A Design and Verification Methodology for Secure Isolated Regions

Rohit Sinha[1], Manuel Costa[2], Akash Lal[2], Nuno P. Lopes[2],

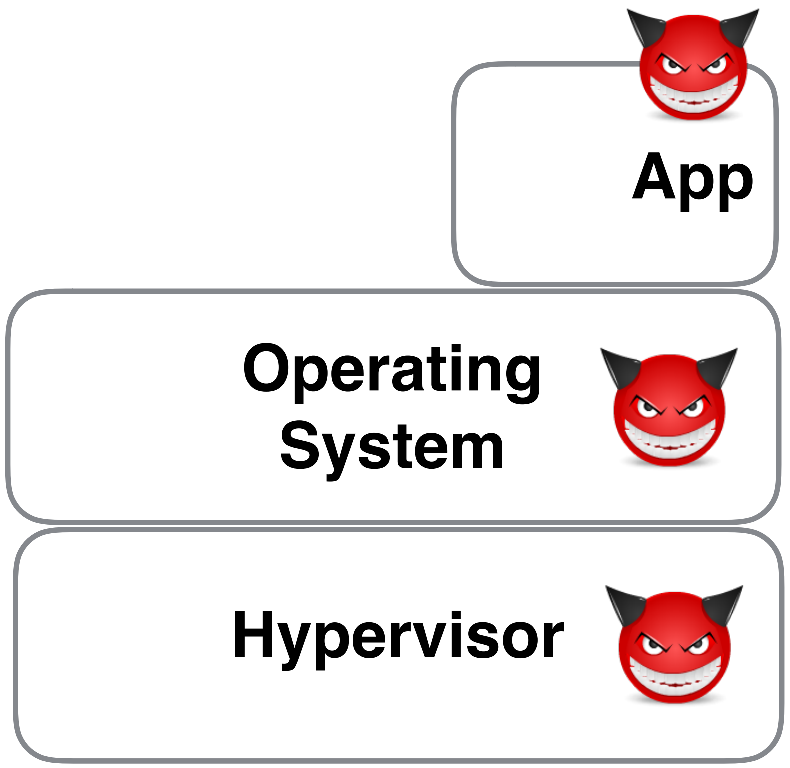
Sriram Rajamani[2], Sanjit A. Seshia[1], Kapil Vaswani[2]

Berkeley UNIVERSITY OF CALIFORNIA [1]   Microsoft Research [2]

