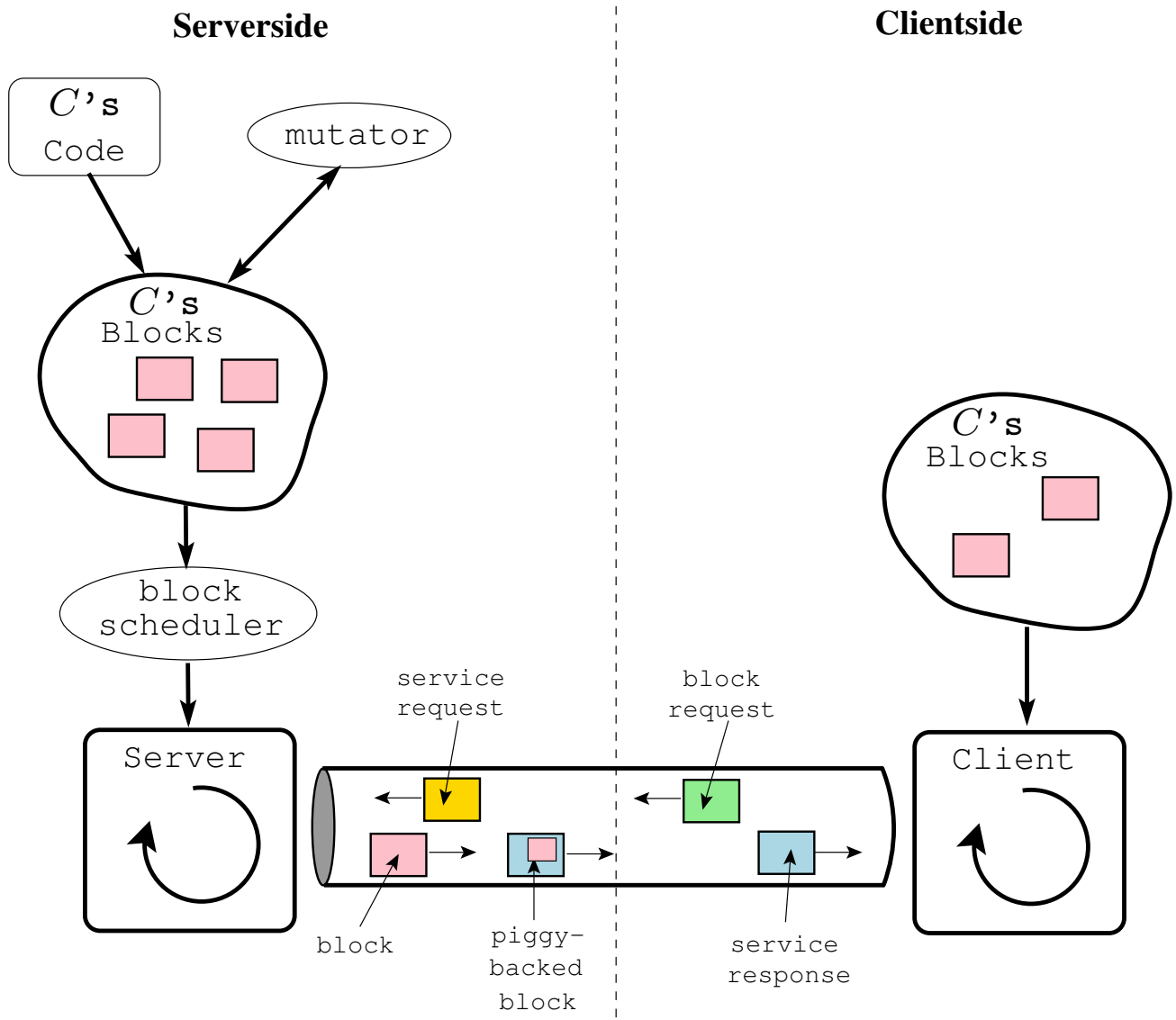
Continuous Replacement

Jasvir Nagra, Christian Collberg

November 19, 2007

Overview


Serverside
Clientside
C's Code
mutator
C's Blocks
block scheduler
Server
service request
block request
block
piggy-backed block
service response
C's Blocks
Client

Level of tamperproofing

The level of tamperproofing is determined by

1. the fraction of blocks that the client keeps

Level of tamperproofing

The level of tamperproofing is determined by

1. the fraction of blocks that the client keeps
2. the rate by which the server pushes mutated blocks to the client

Level of tamperproofing

The level of tamperproofing is determined by

1. the fraction of blocks that the client keeps
2. the rate by which the server pushes mutated blocks to the client
3. the rate by which the adversary can analyze the continuously changing program in the client's bag-of-blocks.

Performance

1. Generate blocks which take as long time to analyze as possible

Performance

1. Generate blocks which take as long time to analyze as possible
2. Little network traffic $\Rightarrow$ low block replacement rate

Performance

1. Generate blocks which take as long time to analyze as possible
2. Little network traffic $\Rightarrow$ low block replacement rate
3. Client's bag-of-blocks must never contain a complete and correct program.

Performance

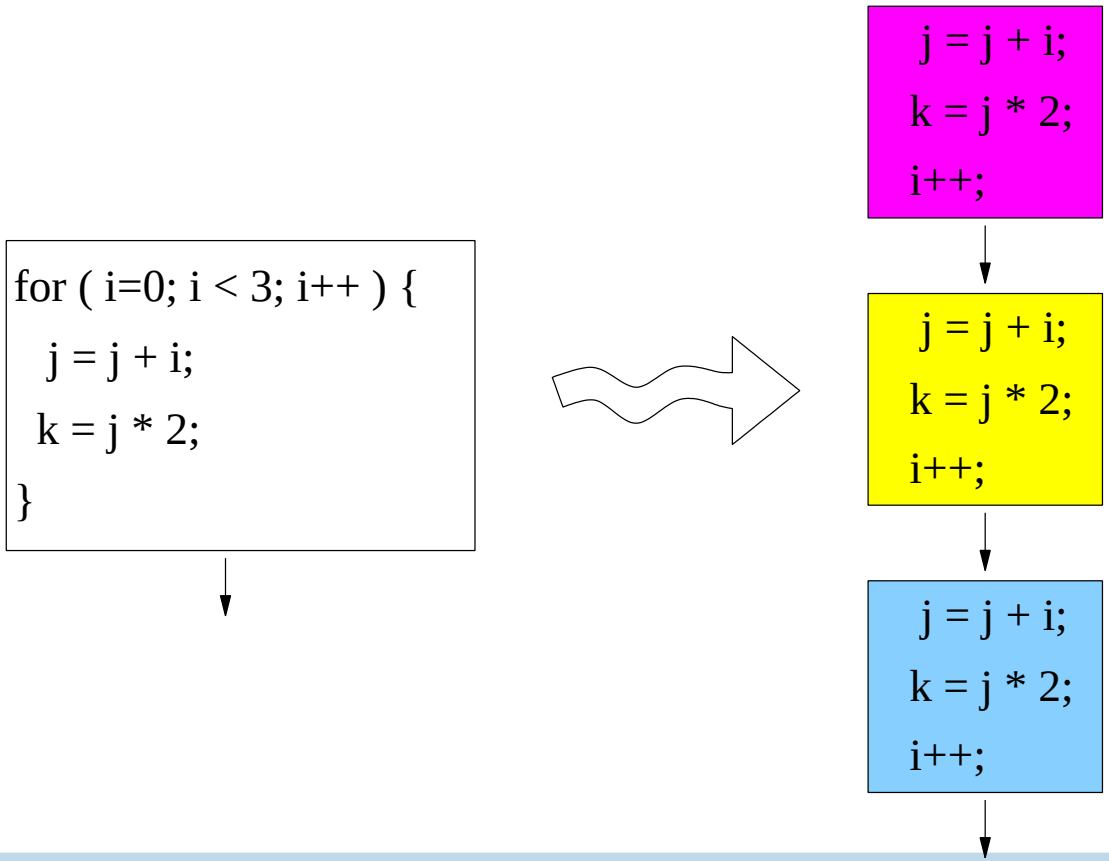
1. Generate blocks which take as long time to analyze as possible
2. Little network traffic $\Rightarrow$ low block replacement rate
3. Client's bag-of-blocks must never contain a complete and correct program.
   - $\Rightarrow$ Prevents the adversary taking snapsshots of the bag to analyze off-line.

