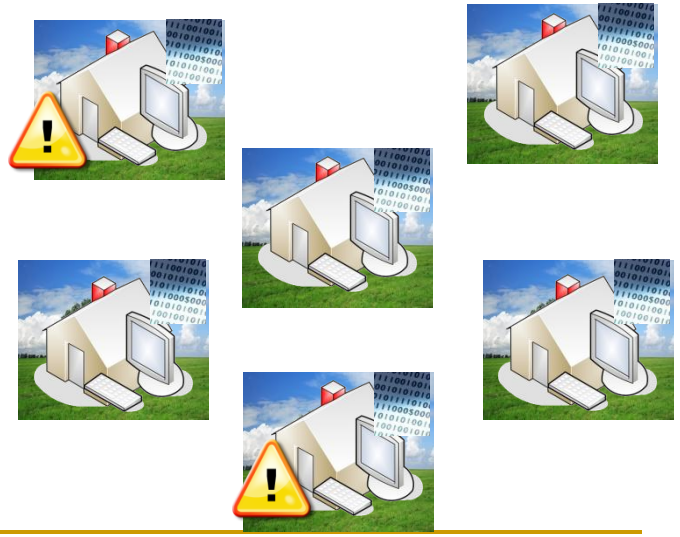
Testudo: Distributed Dynamic Analysis

- Split analysis across population of users
  - + Low HW cost, low runtime overhead, runtime information from the field
  - – Analysis only



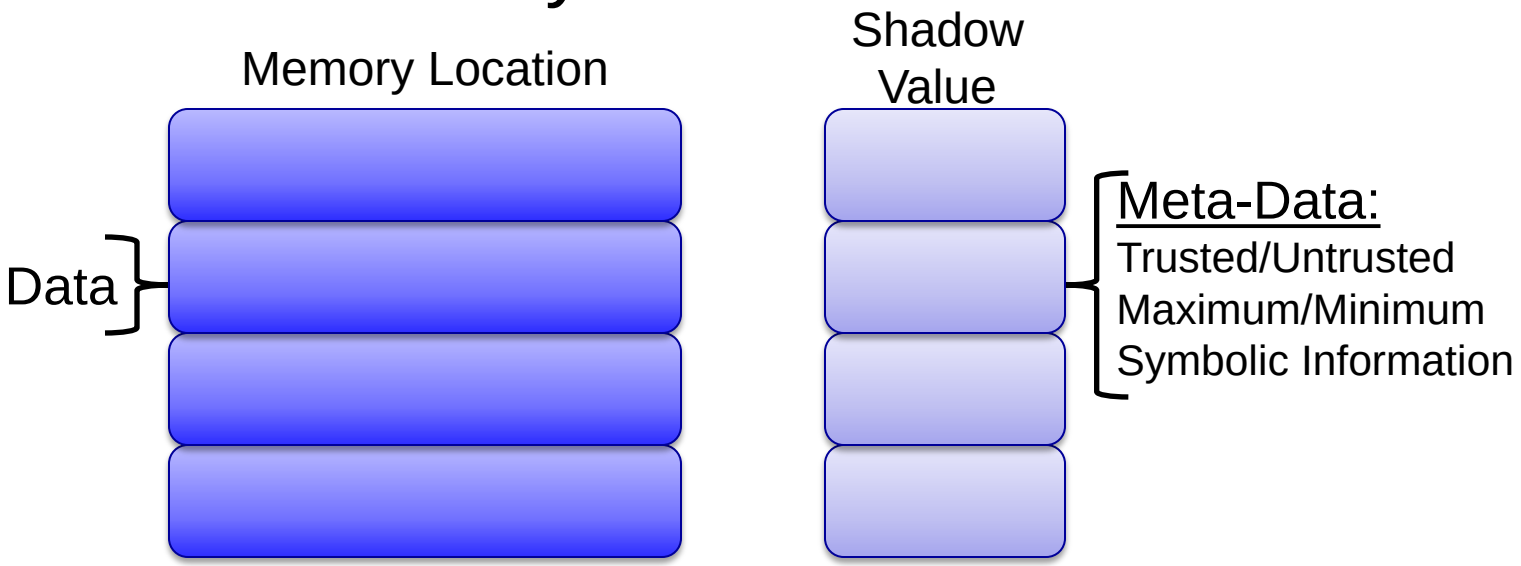
Reported before they are exploited

Users running at full speed



Heavyweight Dynamic Analysis

- Heavyweight analyses use *shadow values.*
- Shadow values hold meta-information about associated memory values



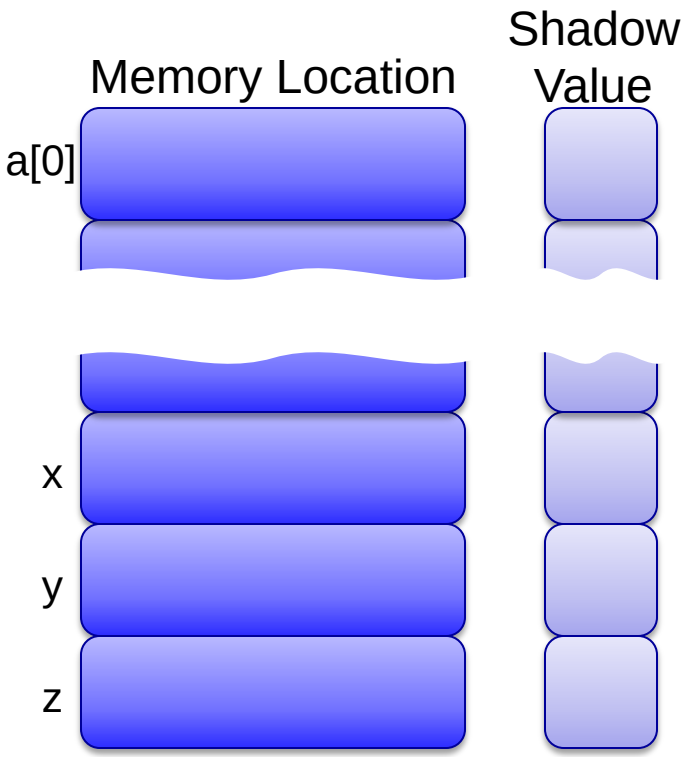
- Can be used to detect potential errors *without an active exploit*.

Example of Heavyweight Analysis

Code:

```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Memory:



Dataflow:

I/O

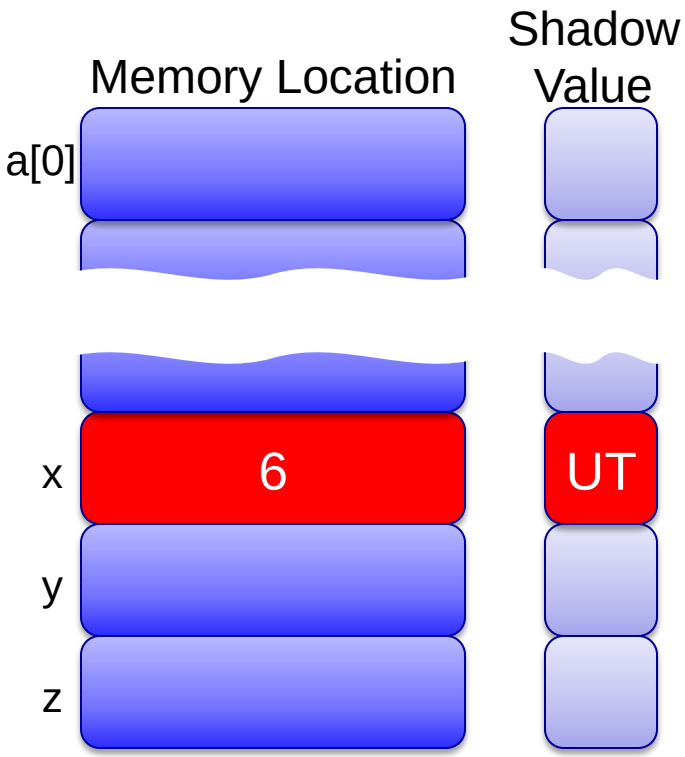
Key:
= has shadow value

Example of Heavyweight Analysis

Code:

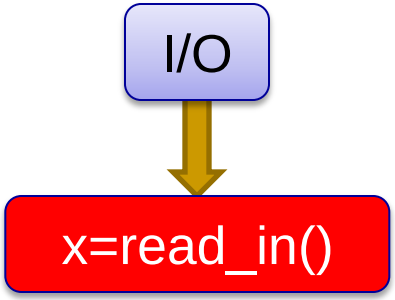
```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Memory:



Dataflow:

Key:
= has shadow value

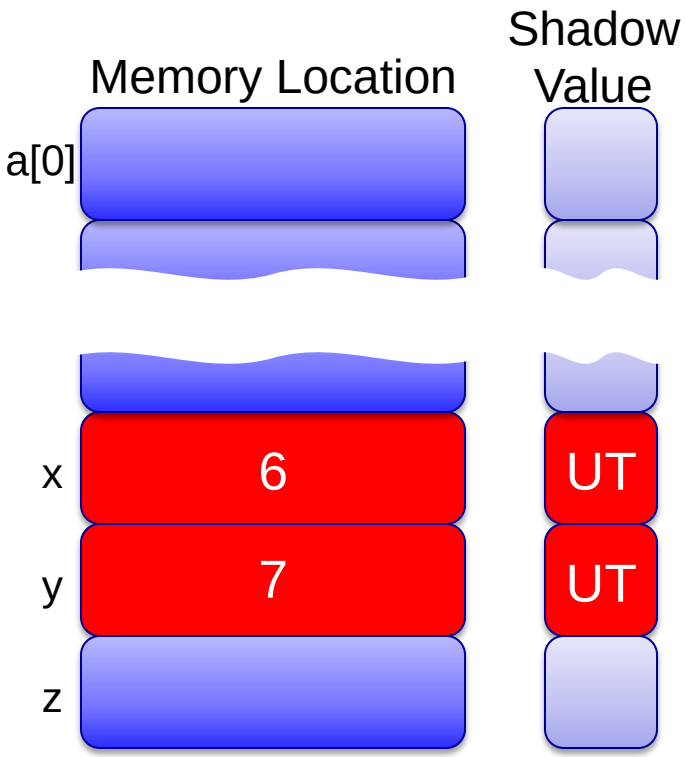


Example of Heavyweight Analysis

Code:

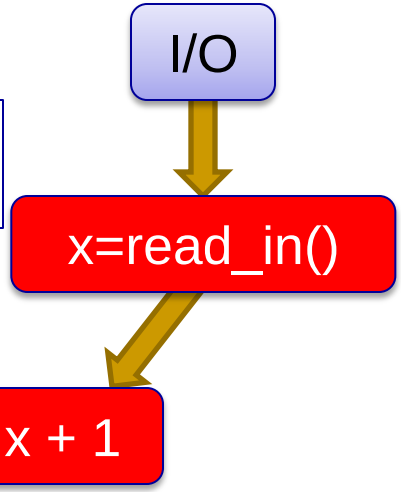
```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Memory:



Dataflow:

Key:
= has shadow value

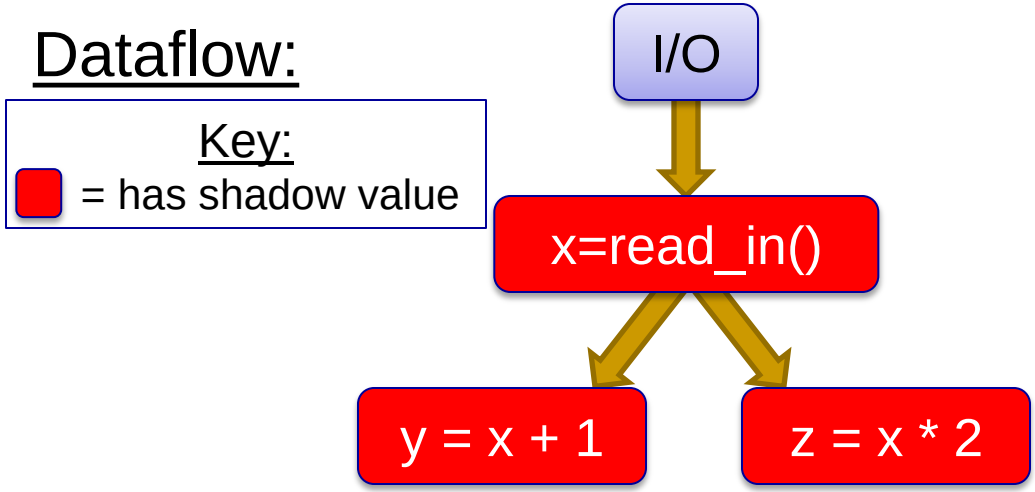


Example of Heavyweight Analysis

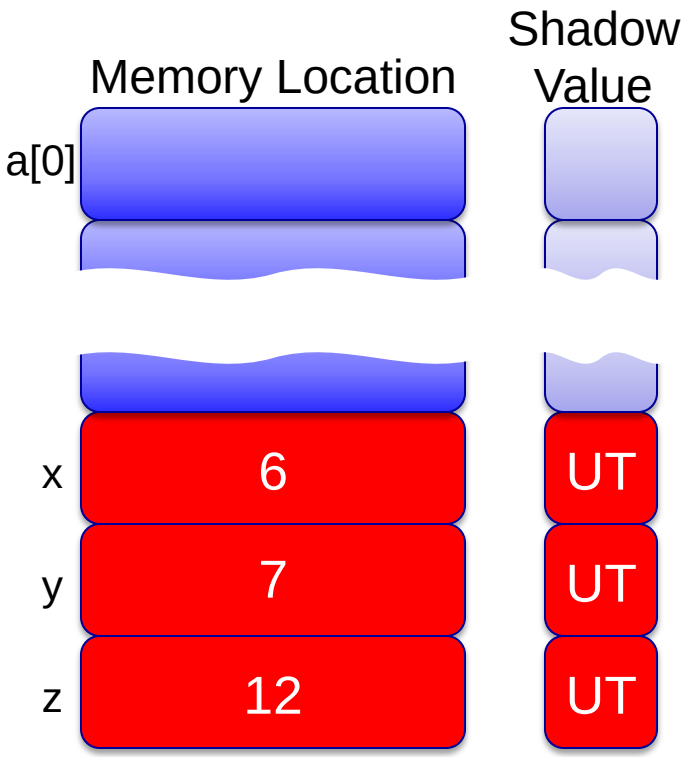
Code:

```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Dataflow:



Memory:

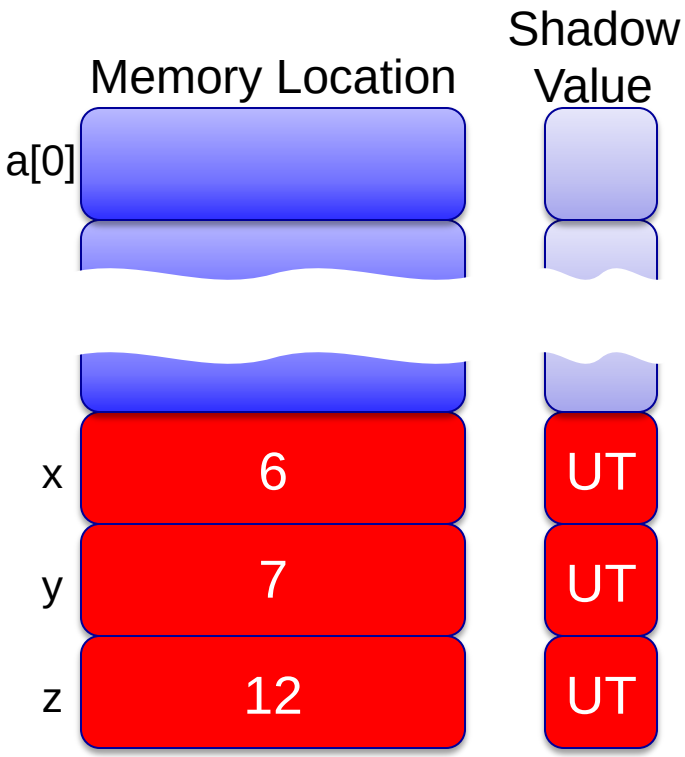


Example of Heavyweight Analysis

Code:

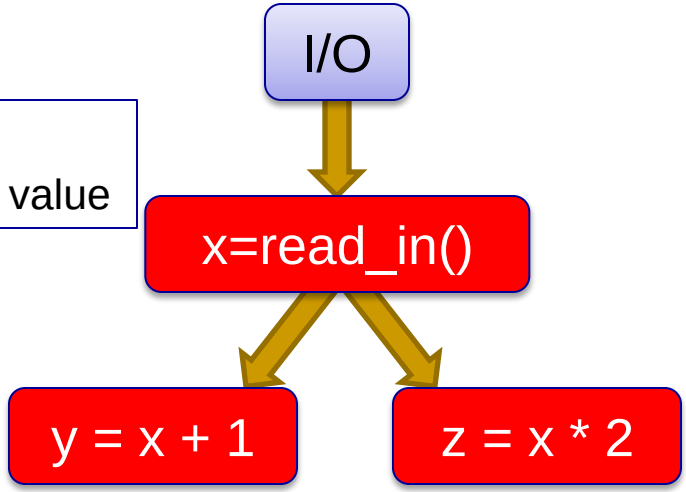
```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Memory:



Dataflow:

Key:
= has shadow value

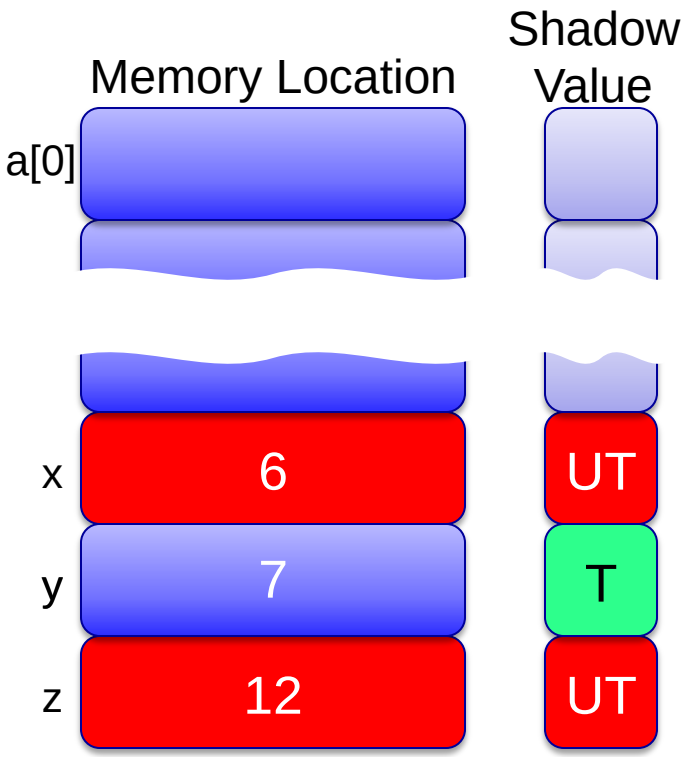


Example of Heavyweight Analysis

Code:

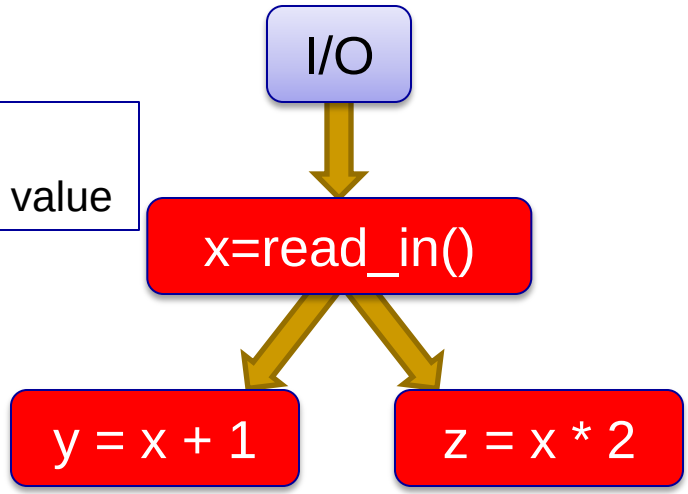
```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Memory:



Dataflow:

Key:
= has shadow value

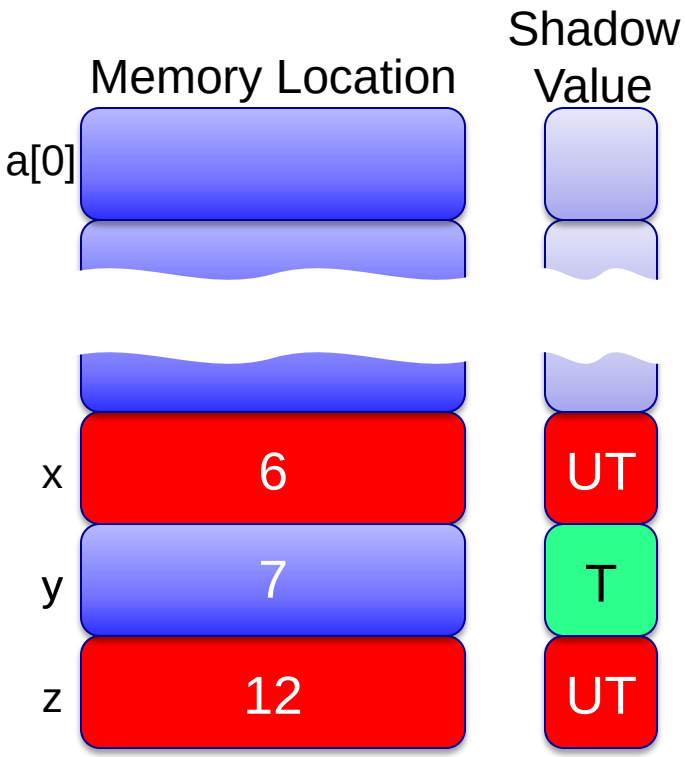


Example of Heavyweight Analysis

Code:

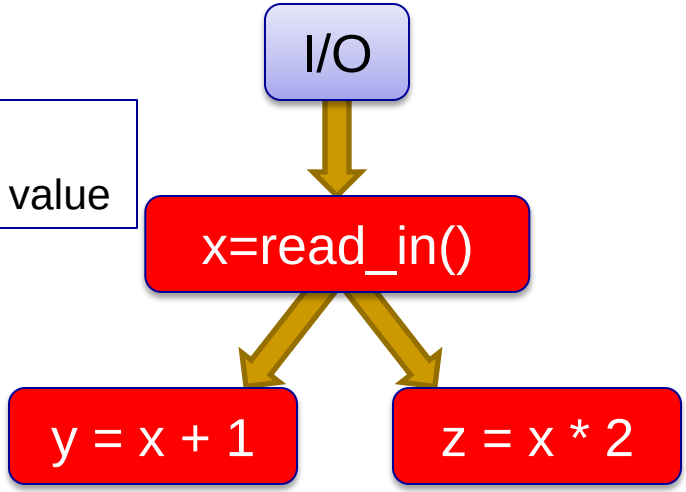
```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Memory:



Dataflow:

Key:
= has shadow value



Contributions of Testudo

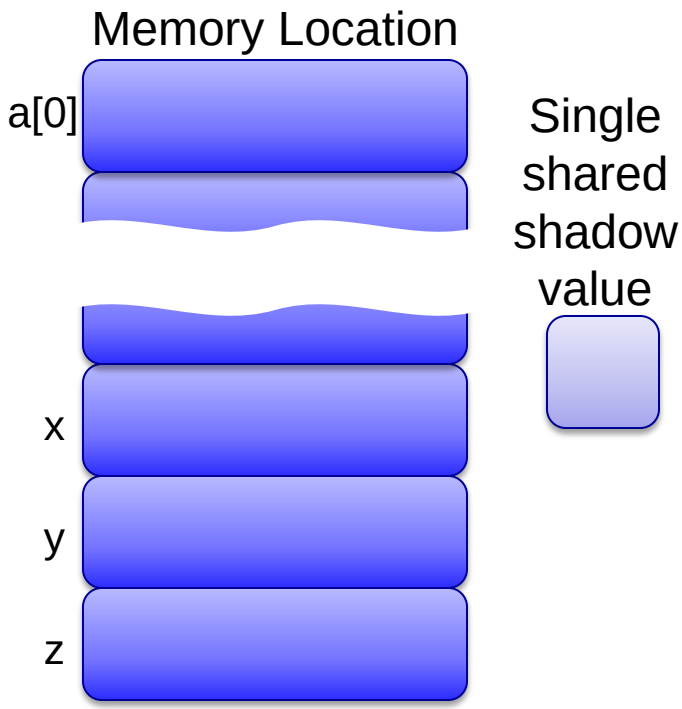
- **Reduce hardware complexity**: Shadow storage is a small, constant size. No out-of-core changes.
- **Reduce runtime overhead:** Divide work across users to reduce overhead for each individual.
- **Increase analysis quality:** Large user population allows analysis of large, varying state spaces.

Dataflow Sampling Example: 1st User

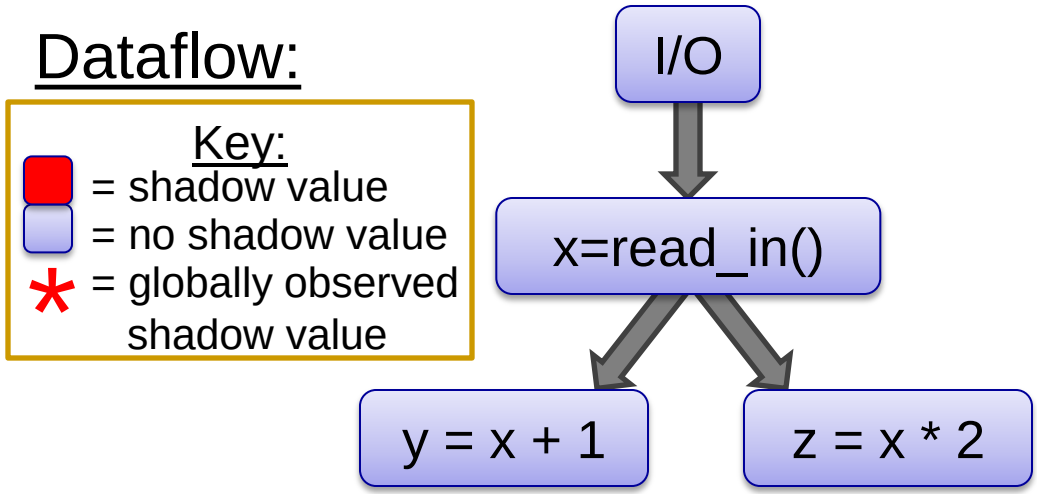
Code:

```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Memory:



Dataflow:

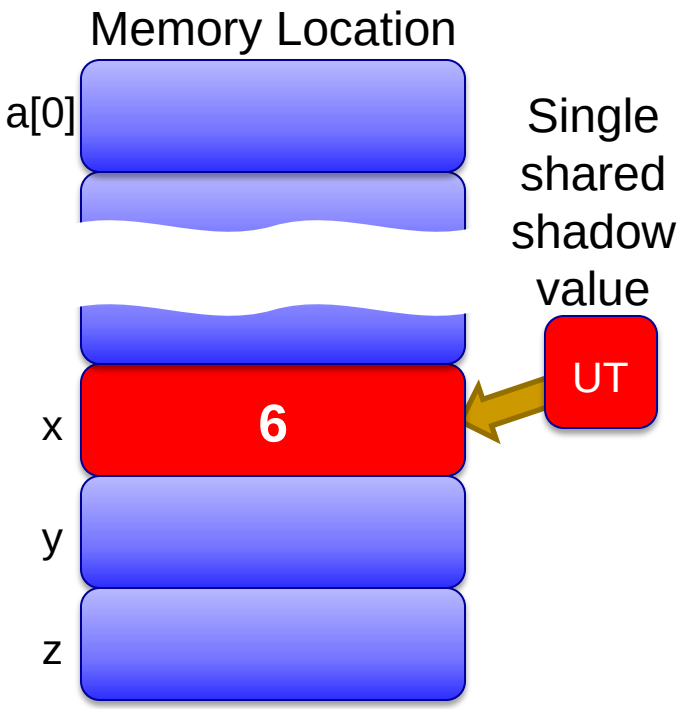


Dataflow Sampling Example: 1st User

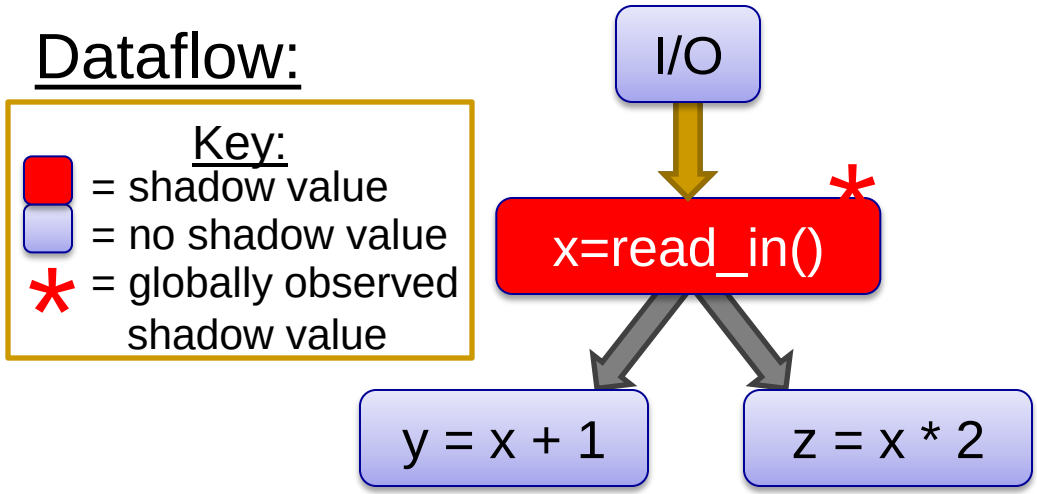
Code:

```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Memory:



Dataflow:



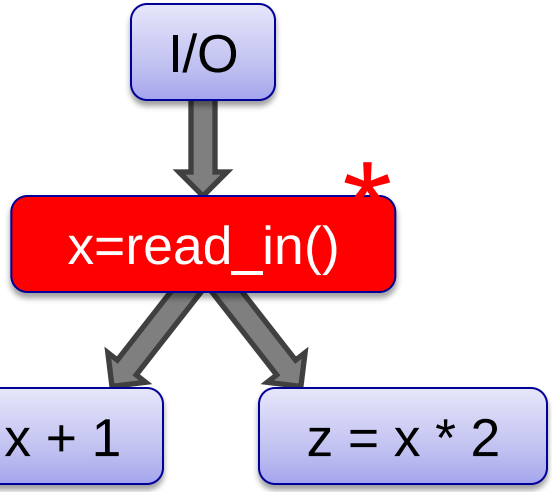
Dataflow Sampling Example: 1st User

Code:

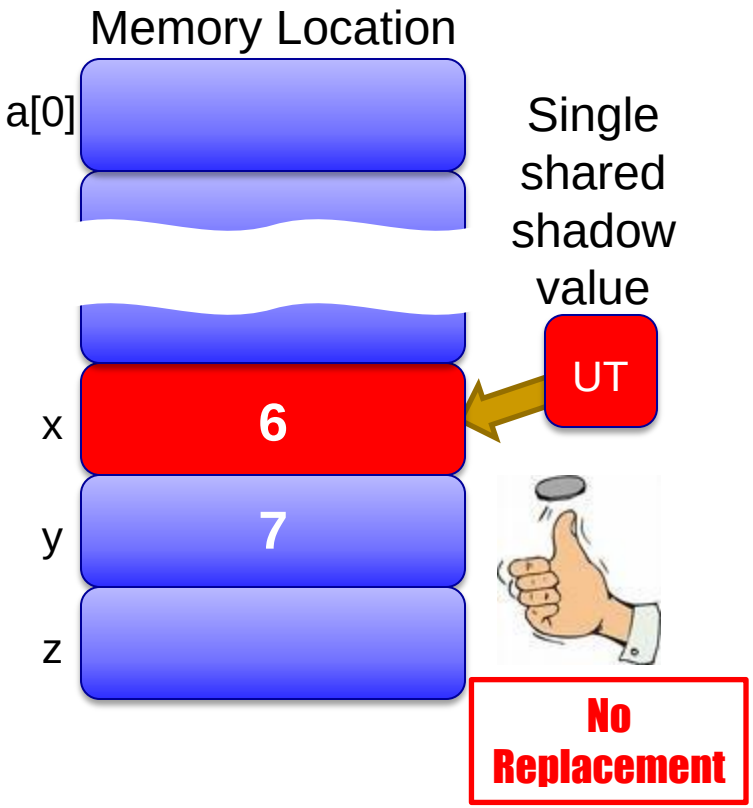
```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Dataflow:

Key:
- = shadow value
- = no shadow value
- * = globally observed shadow value



Memory:



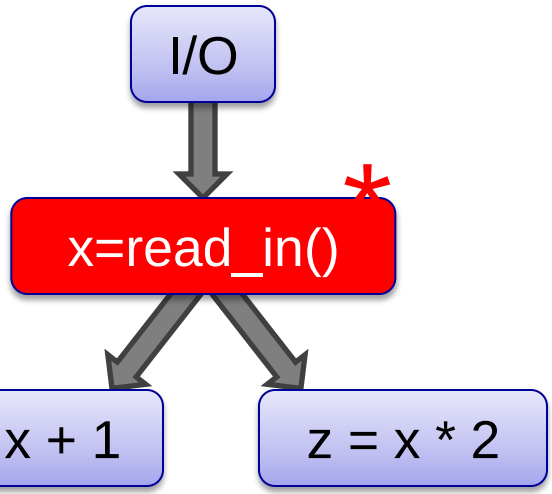
Dataflow Sampling Example: 1st User

Code:

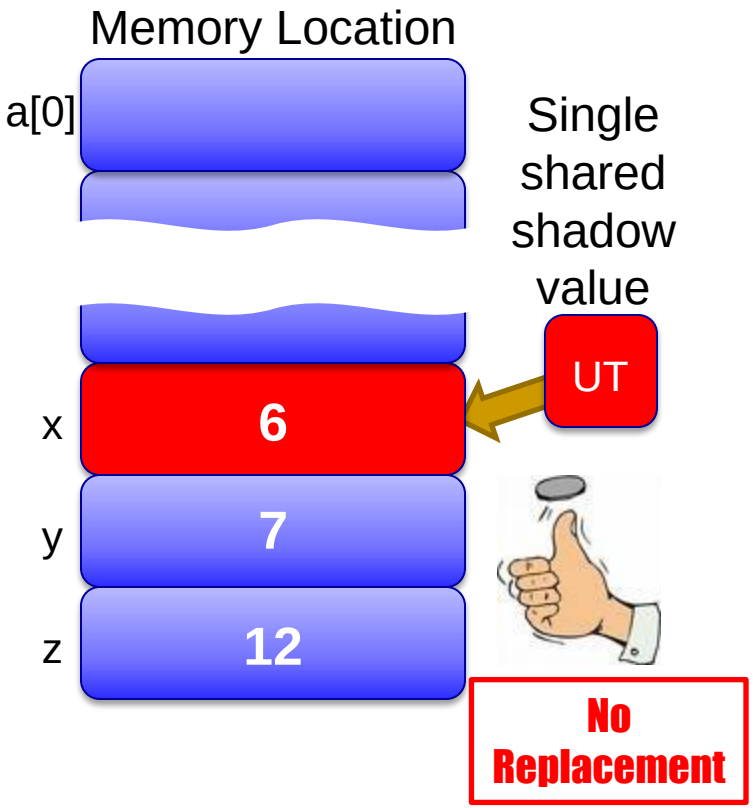
```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Dataflow:

Key:
- = shadow value
- = no shadow value
- * = globally observed shadow value



Memory:

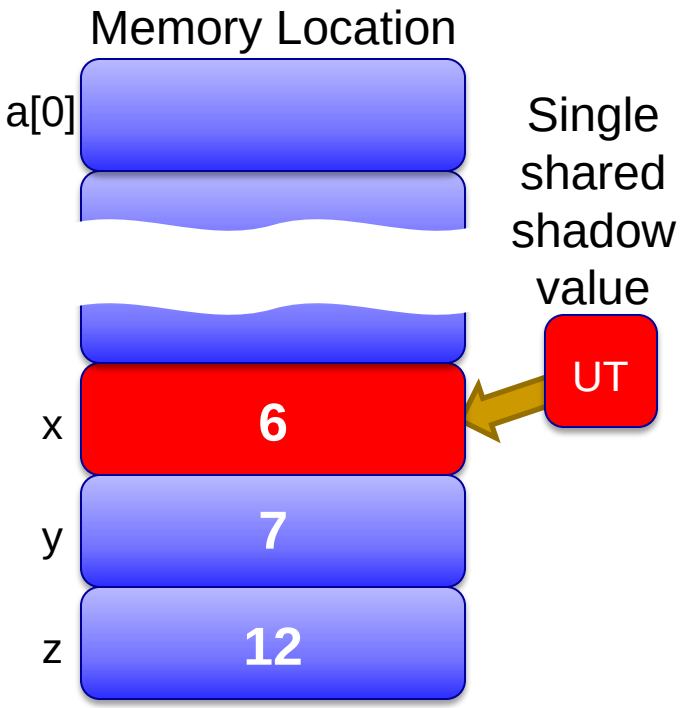


Dataflow Sampling Example: 1st User

Code:

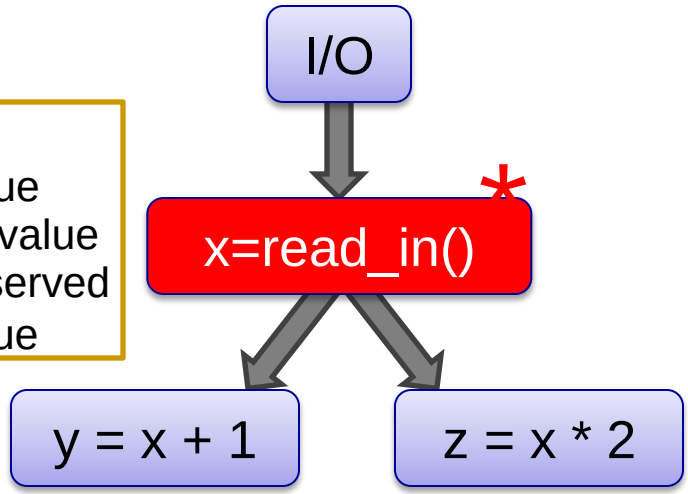
```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Memory:



Dataflow:

Key:
= shadow value
= no shadow value
* = globally observed shadow value

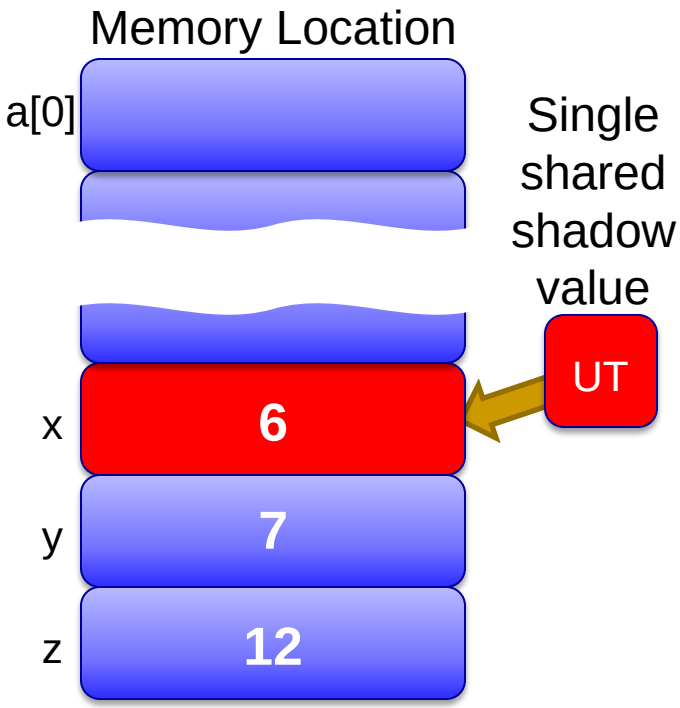


Dataflow Sampling Example: 1st User

Code:

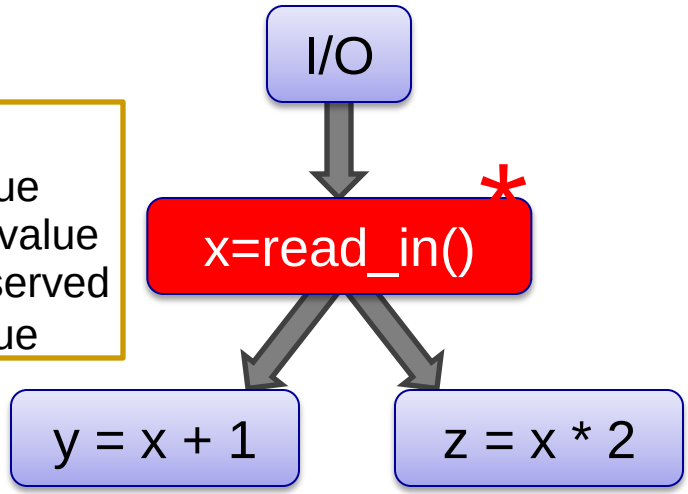
```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Memory:



Dataflow:

Key:
= shadow value
= no shadow value
* = globally observed shadow value

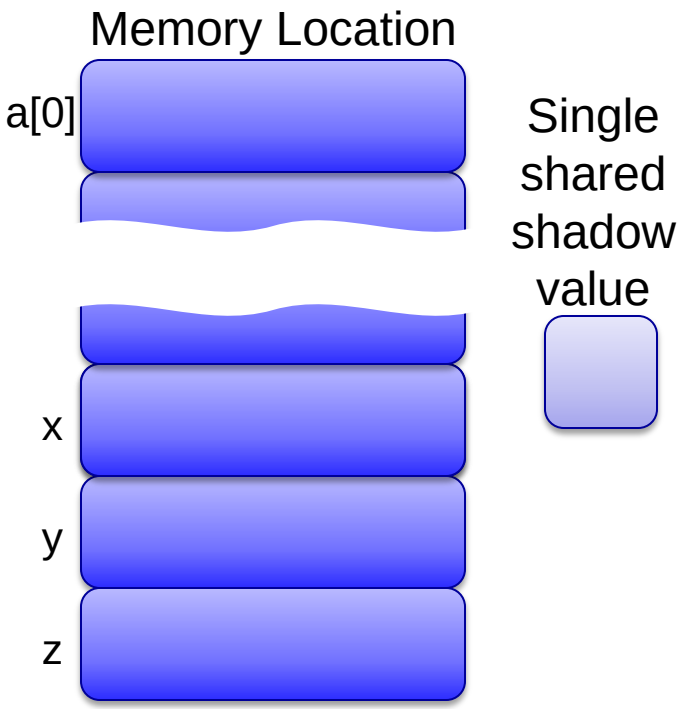


Dataflow Sampling Example: 2nd User

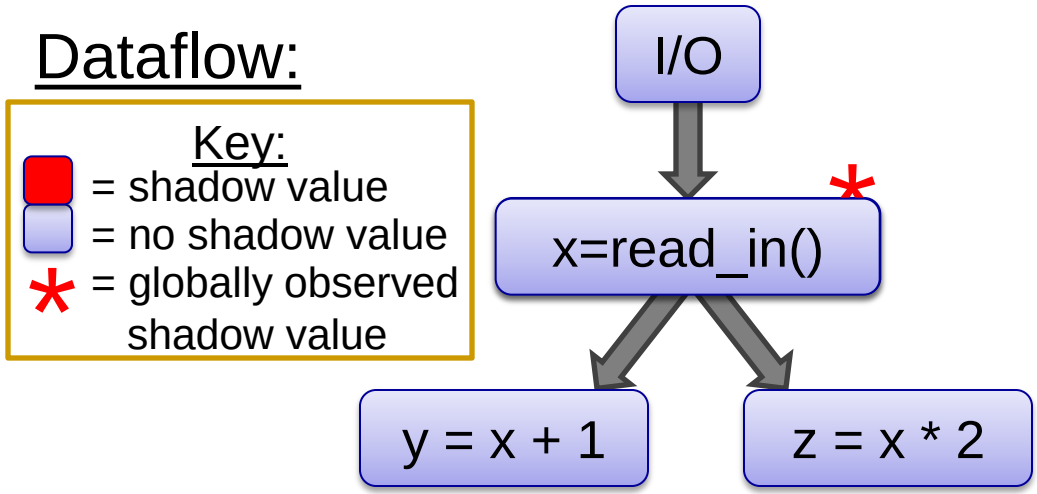
Code:

```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Memory:



Dataflow:

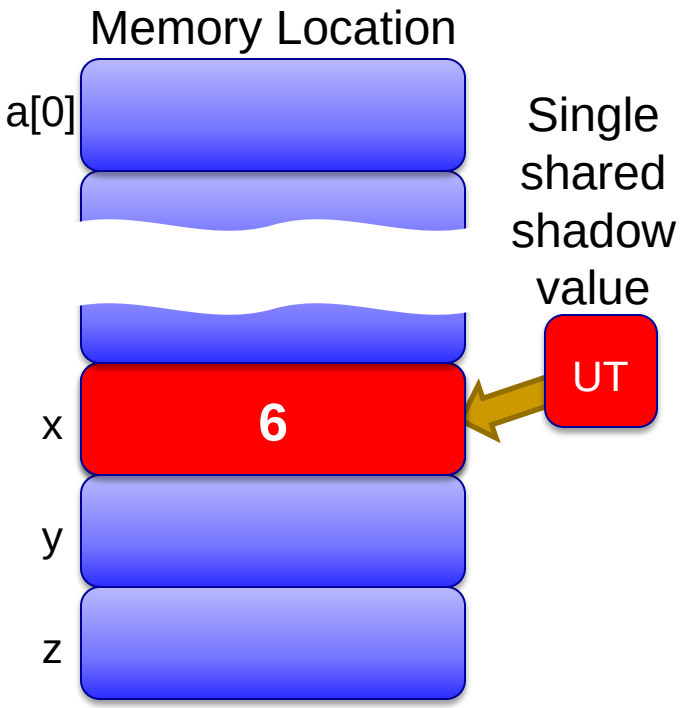


Dataflow Sampling Example: 2nd User

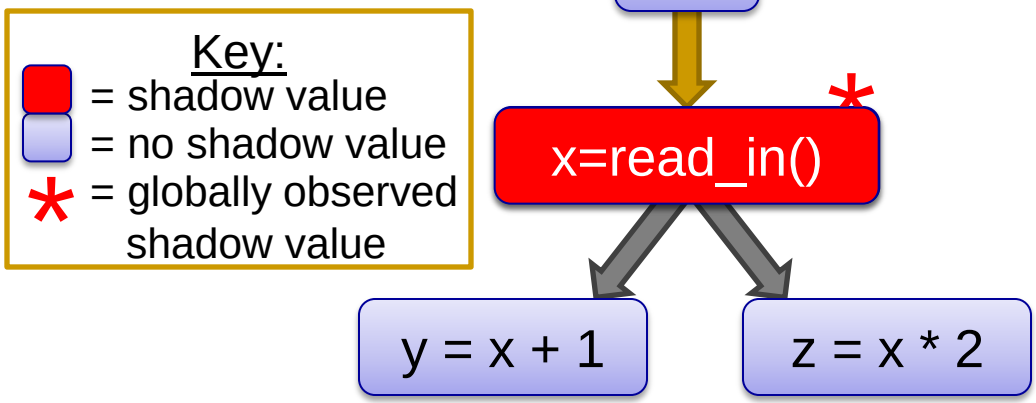
Code:

```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Memory:



Dataflow:

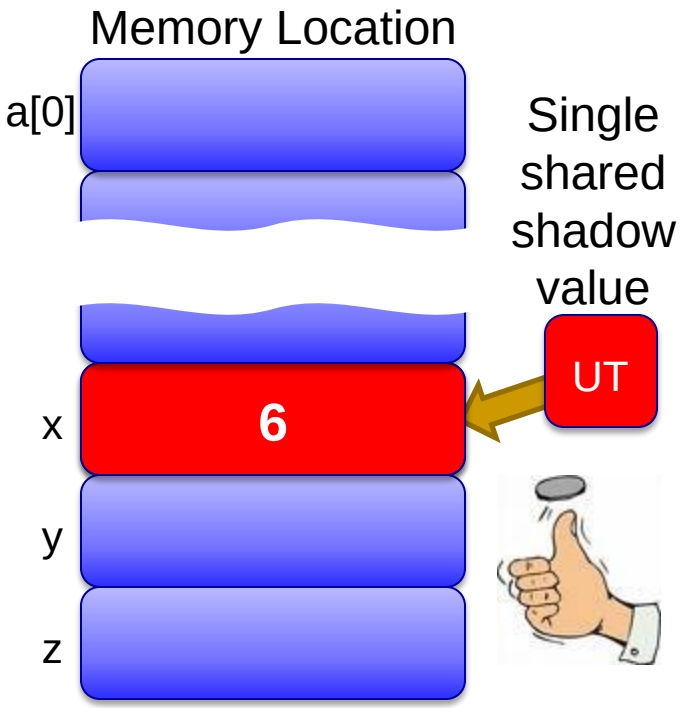


Dataflow Sampling Example: 2nd User

Code:

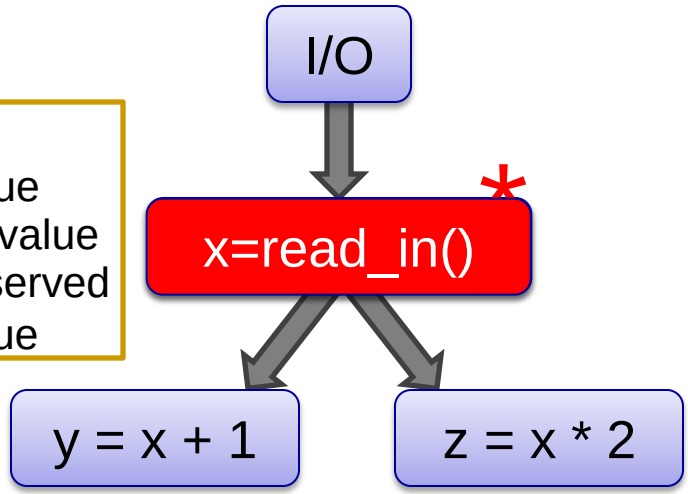
```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Memory:



Dataflow:

Key:
= shadow value
= no shadow value
* = globally observed shadow value

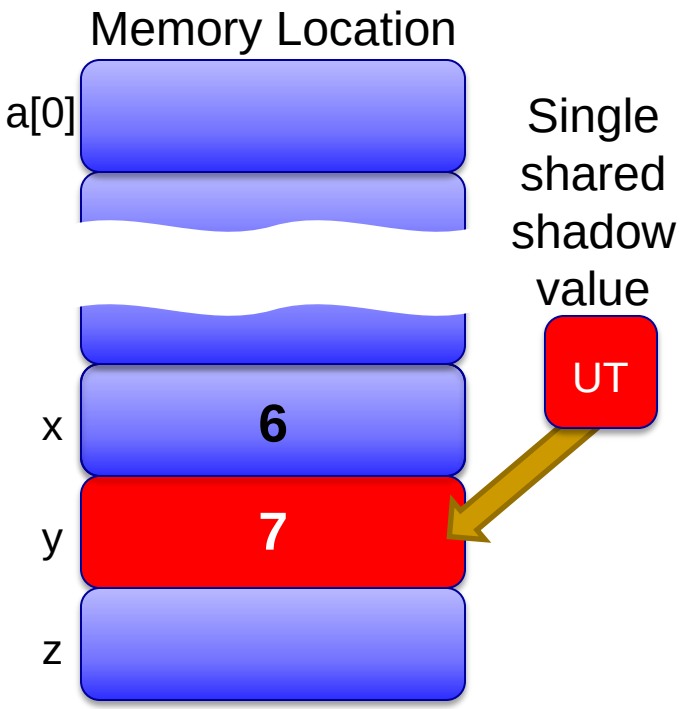


Dataflow Sampling Example: 2nd User

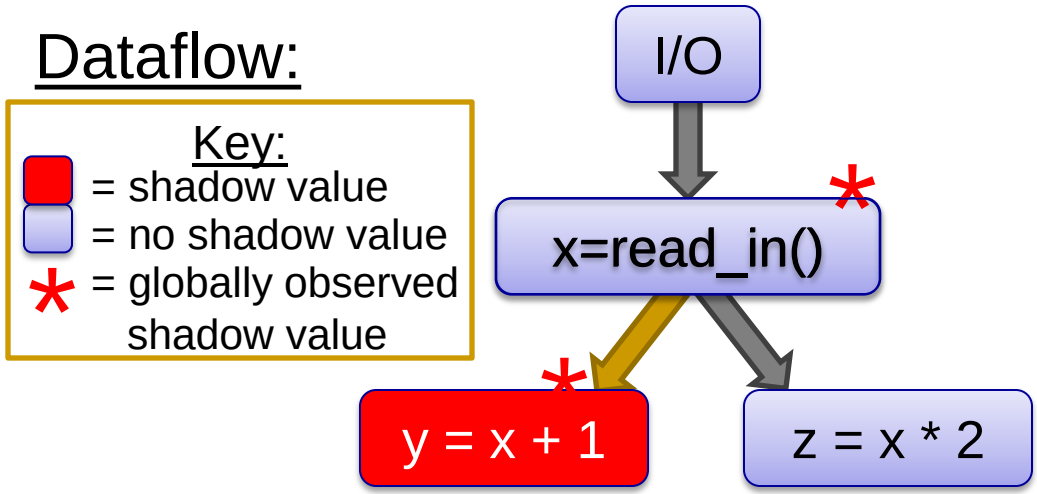
Code:

```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Memory:



Dataflow:

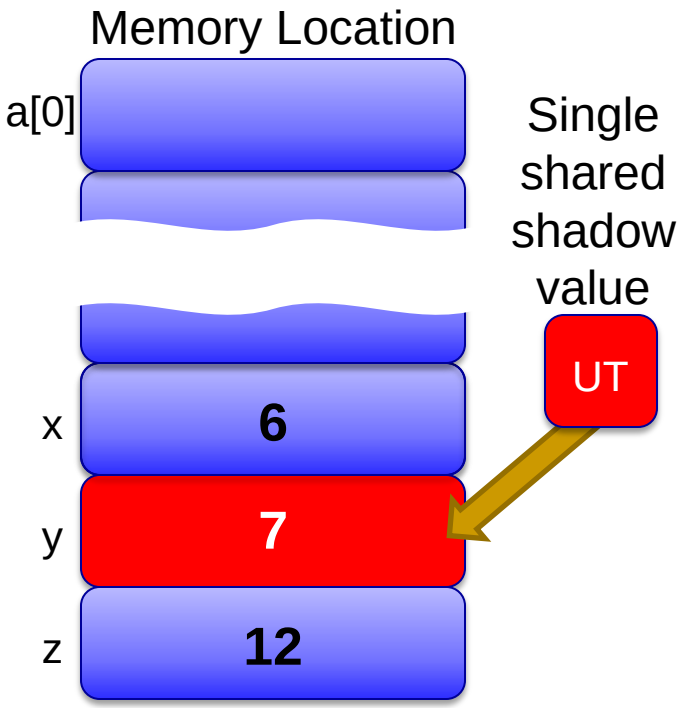


Dataflow Sampling Example: 2nd User

Code:

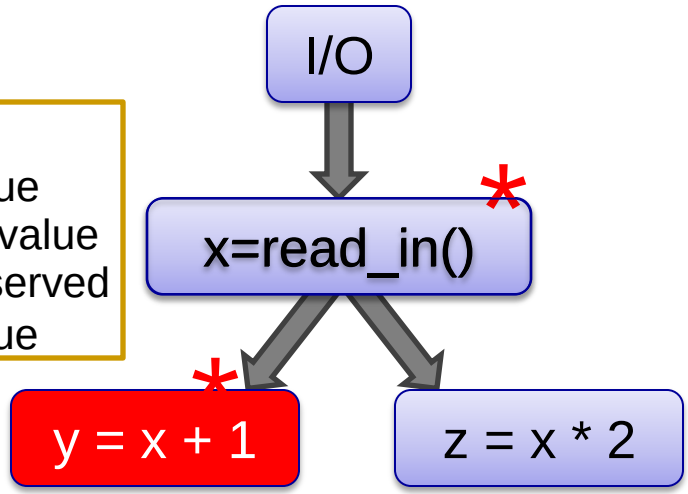
```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Memory:



Dataflow:

Key:
= shadow value
= no shadow value
* = globally observed shadow value

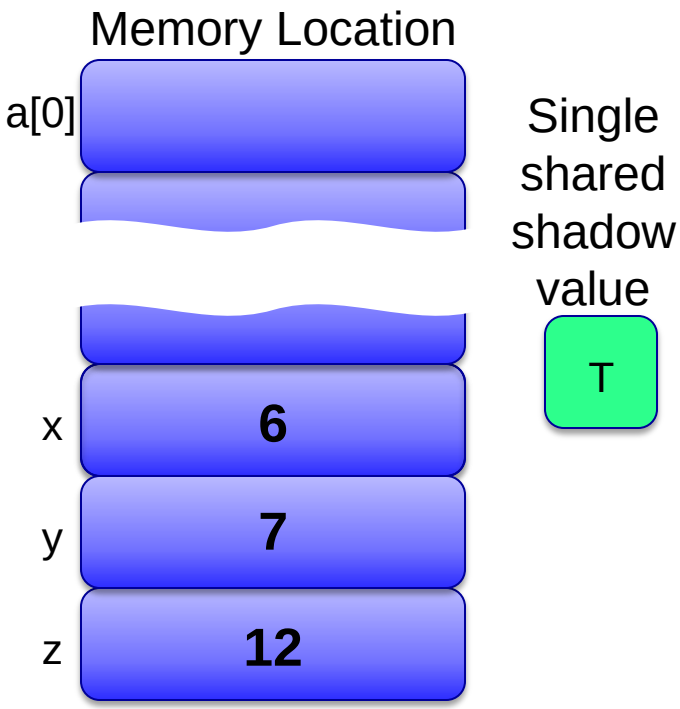


Dataflow Sampling Example: 2nd User

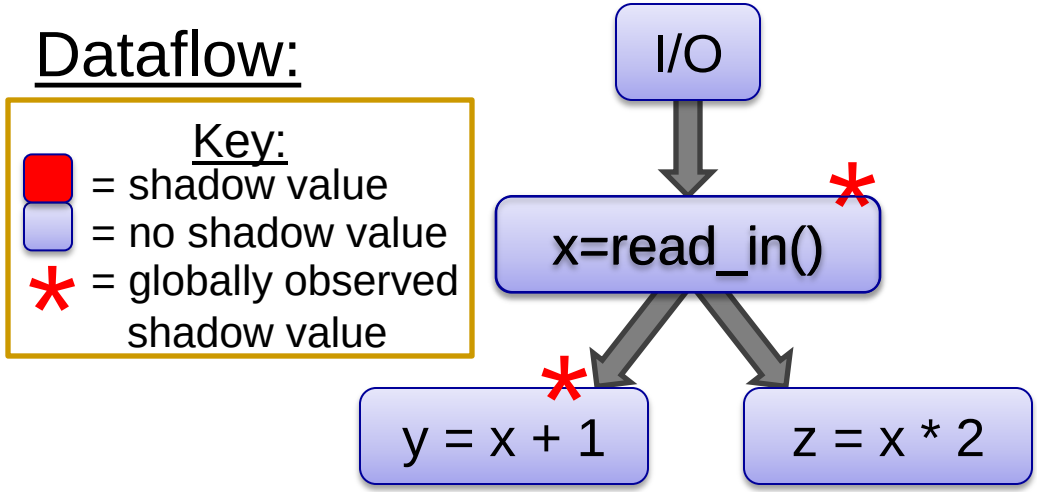
Code:

```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Memory:



Dataflow:

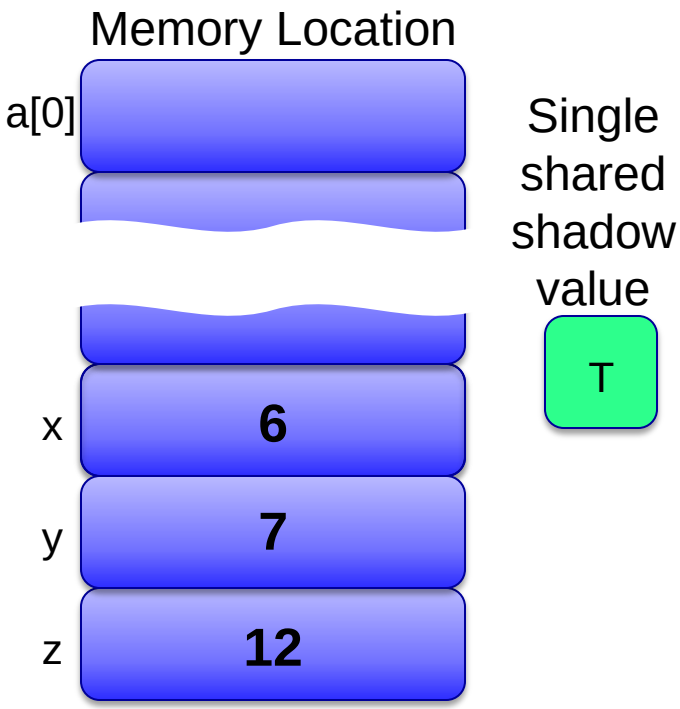


Dataflow Sampling Example: 2nd User

Code:

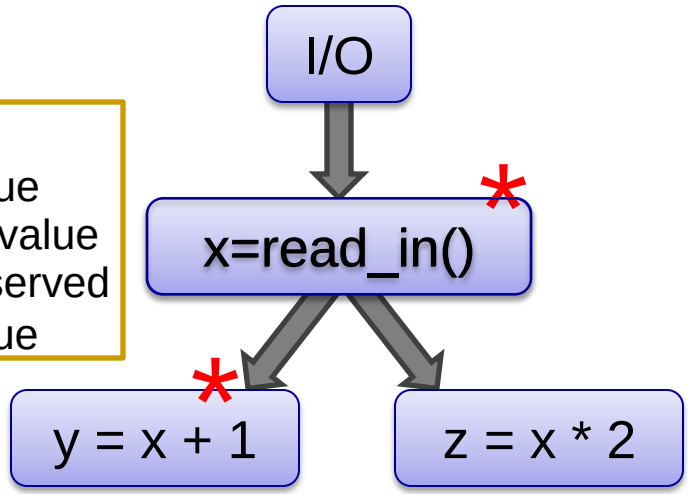
```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Memory:



Dataflow:

Key:
= shadow value
= no shadow value
* = globally observed shadow value

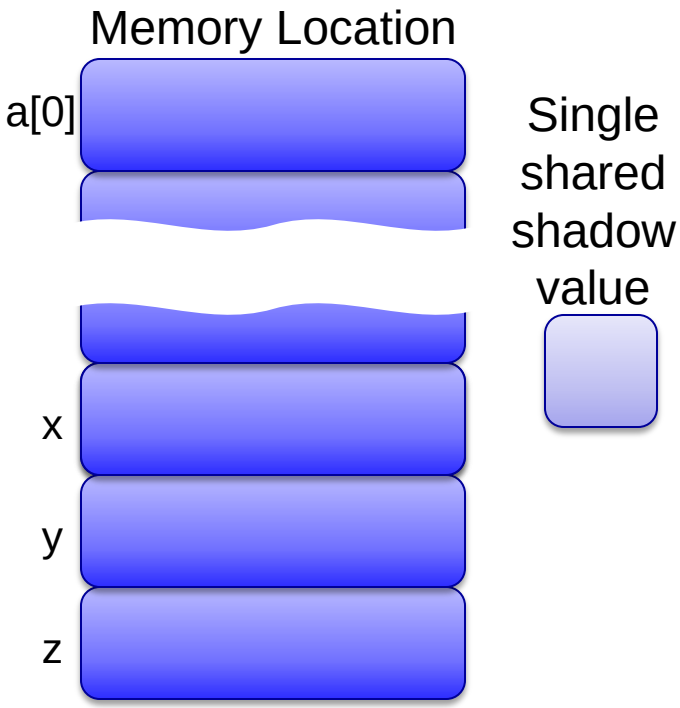


Dataflow Sampling Example: 3rd User

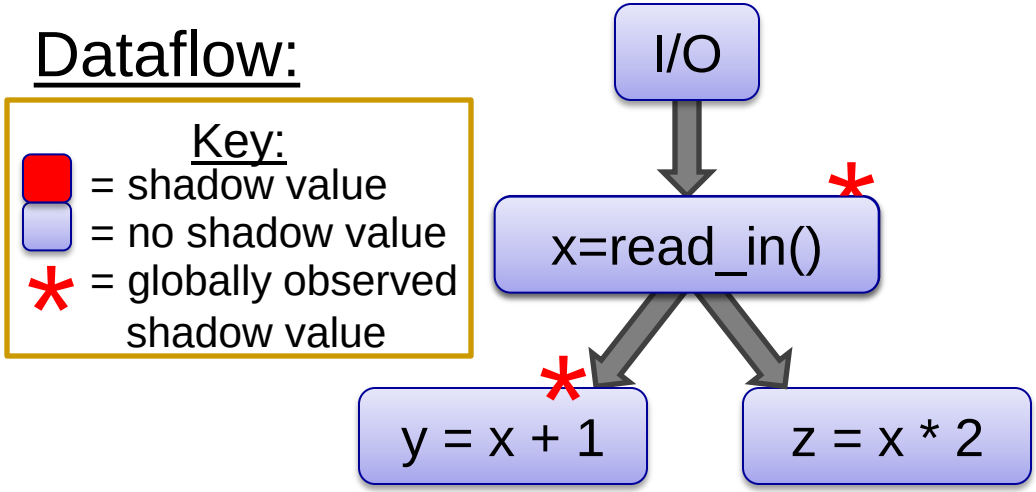
Code:

```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Memory:



Dataflow:

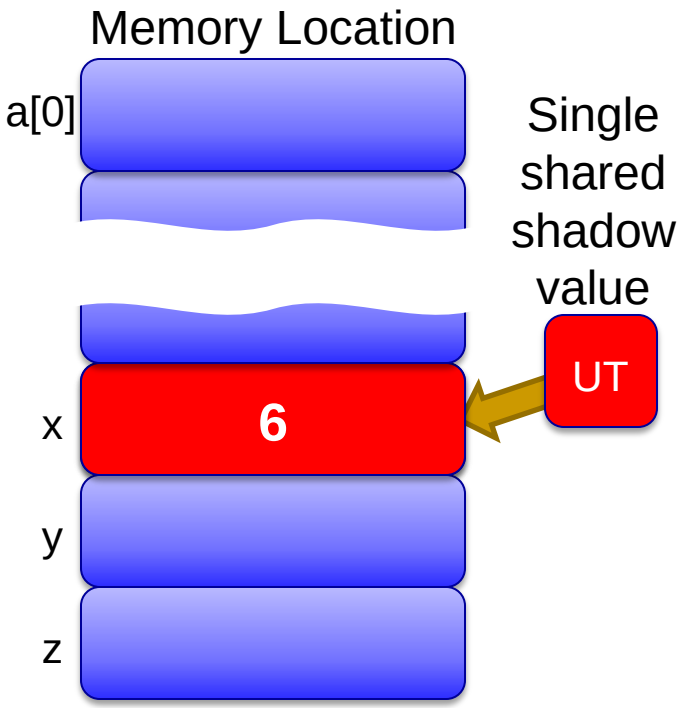


Dataflow Sampling Example: 3rd User

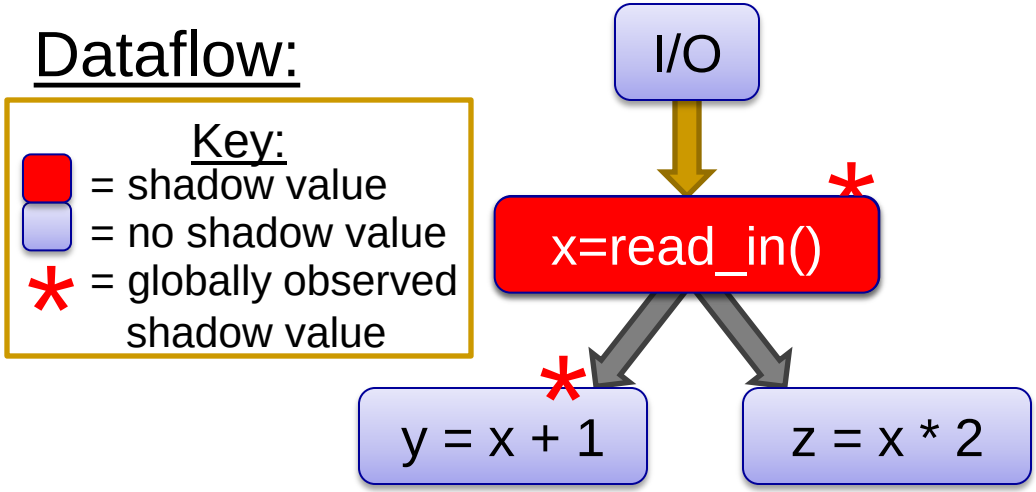
Code:

```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Memory:



Dataflow:



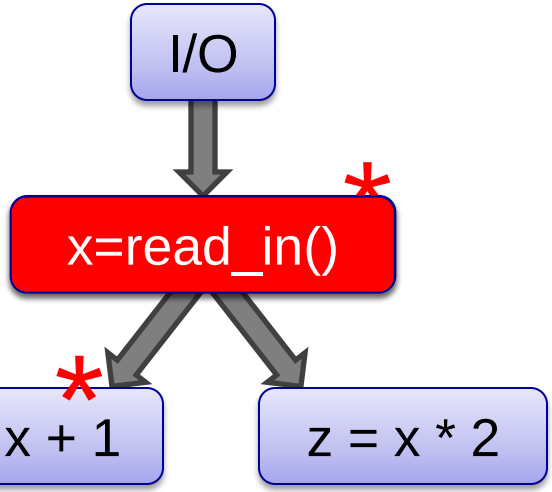
Dataflow Sampling Example: 3rd User

Code:

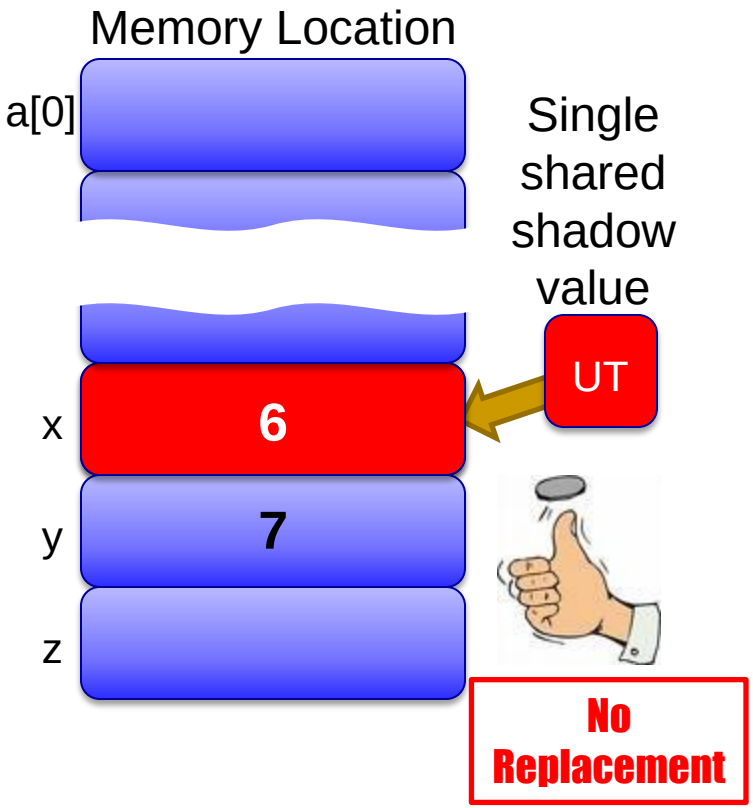
```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Dataflow:

Key:
- = shadow value
- = no shadow value
- * = globally observed shadow value



Memory:

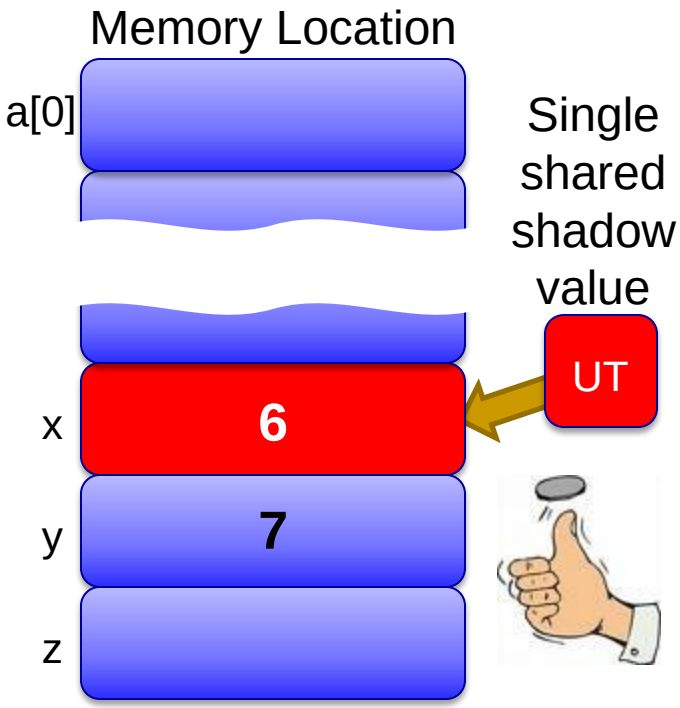


Dataflow Sampling Example: 3rd User

Code:

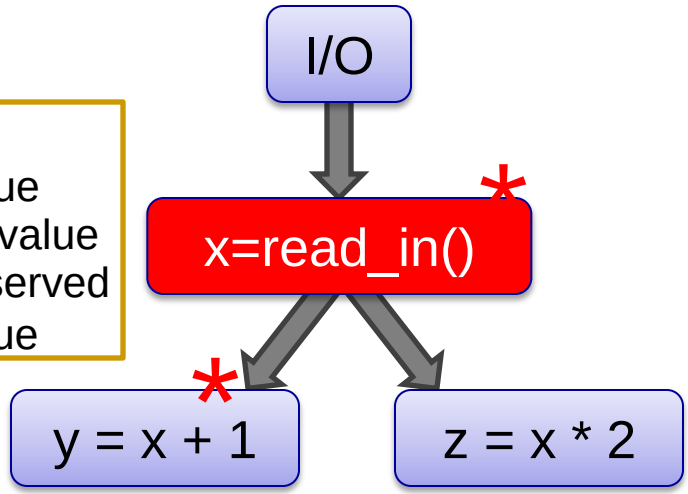
```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Memory:



Dataflow:

Key:
= shadow value
= no shadow value
* = globally observed shadow value

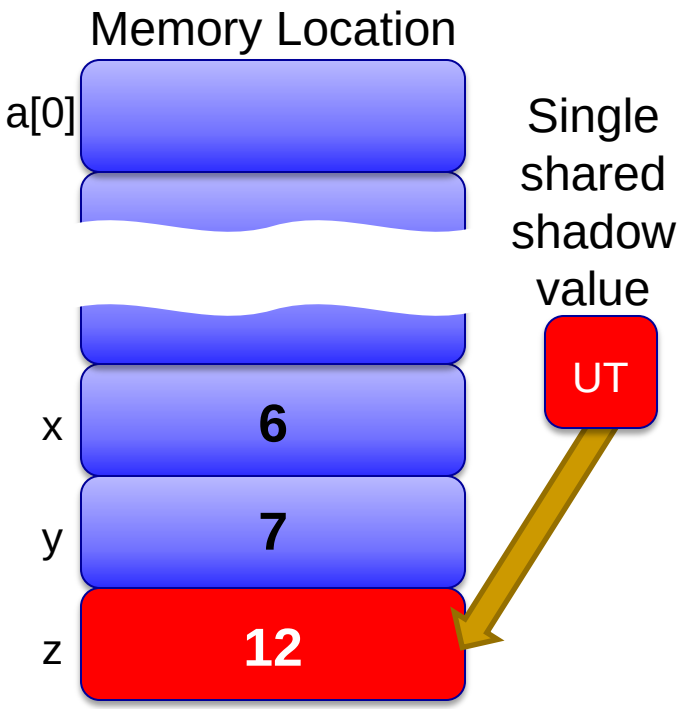


Dataflow Sampling Example: 3rd User

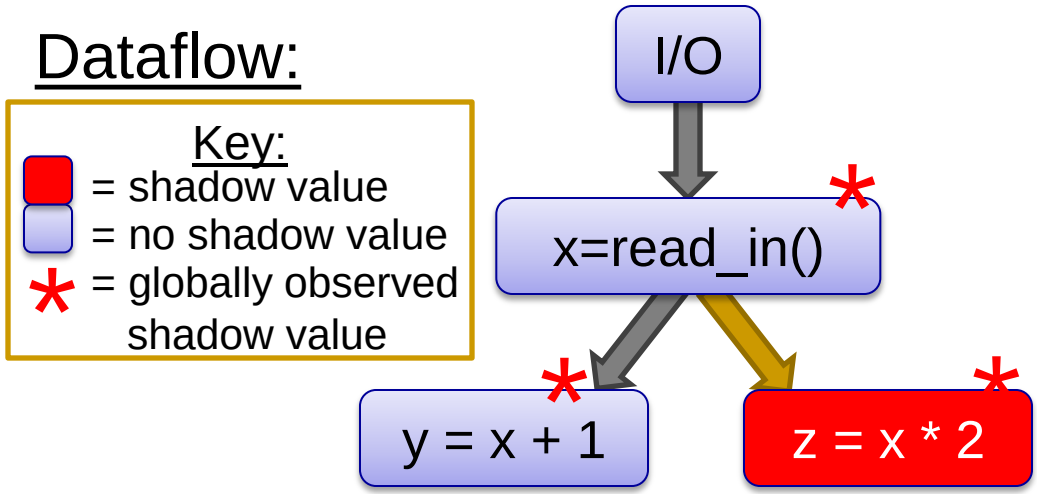
Code:

```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Memory:



Dataflow:

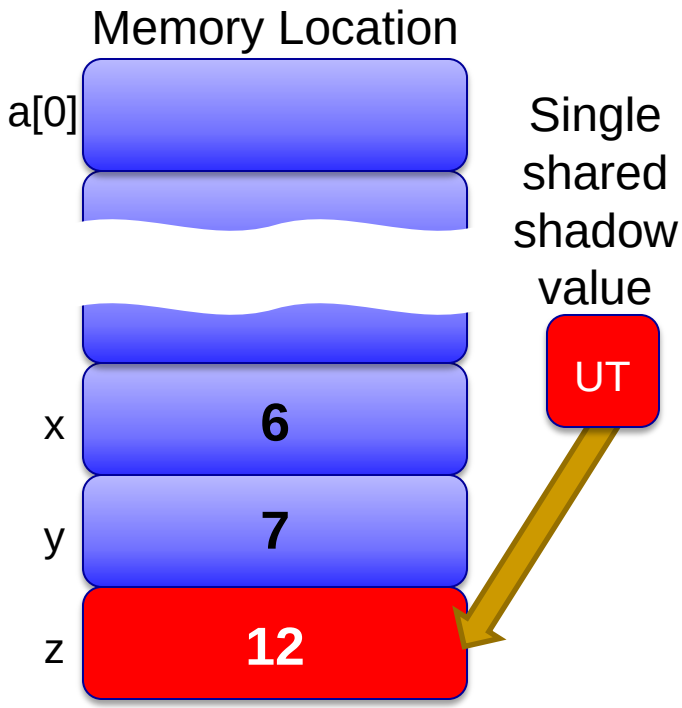


Dataflow Sampling Example: 3rd User

Code:

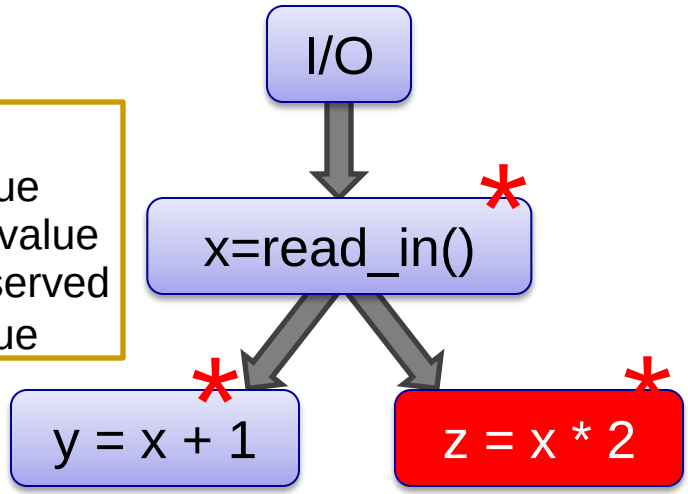
```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Memory:



Dataflow:

Key:
= shadow value
= no shadow value
* = globally observed shadow value

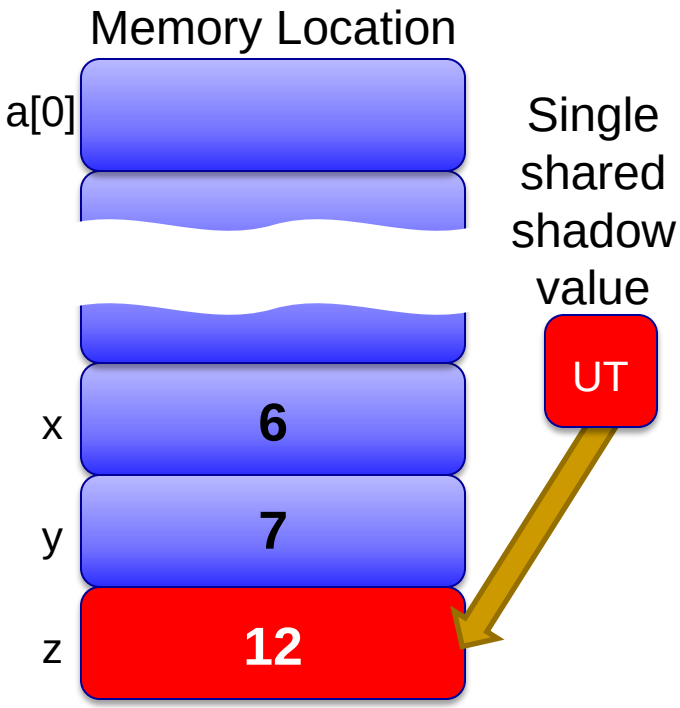


Dataflow Sampling Example: 3rd User

Code:

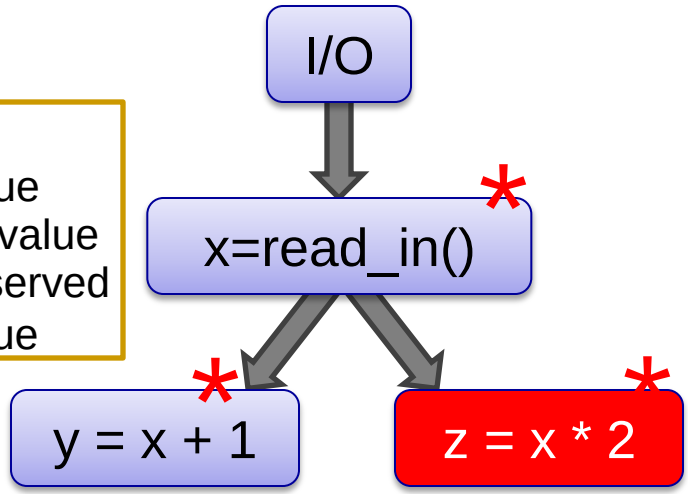
```
int sample(int a[8]){
  int x = read_in();
  int y = x + 1;
  int z = x * 2;
  print a[x];
  if(y>0&&y<8)
    print a[y];
  return a[z];
}
```

Memory:

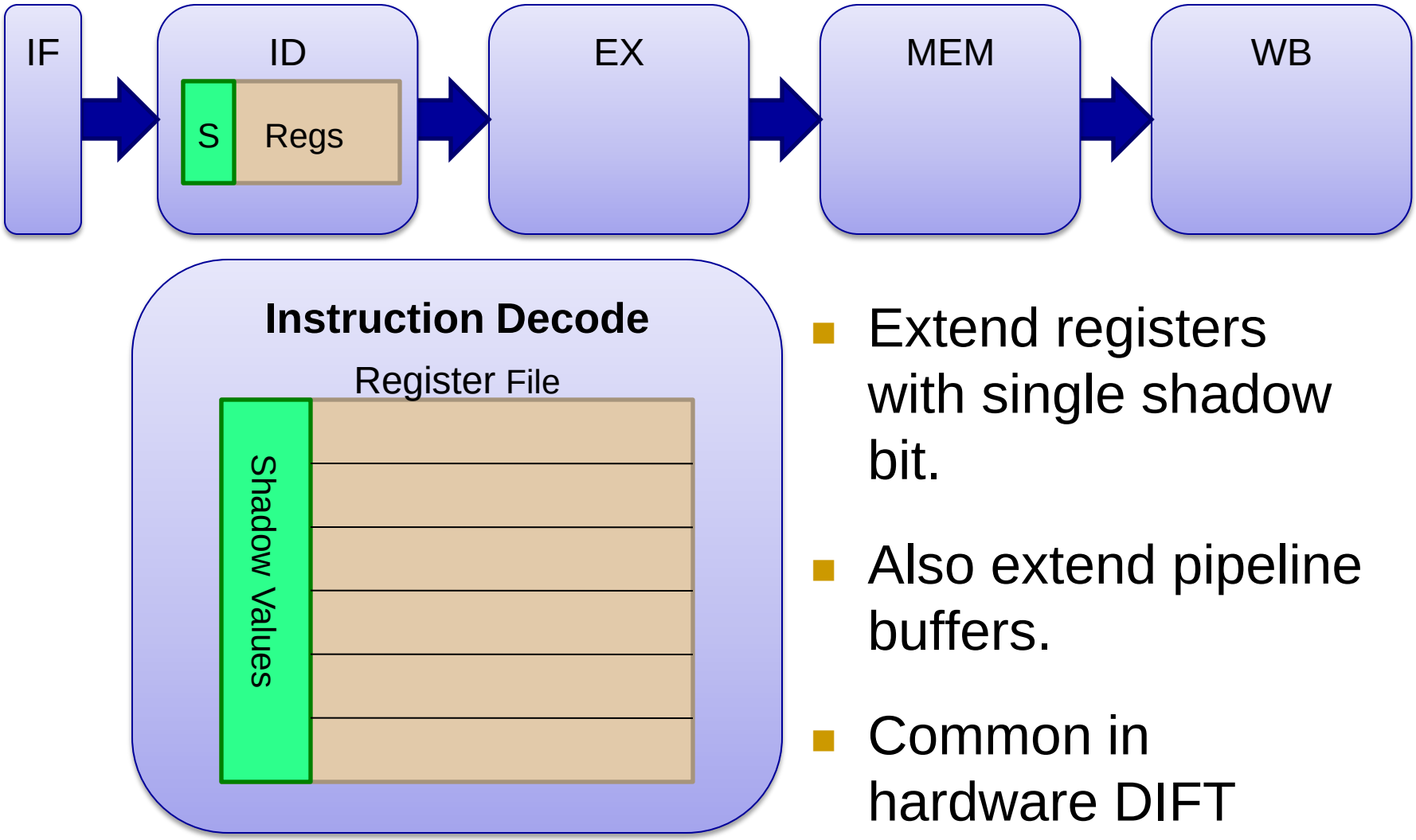


Dataflow:

Key:
= shadow value
= no shadow value
* = globally observed shadow value

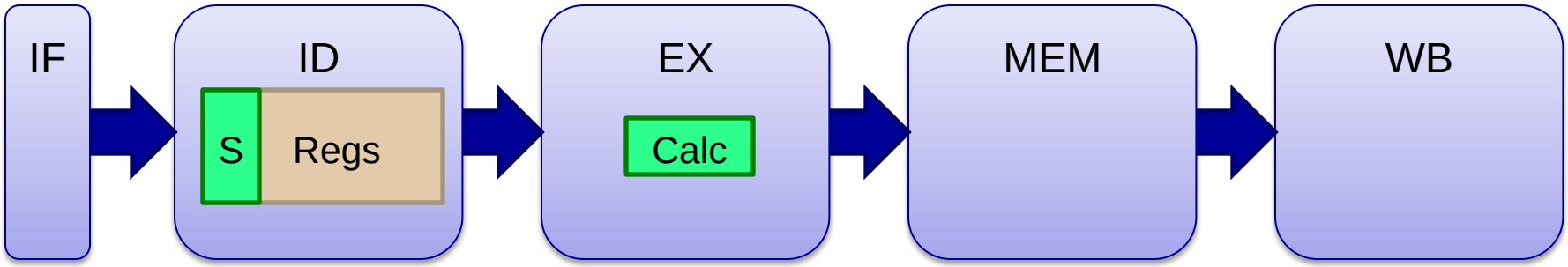


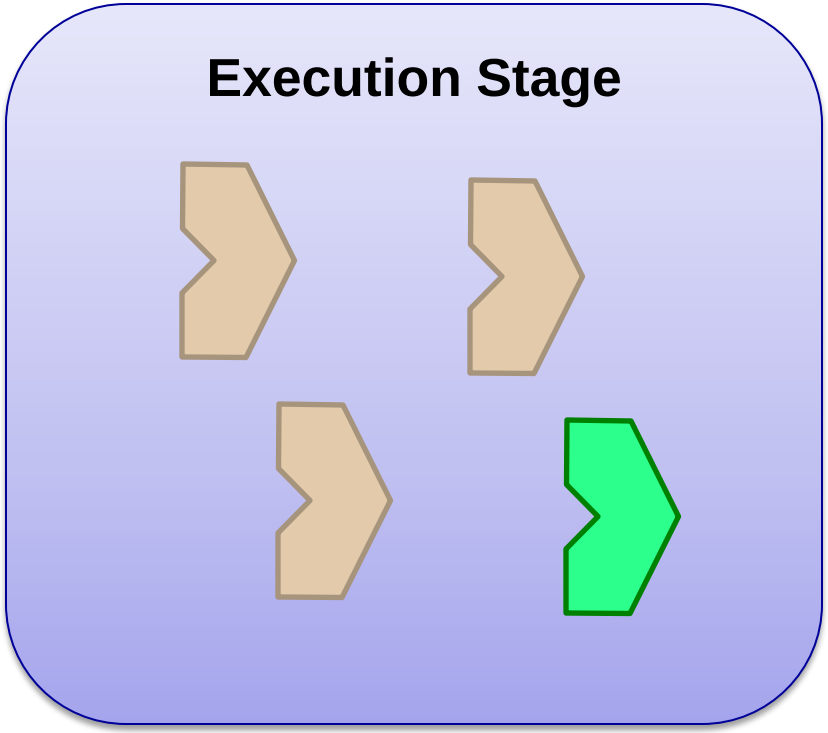
A Pipeline for Testudo



- Extend registers with single shadow bit.
- Also extend pipeline buffers.
- Common in hardware DIFT

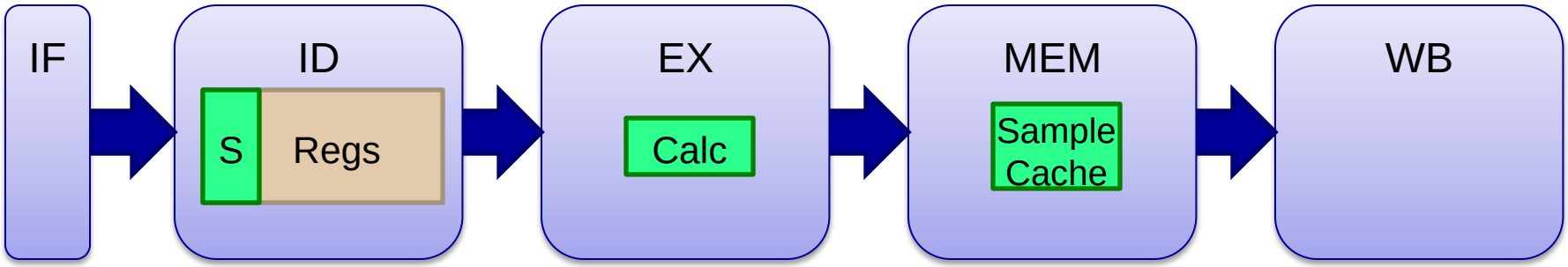
A Pipeline for Testudo

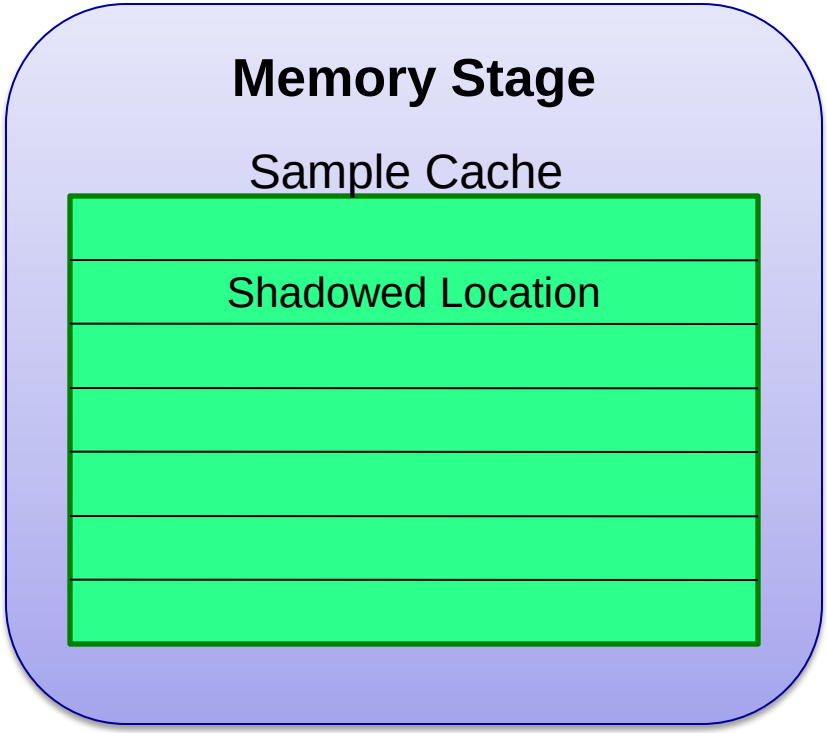




- Compute new shadow bits
- Generally hard-coded in some way
- Common in hardware DIFT

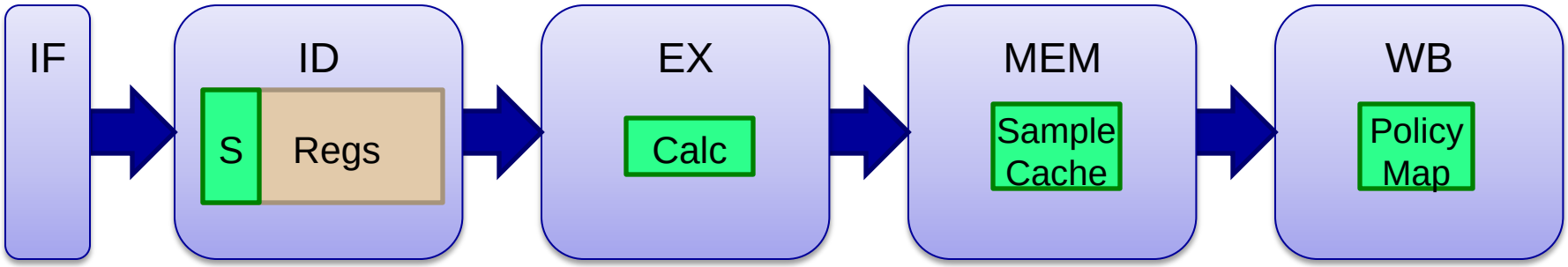
A Pipeline for Testudo

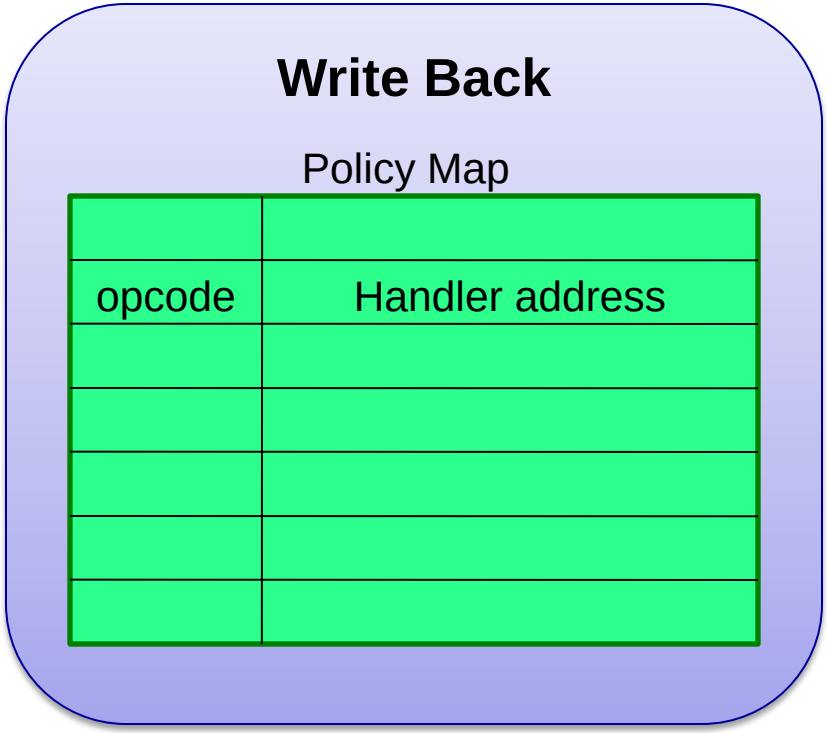




- Contains addresses of shadowed variables
- Drop some values when this small cache is full
- Replacement scheme *not* deterministic

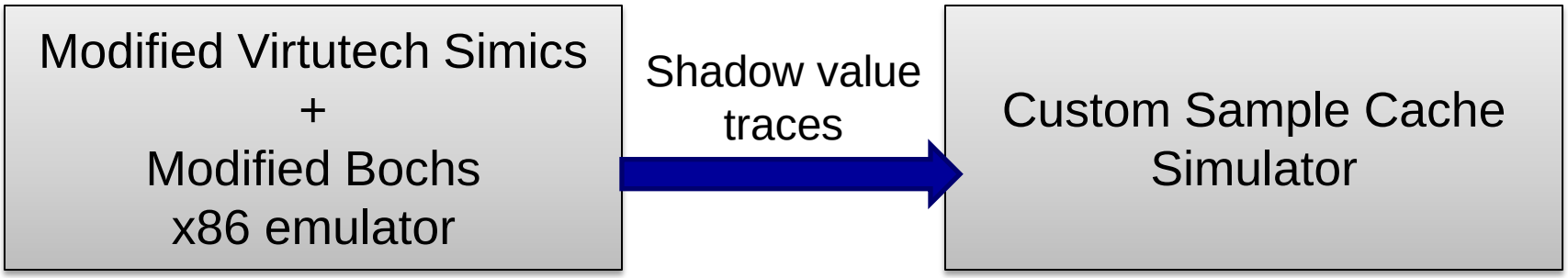
A Pipeline for Testudo





- Different actions & opcodes lead to different handler code
- Replaces or augments EX-stage modifications
- Allows for complicated testing and propagation

Experimental Framework



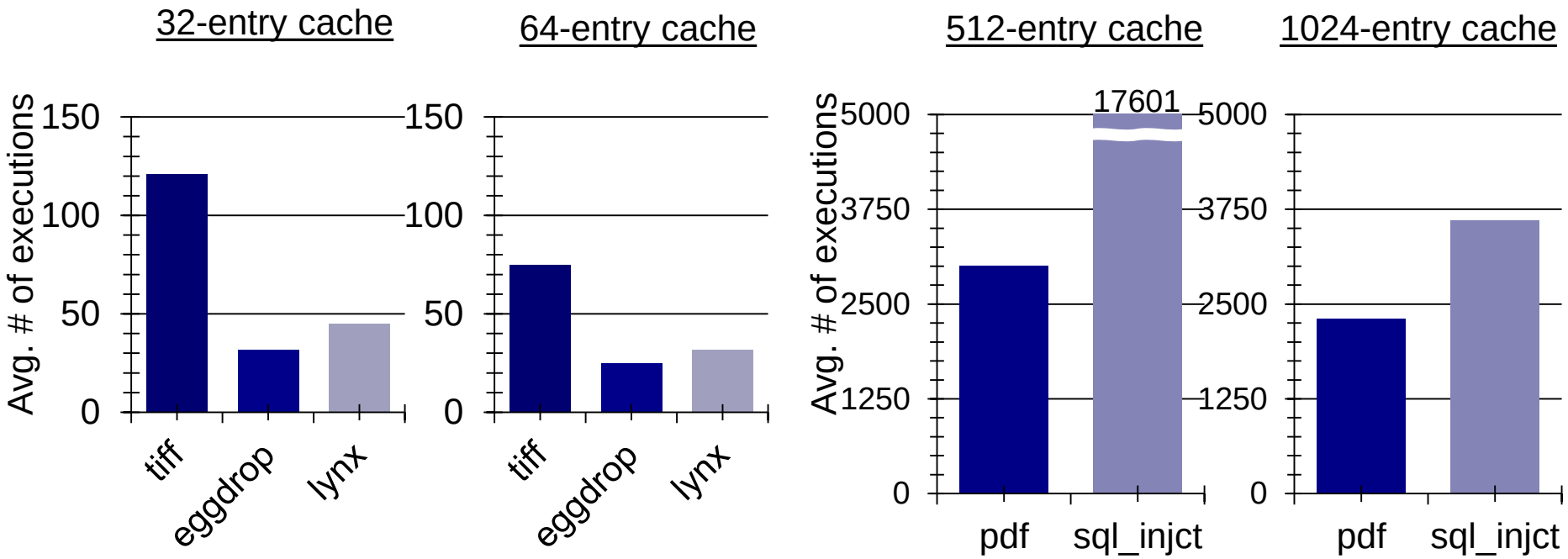
- Insecure programs (with exploits), including:
  - TIFF image engine
  - Eggdrop IRC bot
  - Lynx web browser
  - PDF library
  - Simulated SQL injection
- CACTI v5.0 used for cache estimation.

How many runs will I need?

To see all of a program's dataflows with high statistical confidence.