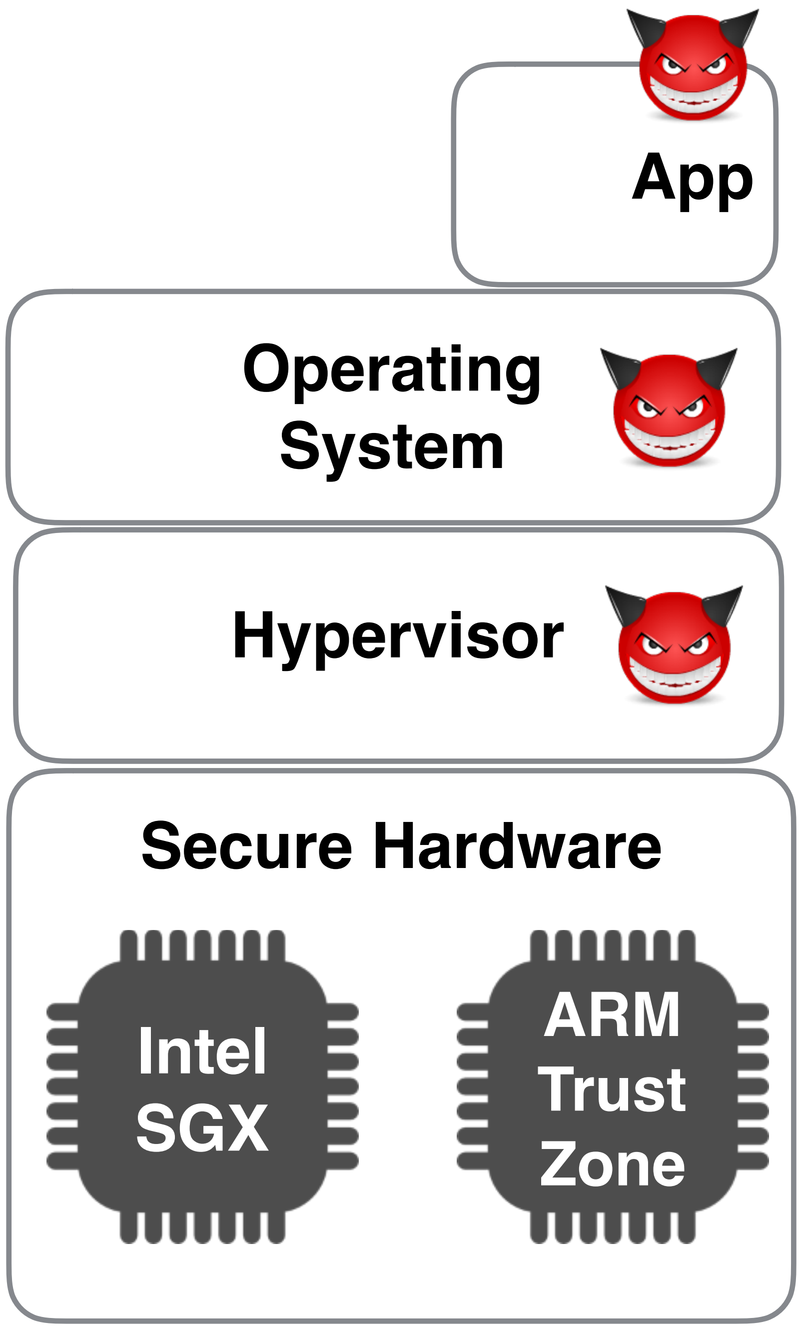
PLDI 2016

Secure Isolated Regions (SIR)