Performance

1. Generate blocks which take as long time to analyze as possible
2. Little network traffic $\Rightarrow$ low block replacement rate
3. Client's bag-of-blocks must never contain a complete and correct program.
   - $\Rightarrow$ Prevents the adversary taking snapsshots of the bag to analyze off-line.
   - Ideally, even if the client saves a copy of all the blocks it has ever seen, the adversary still shouldn't be able to fully analyze the code (hard)

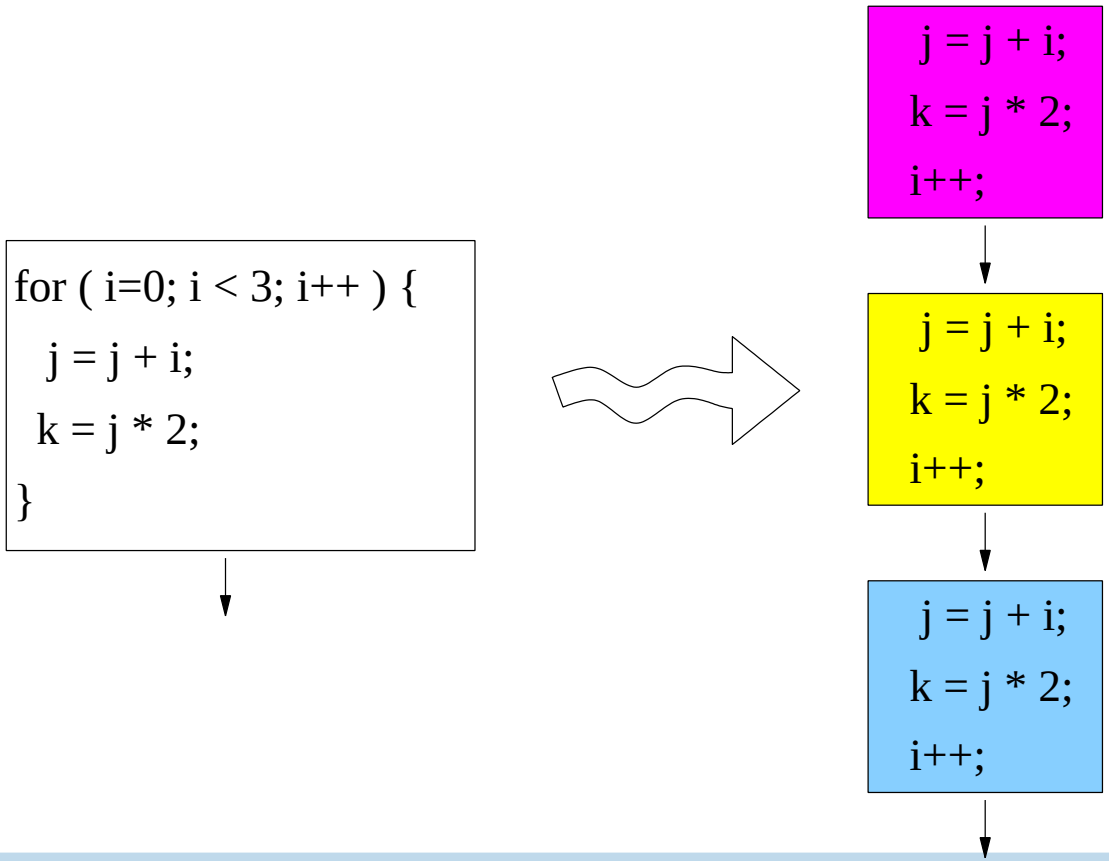
Block 1: Unrolled block

- Unroll a loop and send the client one iteration at a time.



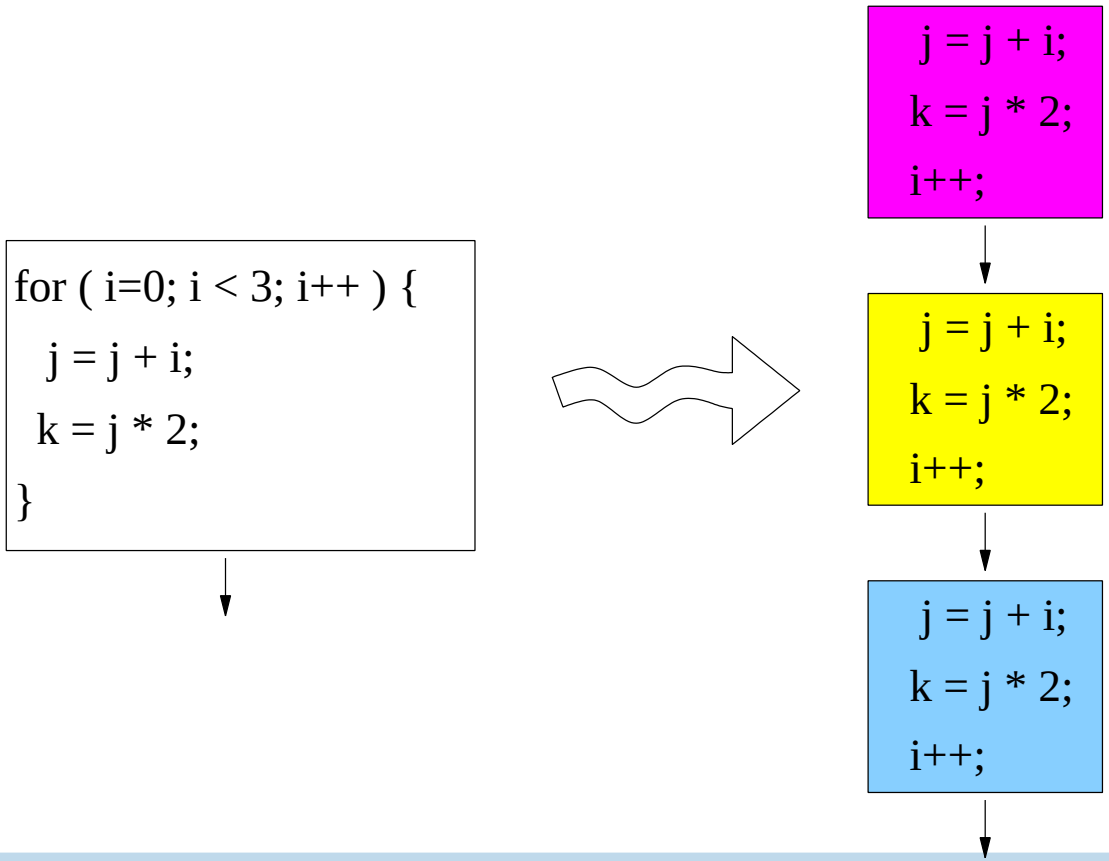
Block 1: Unrolled block

- Unroll a loop and send the client one iteration at a time.
- Renumber and/or obfuscate each loop body differently