Some need few samples with tiny sample caches

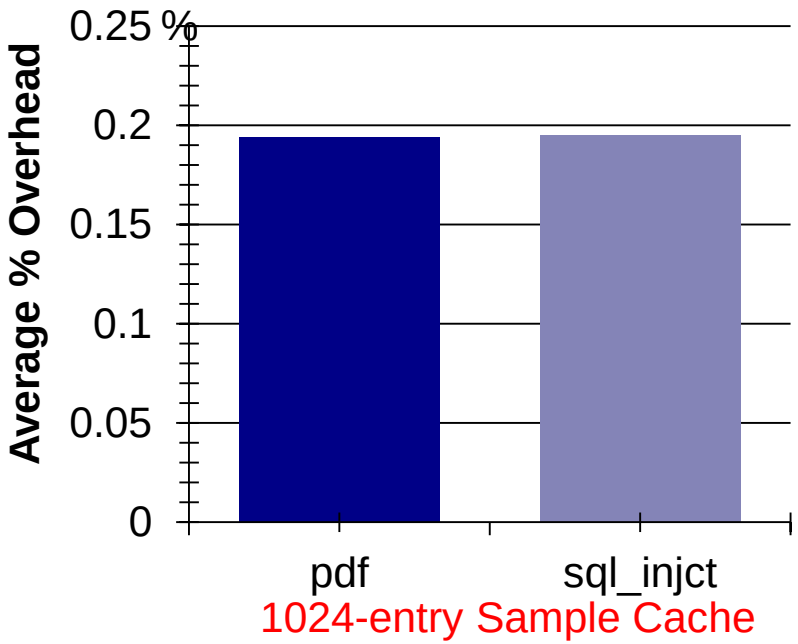
Others need a larger cache and/or more runs for good results



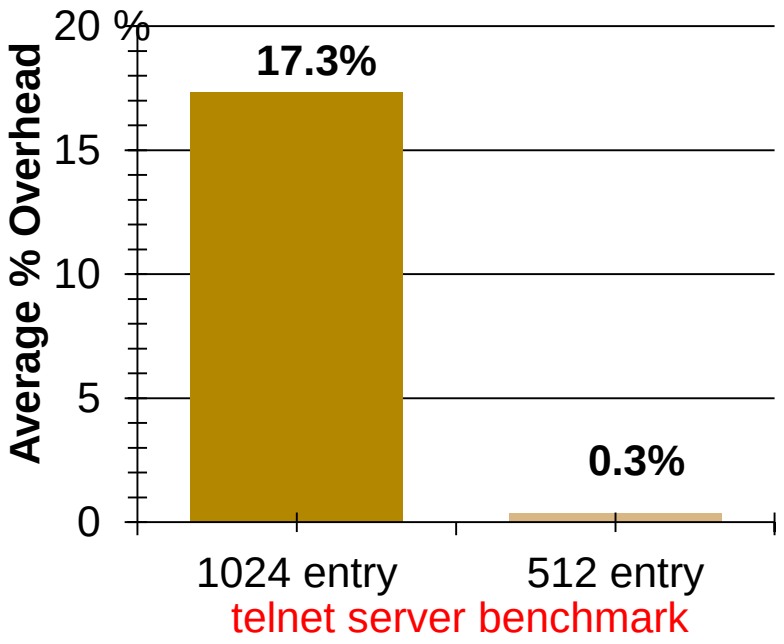
Does it scale to complex analyses?

If each shadow operation uses 1000 instructions:

Each execution sees few shadow values



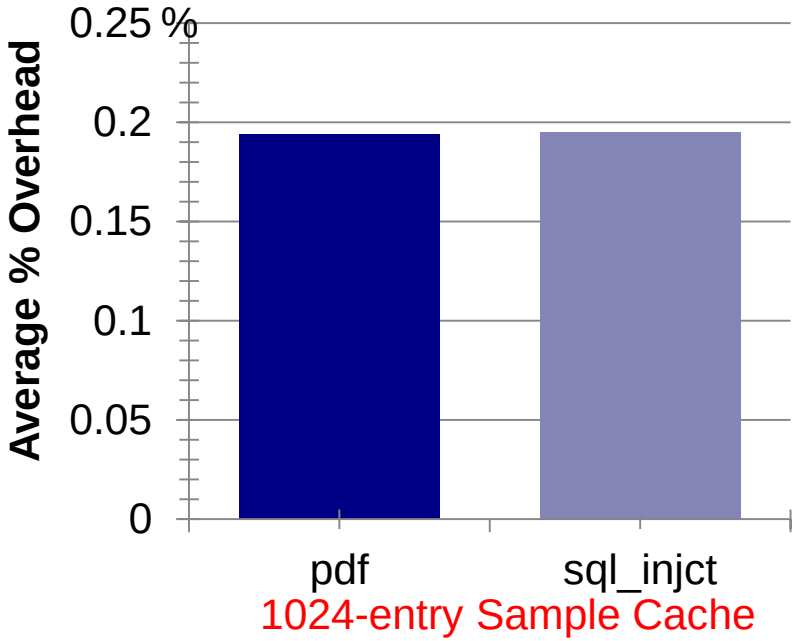
Fewer shadow values reduce overhead



Does it scale to complex analyses?

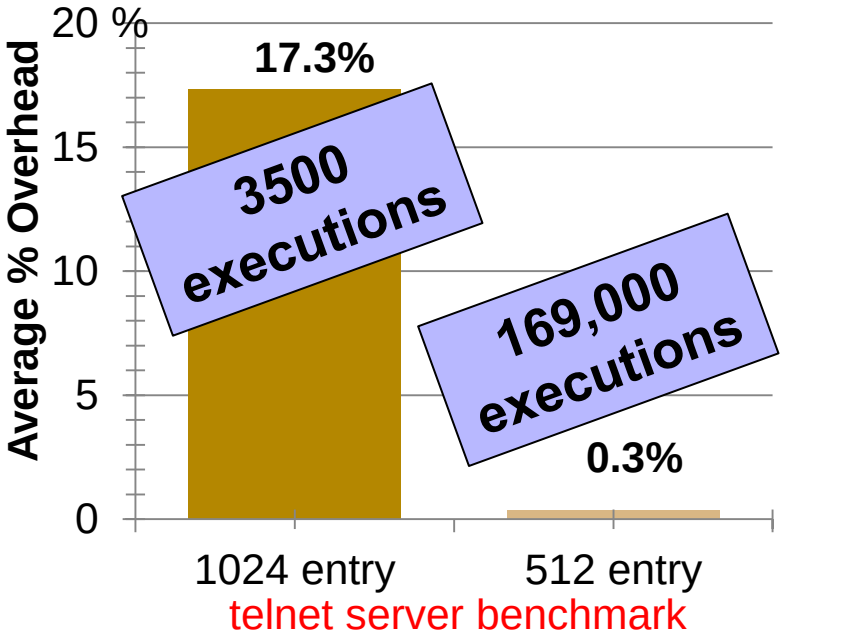
If each shadow operation uses 1000 instructions:

Each execution sees few shadow values



Fewer shadow values reduce overhead

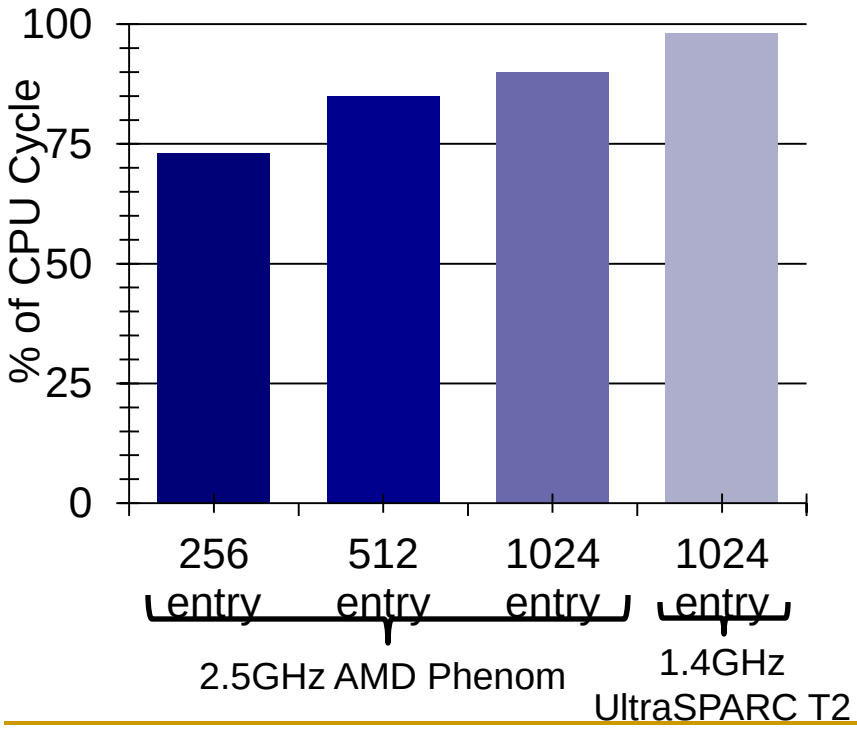
Tradeoff: need more executions



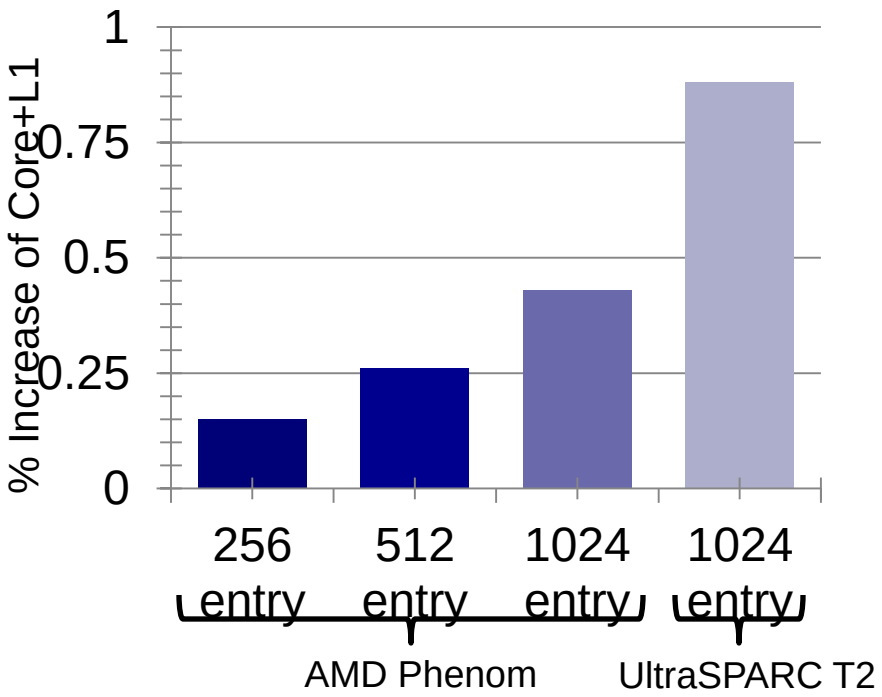
How much will it cost?

- ~0 change in clock period of modern CPU
- No overhead outside of the CPU core
- Very low hardware overhead in CPU core

Sample Cache Access Time



Sample Cache Area Overhead



Conclusions from Testudo

- Simplified hardware design for dynamic analysis
- Reduced runtime overhead for heavyweight security analysis
- Increased heavyweight dynamic analysis quality

Future Directions

- Adapting Testudo hardware for multicore
- What is the best cache replacement method?
- Can we skip the hardware additions?

Thank you

BACKUP SLIDES

- Picture rights:
  - One of the following Testudo pictures has been removed, but I don't remember which one.
- Testudo picture 1
  http://www.flickr.com/photos/frield/1254958611/
- Testudo picture 2
  http://www.flickr.com/photos/manel/154985772/
- 'The Art of Debugging ...' and 'The IDA Pro Book':
  http://nostarch.com/
- Fuzzing for Software Security Testing and Quality Assurance
  http://www.artechhouse.com/Detail.aspx?strIsbn=978-1-59693-214-2
- Secure Programming with Static Analysis
  copyright Addison-Wesley

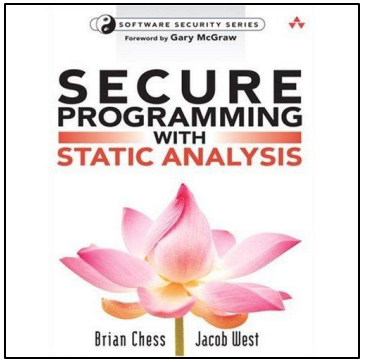
Systems for Detecting Security Errors

- Eyeballs
  - Disassembly, debugging, fuzz testing, whitehat/grayhat hackers
  - Time-consuming, difficult

- Static Analysis
  - Analyze source, formal reasoning methods, compile-time checks
  - Intractable, requires expert input, no system state

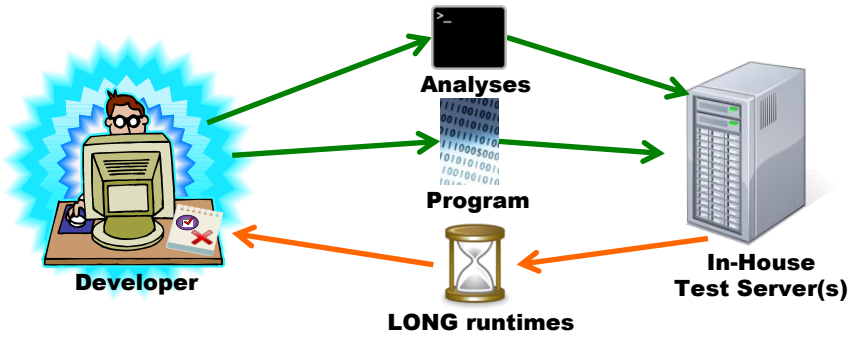
Klocwork

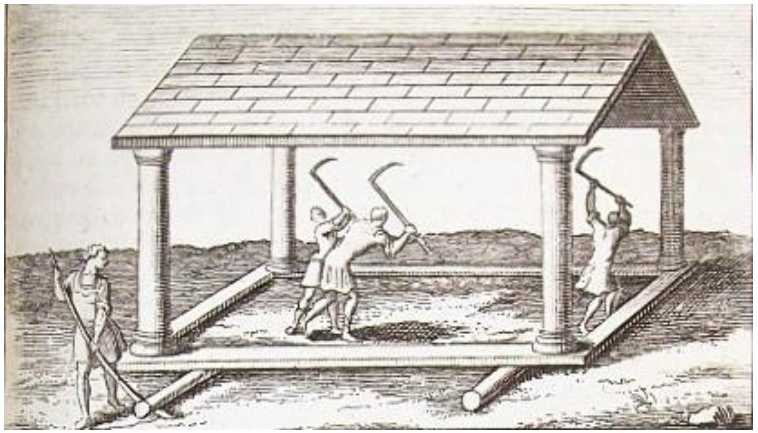
FORTIFY SOFTWARE



Testudo–Distributed Dynamic Debugging

Current Heavyweight Analysis Systems





Testudo: Heavyweight Sampling Analysis