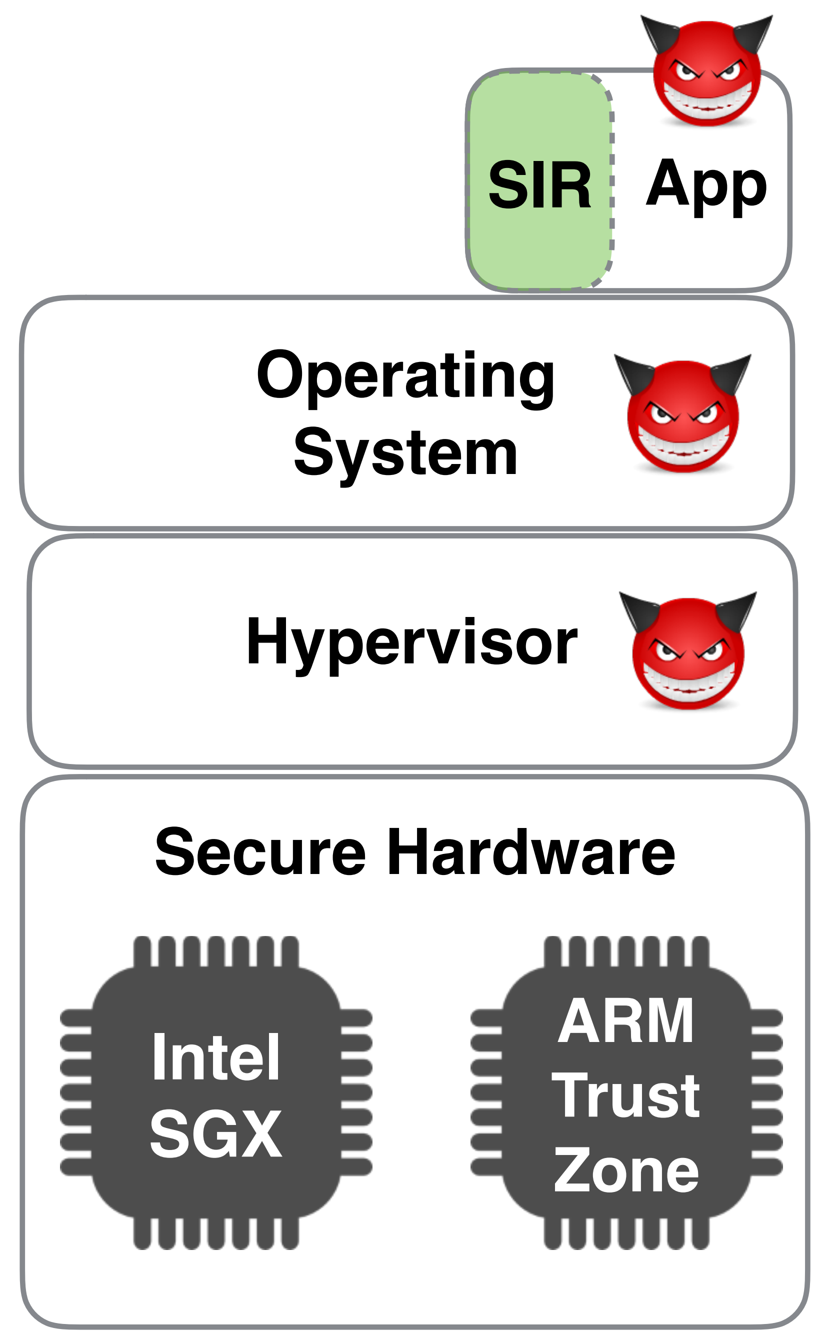
Secure Isolated Regions (SIR)