Block 1: Unrolled block

- Unroll a loop and send the client one iteration at a time.
- Renumber and/or obfuscate each loop body differently
- $\Rightarrow$ Difficult for the adversary to reconstitute the loop.

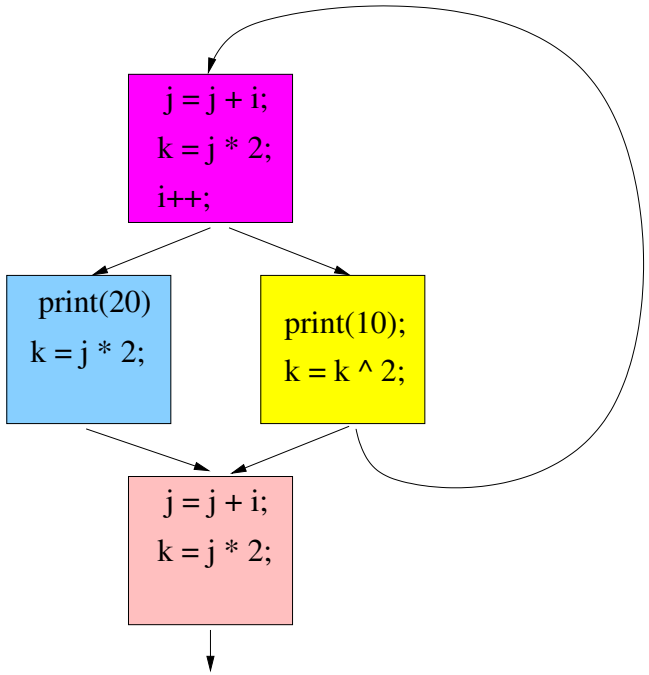


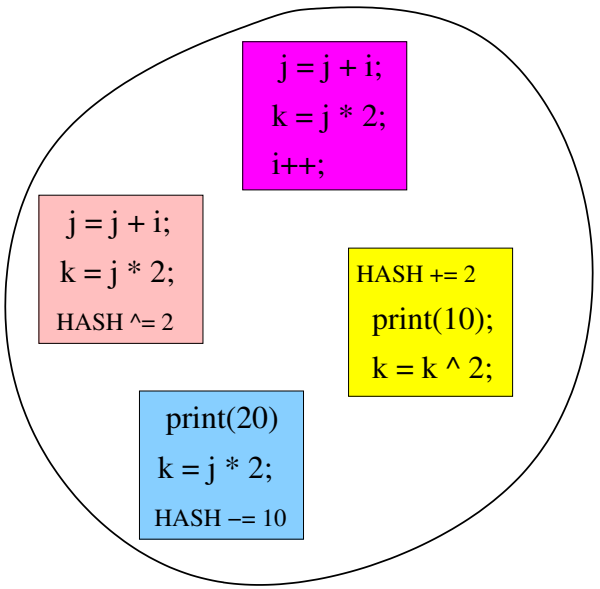
Block 2: Hashing block

- Compute a hash over another block and return the result to the server.

Block 3: Oblivious Hashing Block

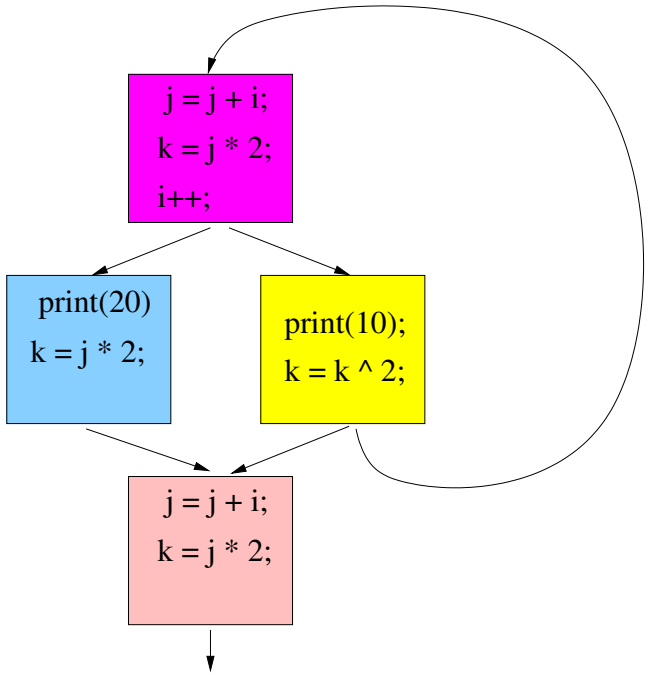
- Weave a hash computation into the control flow:

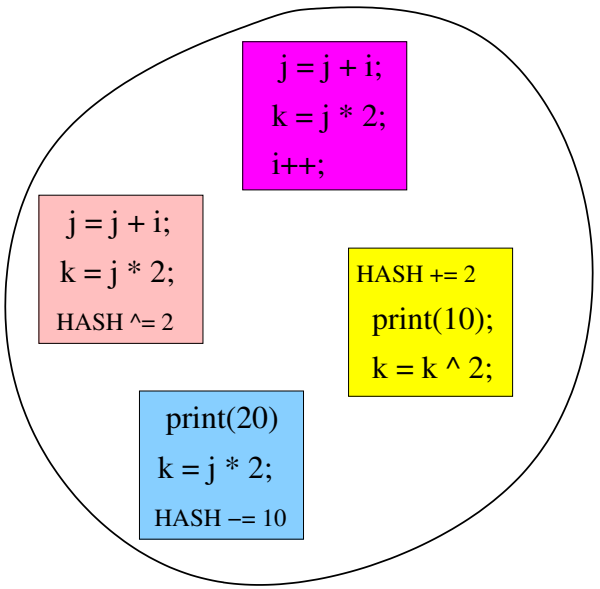




Block 3: Oblivious Hashing Block

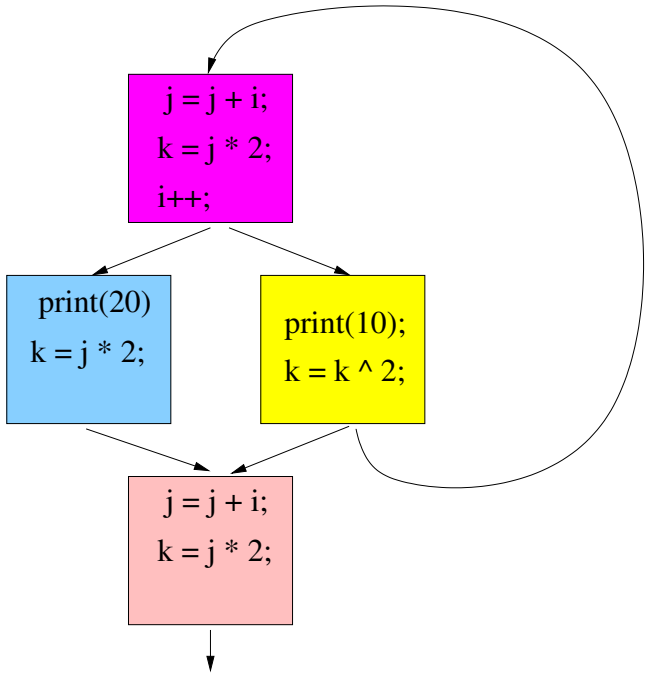
- Weave a hash computation into the control flow:
  1. Compute the hash-value as a side-effect of the real control flow

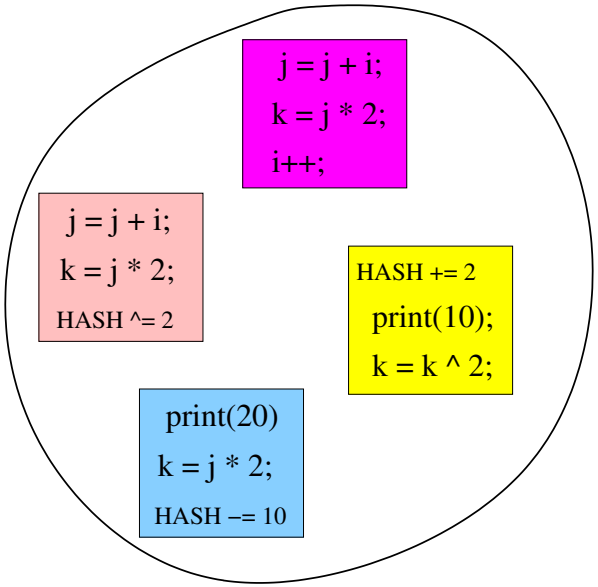




Block 3: Oblivious Hashing Block

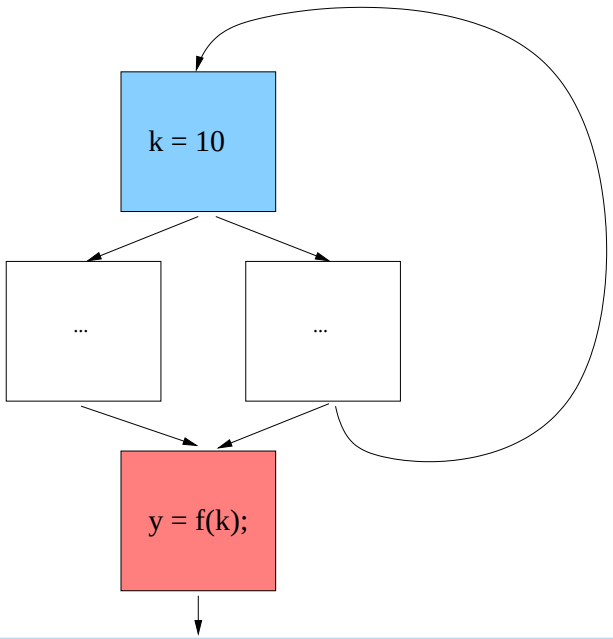
- Weave a hash computation into the control flow:
  1. Compute the hash-value as a side-effect of the real control flow
  2. Compute the hash as the result of challenge input values

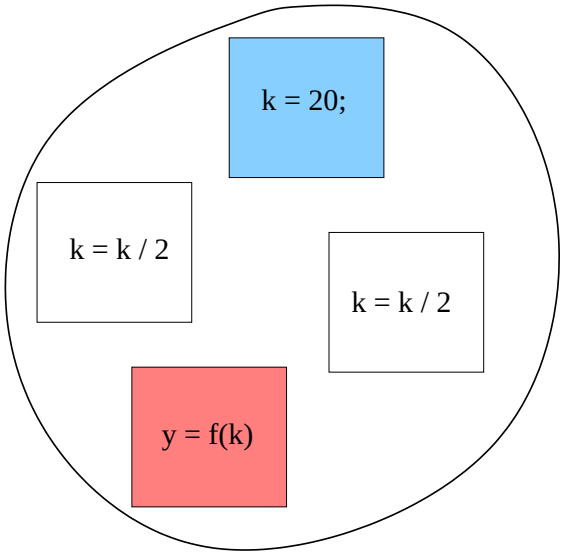




Block 4: Good-block-bad-block

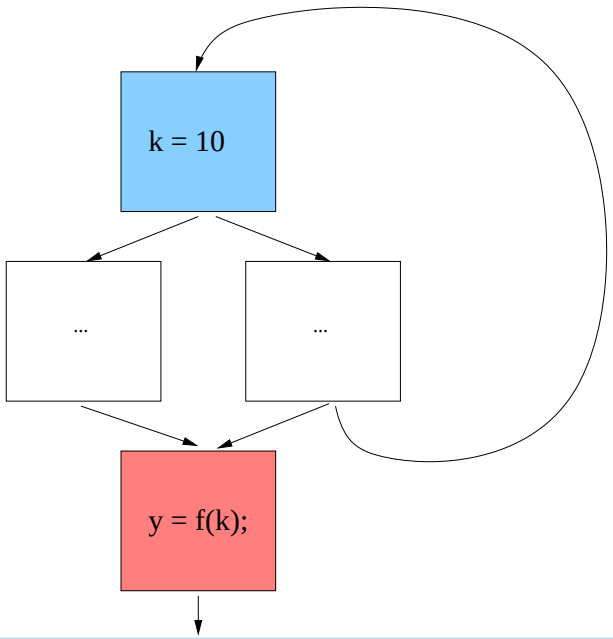
- Send the client a buggy block A and a fixup block B.

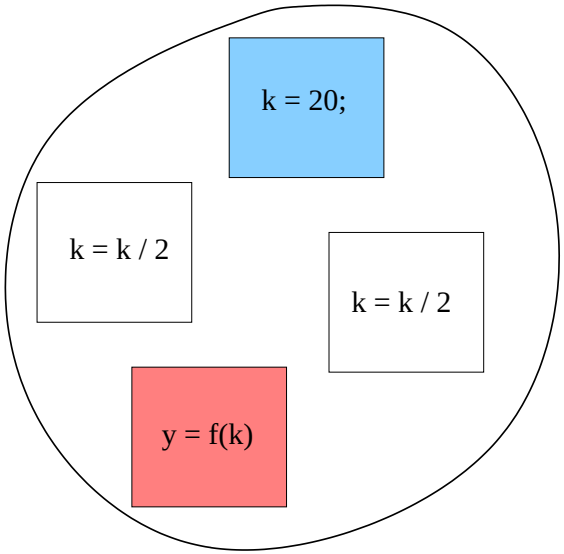




Block 4: Good-block-bad-block

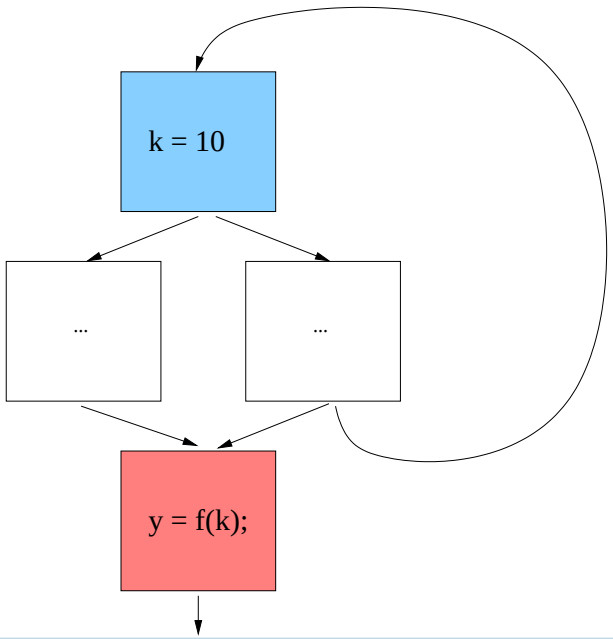
- Send the client a buggy block A and a fixup block B.
- A contains a bug

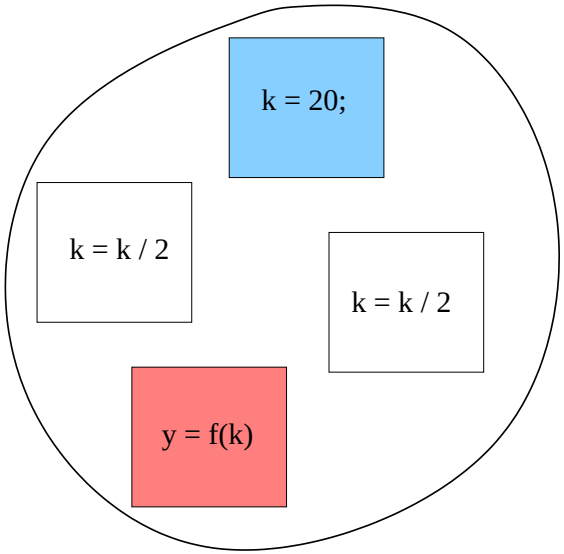




Block 4: Good-block-bad-block

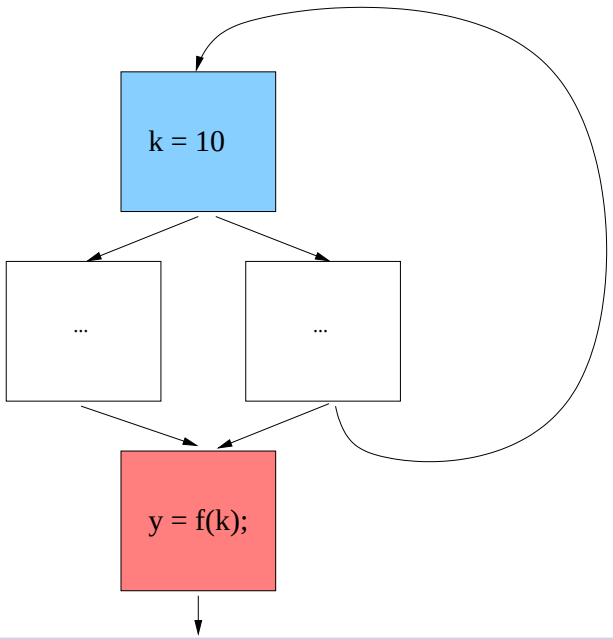
- Send the client a buggy block A and a fixup block B.
- A contains a bug
- Before A's result is used, the fixup block B corrects it.

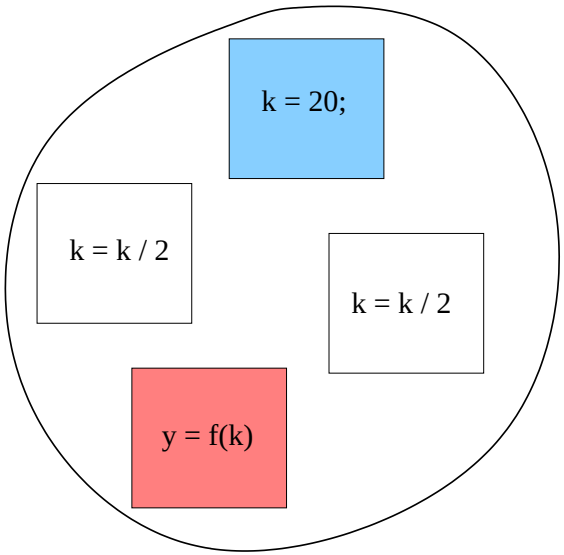




Block 4: Good-block-bad-block

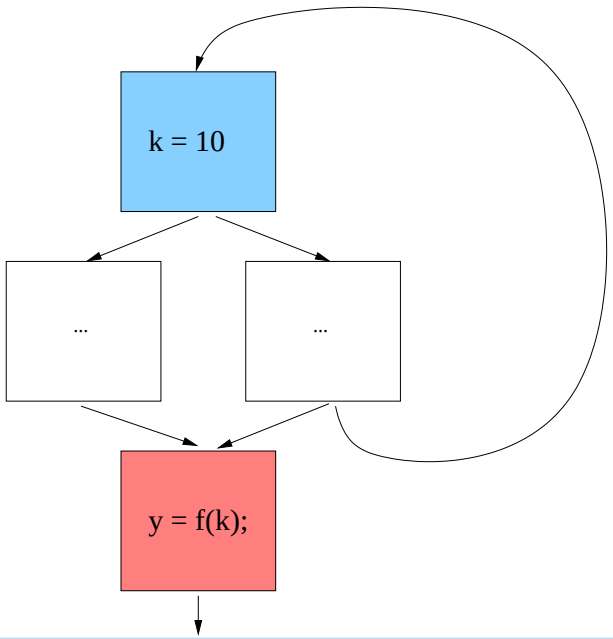
- Send the client a buggy block A and a fixup block B.
- A contains a bug
- Before A's result is used, the fixup block B corrects it.
- The block scheduler arranges for A and B not to be in the client's bag at the same time.