Secure Isolated Regions (SIR)

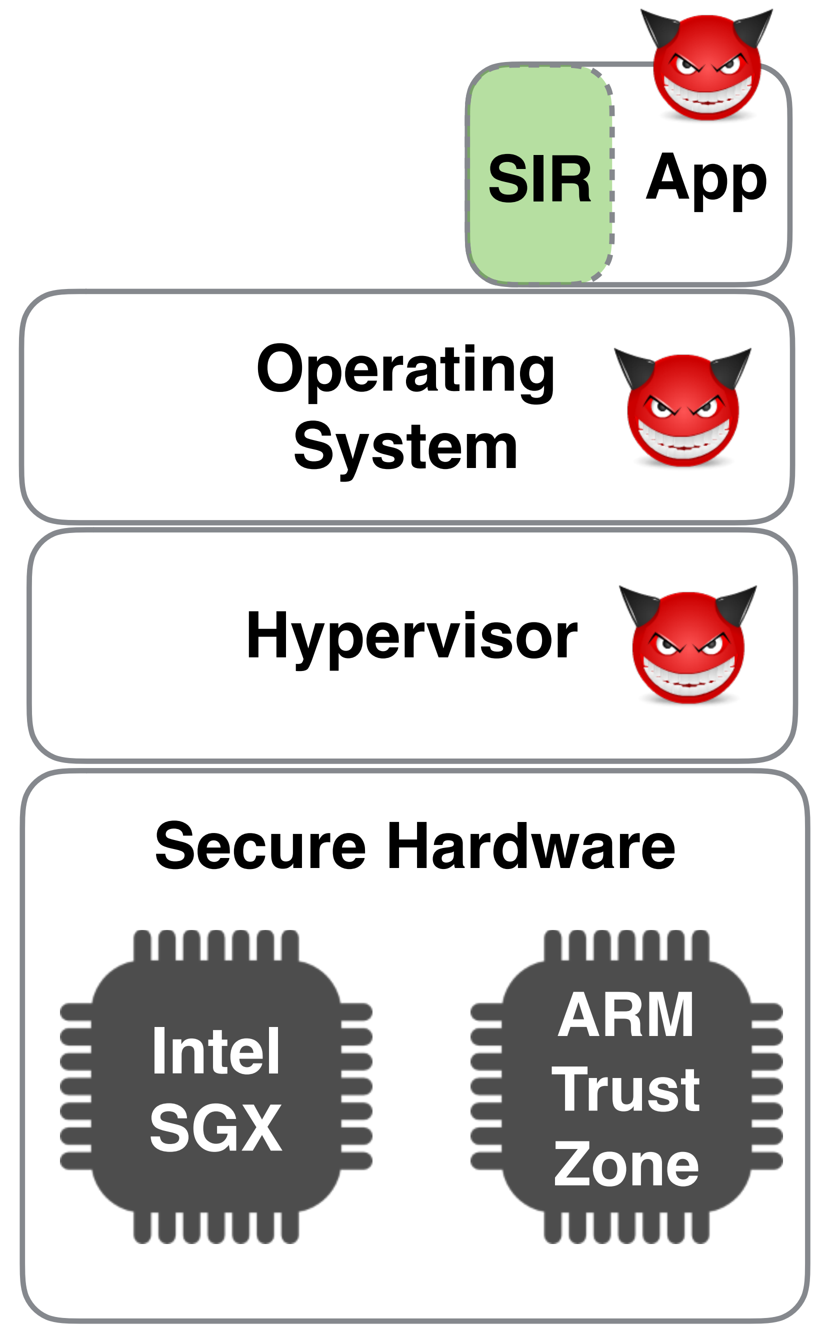


Secure Isolated Regions (SIR)



SIR memory is protected: only SIR code can access it