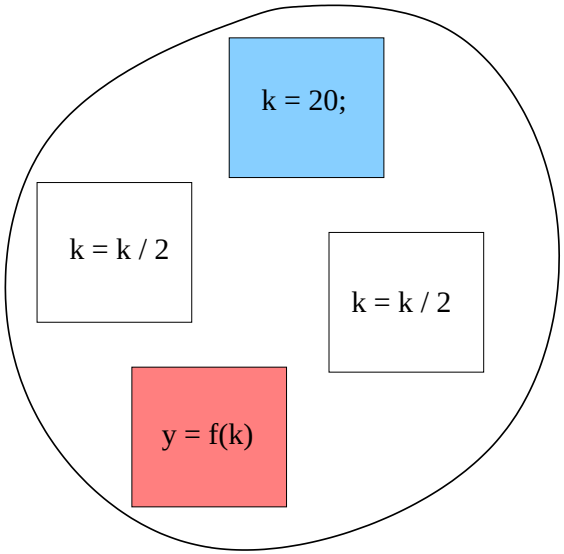




Block 4: Good-block-bad-block

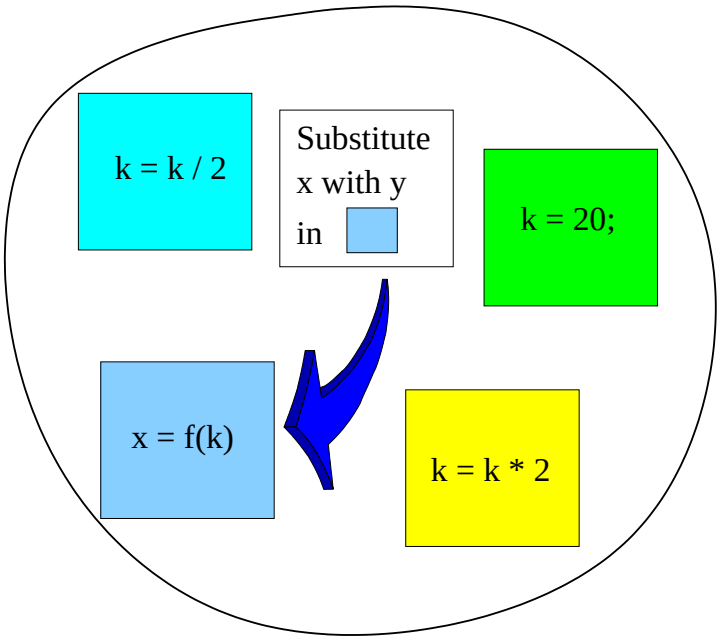
- Send the client a buggy block A and a fixup block B.
- A contains a bug
- Before A's result is used, the fixup block B corrects it.
- The block scheduler arranges for A and B not to be in the client's bag at the same time.
- $\Rightarrow$ Bag-of-blocks is always incorrect.





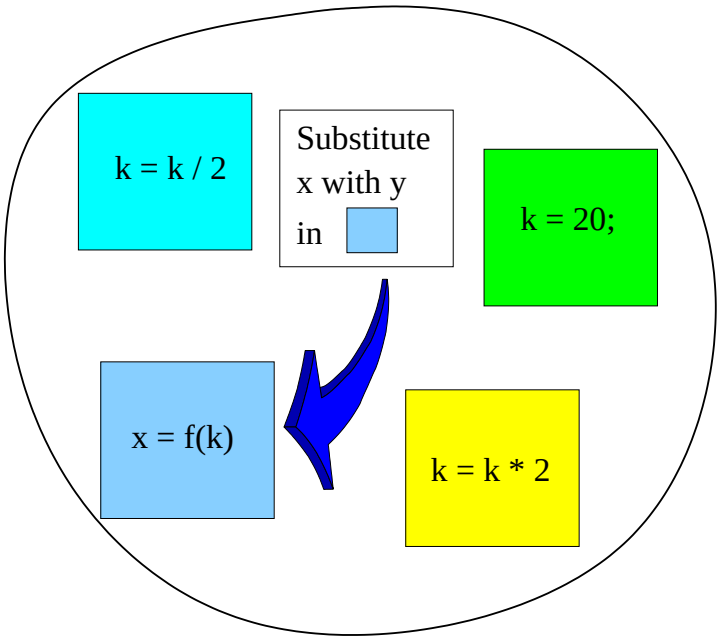
Block 5: Mutation block

- Mini-obfuscator blocks modify blocks in the bag:



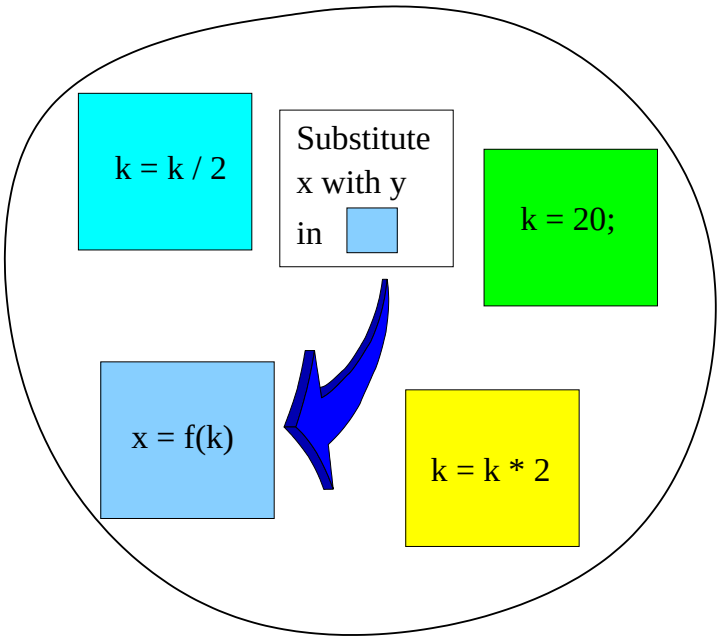
Block 5: Mutation block

- Mini-obfuscator blocks modify blocks in the bag:
  1. introduce bugs into blocks after they've executed,



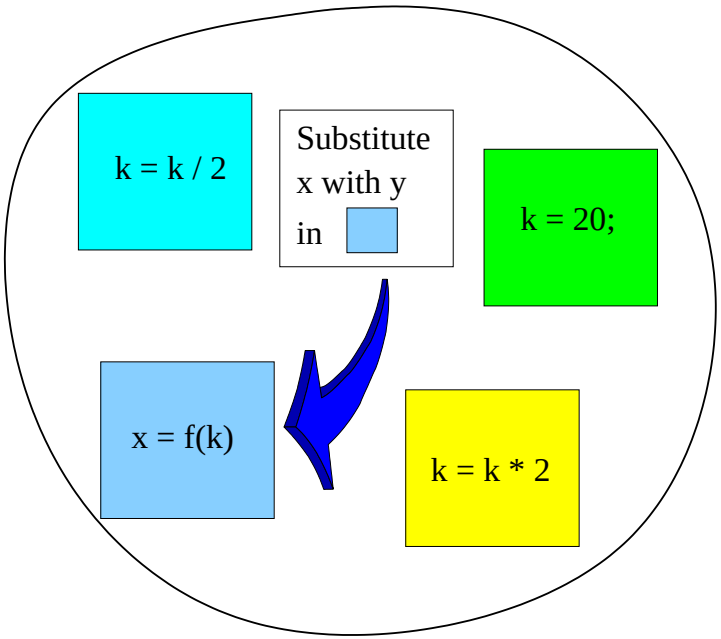
Block 5: Mutation block

- Mini-obfuscator blocks modify blocks in the bag:
  1. introduce bugs into blocks after they've executed,
  2. correct bugs in incorrect blocks before they execute,



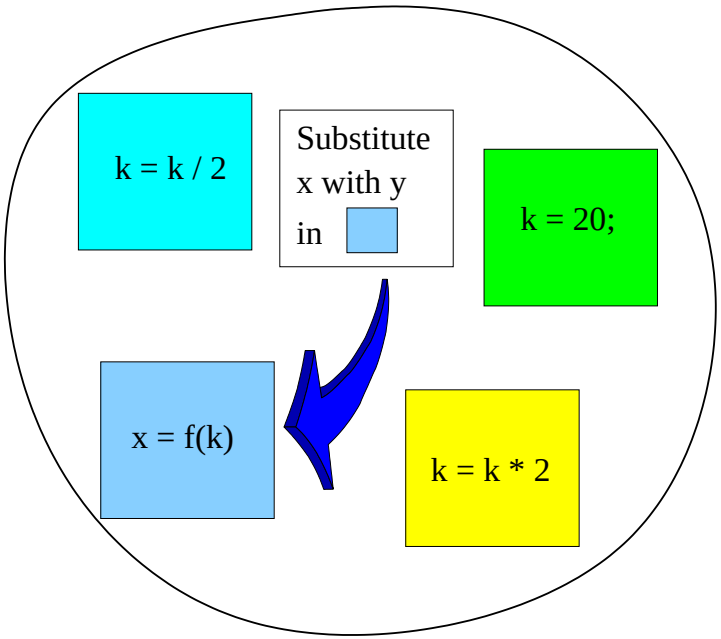
Block 5: Mutation block

- Mini-obfuscator blocks modify blocks in the bag:
  1. introduce bugs into blocks after they've executed,
  2. correct bugs in incorrect blocks before they execute,
  3. transform one block into another block



Block 5: Mutation block

- Mini-obfuscator blocks modify blocks in the bag:
  1. introduce bugs into blocks after they've executed,
  2. correct bugs in incorrect blocks before they execute,
  3. transform one block into another block
- $\Rightarrow$ keeps mutation-rate high and communication-rate low.



Block 6: API Mutation block

- Modifies the client code in the bag to effect a client-server API change.

Block 6: API Mutation block

- Modifies the client code in the bag to effect a client-server API change.
- Example RPC API transformations:

Block 6: API Mutation block

- Modifies the client code in the bag to effect a client-server API change.
- Example RPC API transformations:
  1. rename arguments

Block 6: API Mutation block

- Modifies the client code in the bag to effect a client-server API change.
- Example RPC API transformations:
  1. rename arguments
  2. reorder arguments

Block 6: API Mutation block

- Modifies the client code in the bag to effect a client-server API change.
- Example RPC API transformations:
  1. rename arguments
  2. reorder arguments
  3. add bogus arguments

Block 6: API Mutation block

- Modifies the client code in the bag to effect a client-server API change.
- Example RPC API transformations:
  1. rename arguments
  2. reorder arguments
  3. add bogus arguments
  4. change argument types

Block 6: API Mutation block

- Modifies the client code in the bag to effect a client-server API change.
- Example RPC API transformations:
  1. rename arguments
  2. reorder arguments
  3. add bogus arguments
  4. change argument types
  5. calls can be split in two

Block 6: API Mutation block

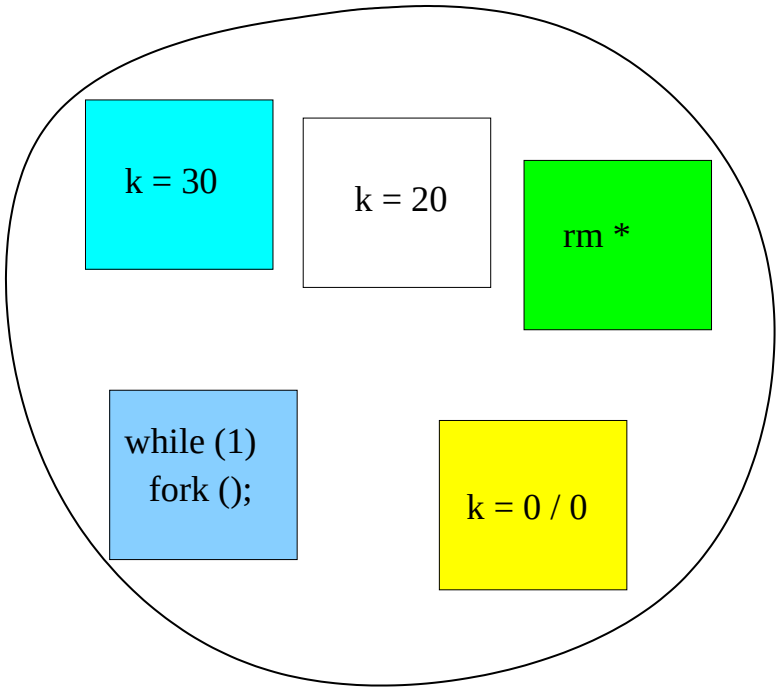
- Modifies the client code in the bag to effect a client-server API change.
- Example RPC API transformations:
  1. rename arguments
  2. reorder arguments
  3. add bogus arguments
  4. change argument types
  5. calls can be split in two
- $\Rightarrow$ the client cannot ignore these blocks if it wants to continue getting service from the server

Block 6: API Mutation block

- Modifies the client code in the bag to effect a client-server API change.
- Example RPC API transformations:
  1. rename arguments
  2. reorder arguments
  3. add bogus arguments
  4. change argument types
  5. calls can be split in two
- $\Rightarrow$ the client cannot ignore these blocks if it wants to continue getting service from the server
- $\Rightarrow$ makes it hard for client to ignore *any* blocks.

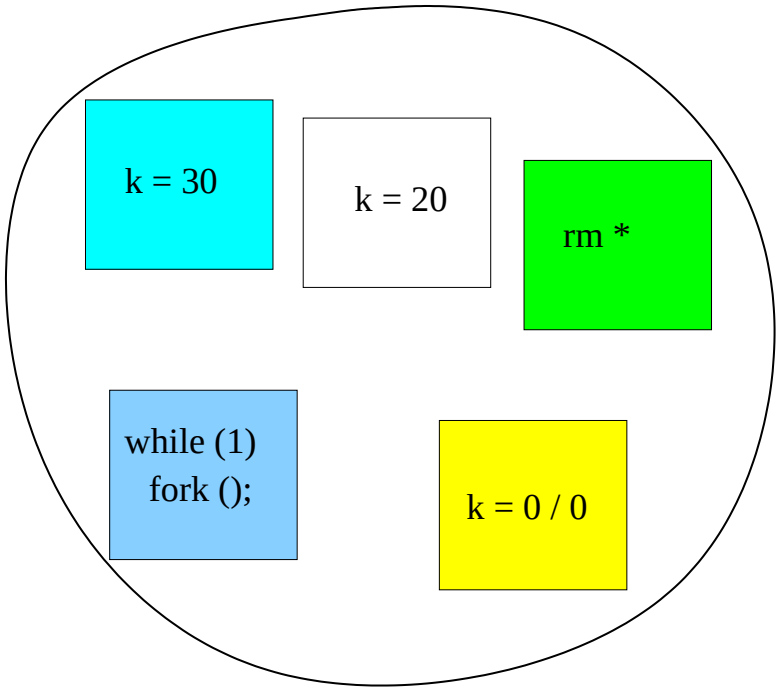
Block 7: Unexecutable block

- This type of block will, if executed, cause the program to crash or otherwise malfunction.