Secure Isolated Regions (SIR)



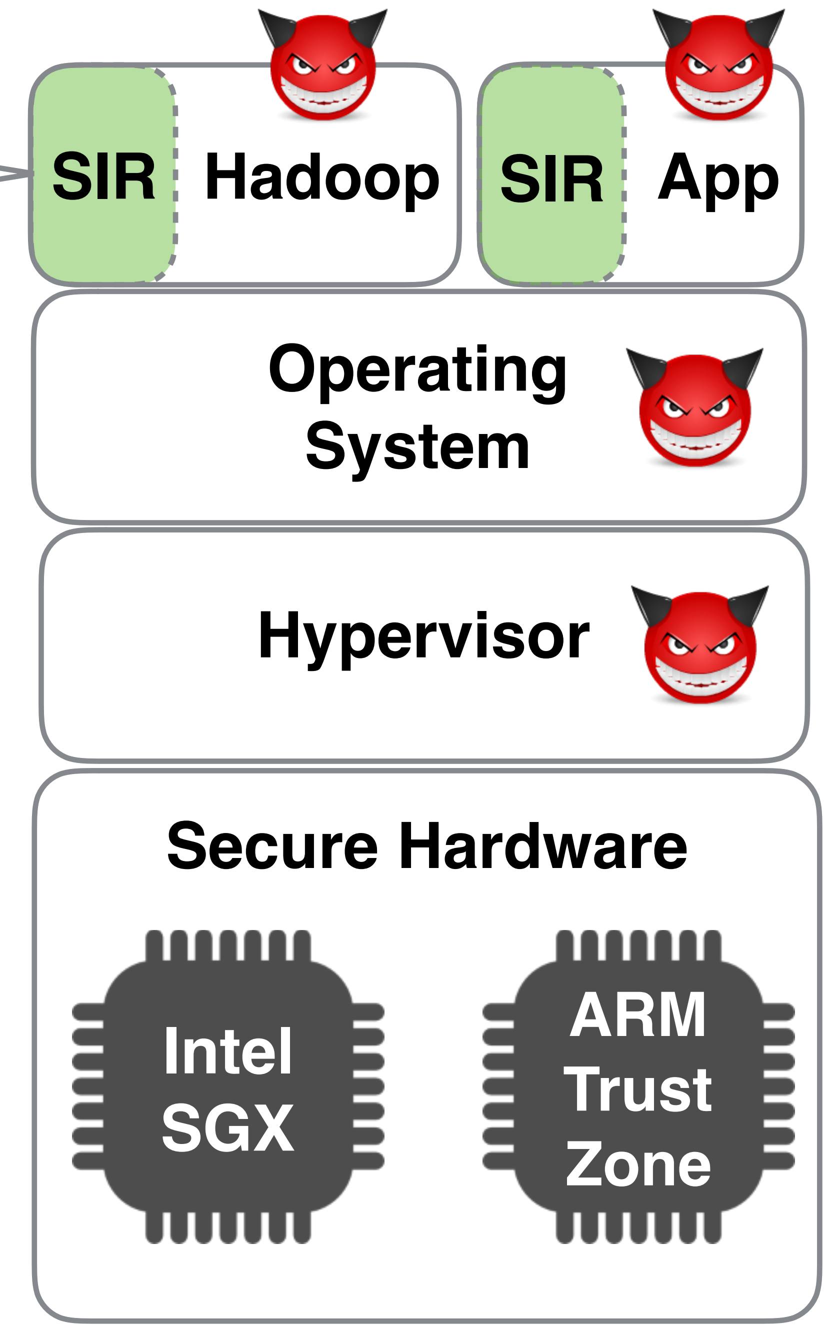
SIR memory is protected: only SIR code can access it