Block 7: Unexecutable block

- This type of block will, if executed, cause the program to crash or otherwise malfunction.
- $\Rightarrow$ makes it harder for the adversary to execute blocks in isolation to experimentally figure out what each one does.



Block 8: Server-side block

- A server-side block passes its arguments to the server and invokes the "real" block on the server.

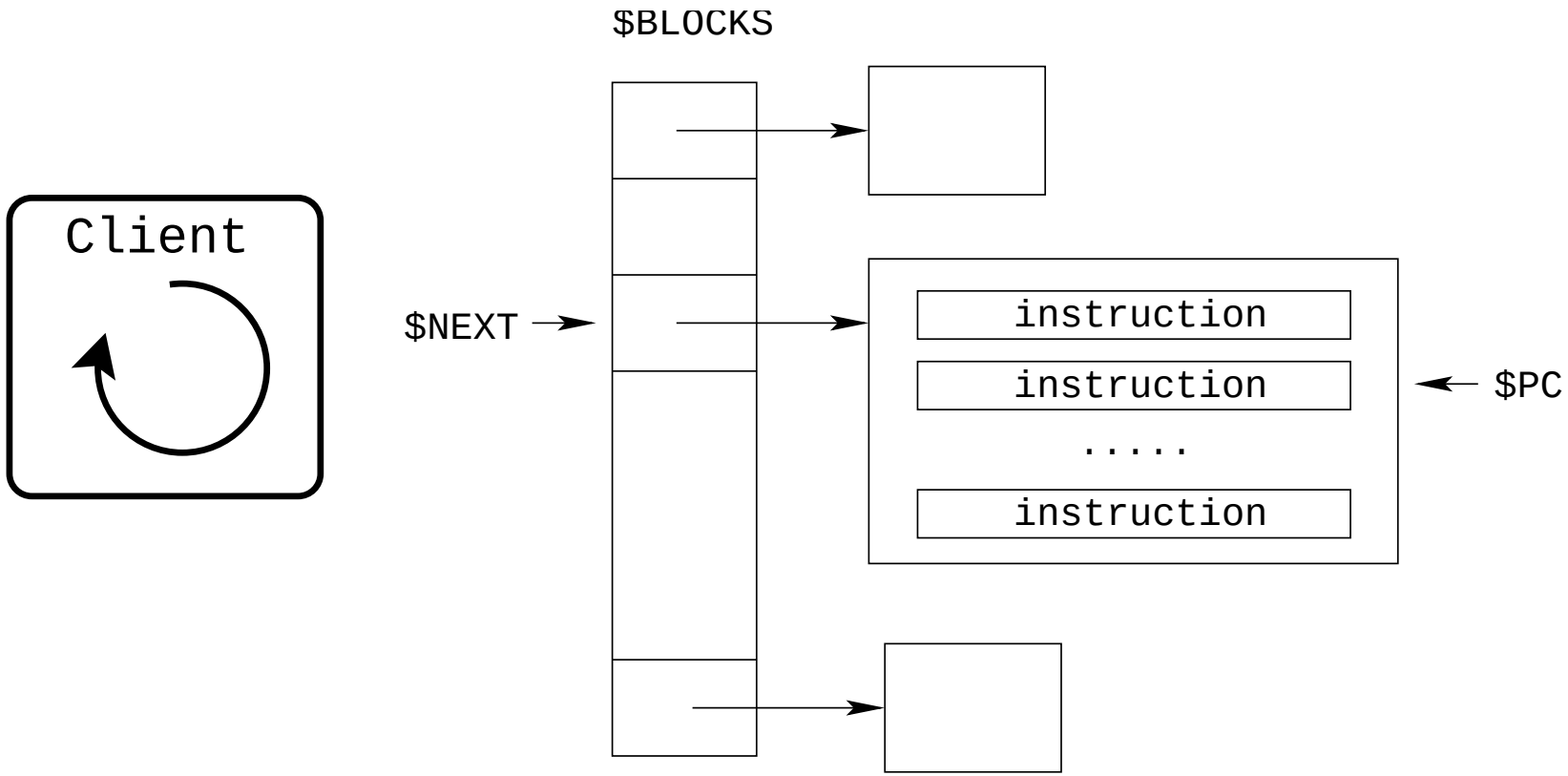
Block 8: Server-side block

- A server-side block passes its arguments to the server and invokes the "real" block on the server.
- $\Rightarrow$ bag-of-blocks is always incomplete

One Block to rule them all...

- We may be able to get away with just one block kind, provided we allow blocks to access the VM itself.
- The VM consists of the bag of blocks (`$BLOCKS`), the next block to execute (`$NEXT`), the current location (`$PC`), and the interpretation loop.
- We can even get away without an actual interpretation loop by threading the blocks, i.e. as its last instruction, each block jumps to the next block.

One Block to find them...



One Block to bring them all...

```
$BLOCKS[84][3] = 'load v' % Modify another block
kill x = x + 1
$BLOCKS[56] = nil         % Invalidate block 56
y = y - 3
$BLOCKS[84][3] = 'load v' % Modify another block
p(5)
$NEXT = 82                % Next block to execute
z = z + 1
$GET(88)                  % Hint: we may need block 88 soon
```

. . . and in the darkness bind them!

- We have two options as to how we should obfuscate the blocks:
- We could try to make the blocks as similar as possible: padded to the same size, obfuscated to have the same structure ("every block contains exactly one loop and one method call"), etc. This increases stealth and makes it hard for the attacker to know which blocks it can ignore and which it has to execute.
- Or, we could try to make the blocks as different as possible to make analysis of the blocks as hard as possible.

Summary

- We have several knobs to tweek:

Summary

- We have several knobs to tweek:
  1. size of the client's bag

Summary

- We have several knobs to tweek:
  1. size of the client's bag
  2. rate of block push

Summary

- We have several knobs to tweek:
  1. size of the client's bag
  2. rate of block push
  3. frequency of each kind of block

Summary

- We have several knobs to tweek:
  1. size of the client's bag
  2. rate of block push
  3. frequency of each kind of block
  4. level of obfuscation of each block

Summary

- We have several knobs to tweek:
  1. size of the client's bag
  2. rate of block push
  3. frequency of each kind of block
  4. level of obfuscation of each block
  5. block size (basic block, function, module)

Summary

- We have several knobs to tweek:
  1. size of the client's bag
  2. rate of block push
  3. frequency of each kind of block
  4. level of obfuscation of each block
  5. block size (basic block, function, module)
- Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia, *Dynamo: a transparent dynamic optimization system*, PLDI 2000: 1-12