Trusted Computing Base includes the SIR and CPU hardware

Secure Isolated Regions (SIR)

```
void map(...)
{ /* compute on sensitive data */ }

void reduce(...)
{ /* compute on sensitive data */ }
```

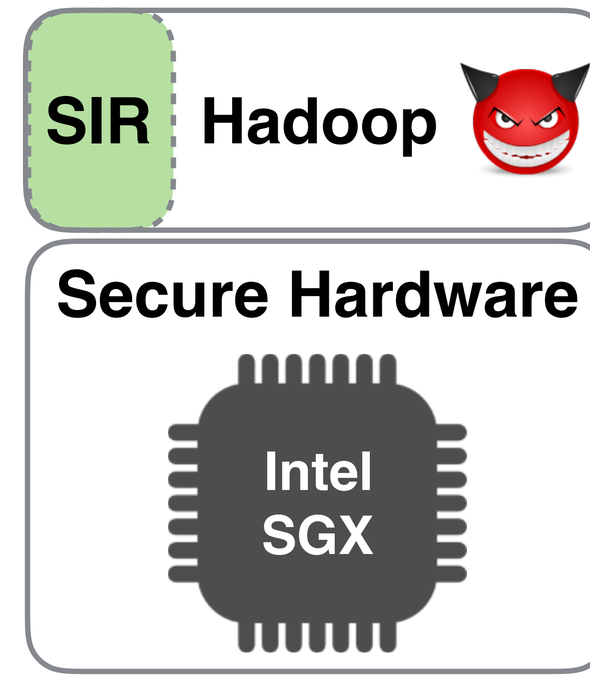
VC3: Trustworthy Data Analytics in the Cloud [Schuster et. al., *S&P'15*]



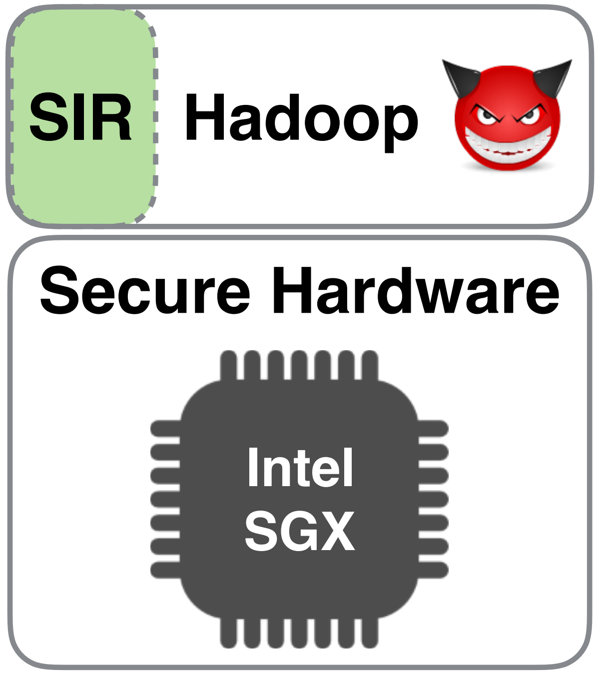
SIR memory is protected: only SIR code can access it

Trusted Computing Base includes the SIR and CPU hardware

Bugs in SIRs can be Exploited

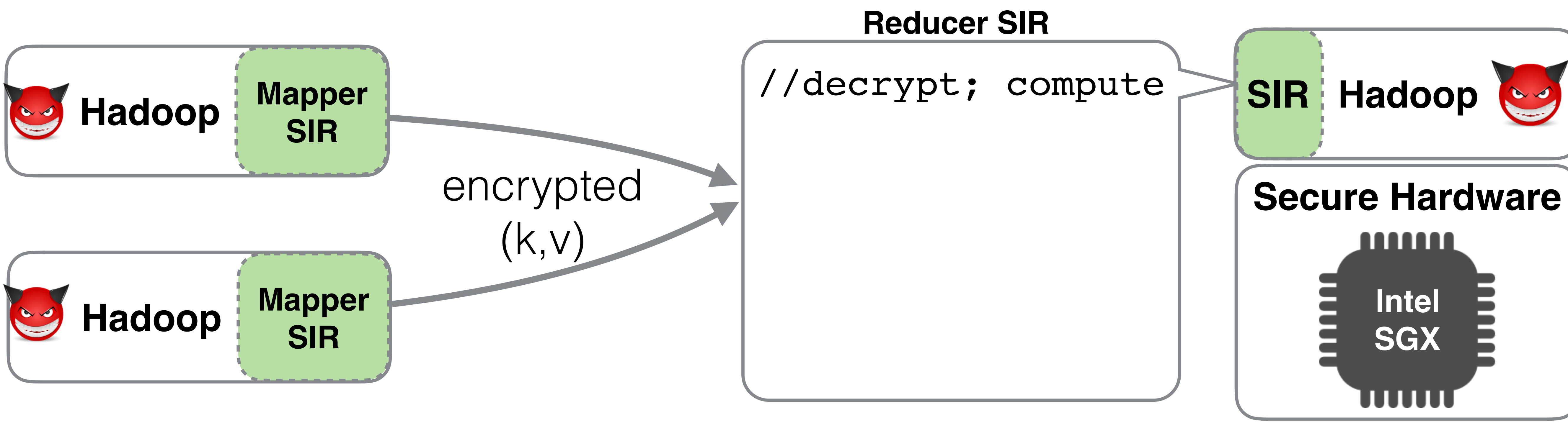


Bugs in SIRs can be Exploited



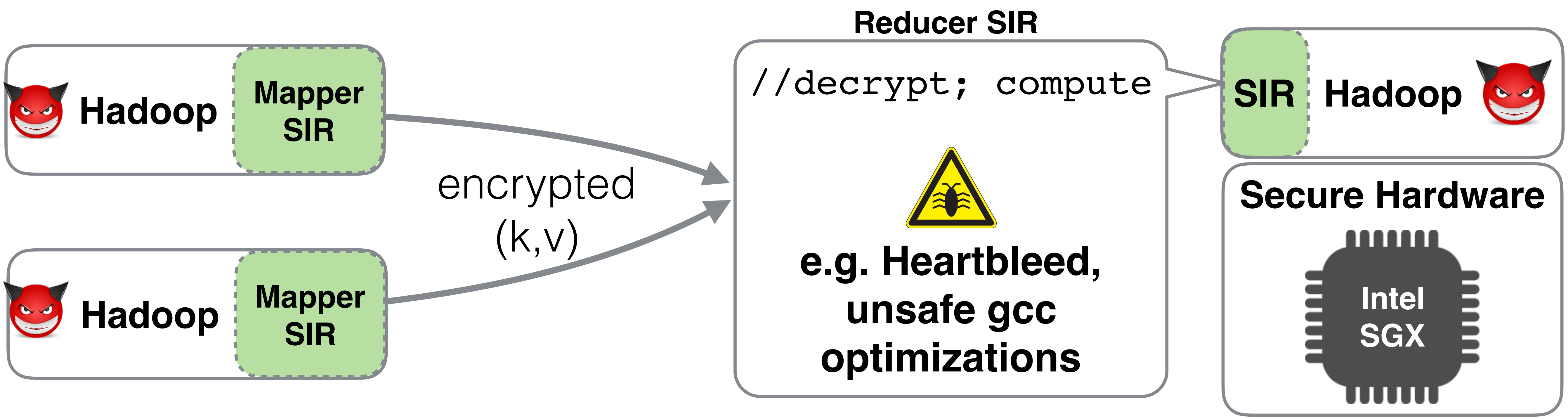
SIR accesses untrusted Hadoop's memory to perform I/O

Bugs in SIRs can be Exploited



SIR accesses untrusted Hadoop's memory to perform I/O

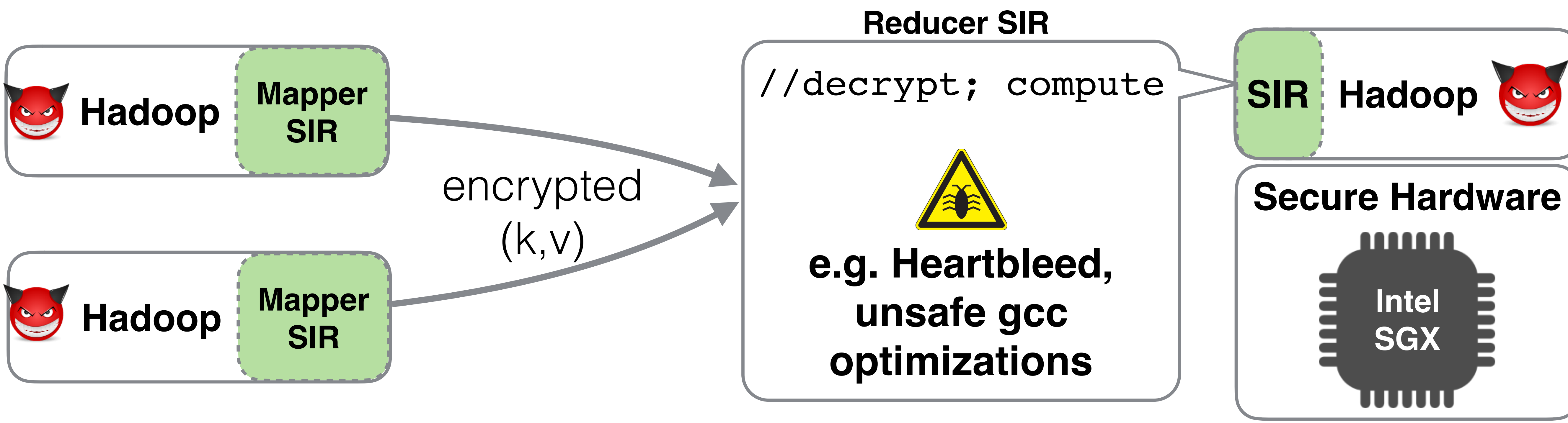
Bugs in SIRs can be Exploited



SIR accesses untrusted Hadoop's memory to perform I/O

Adversary can exploit SIRs using I/O interactions

Bugs in SIRs can be Exploited



SIR accesses untrusted Hadoop's memory to perform I/O

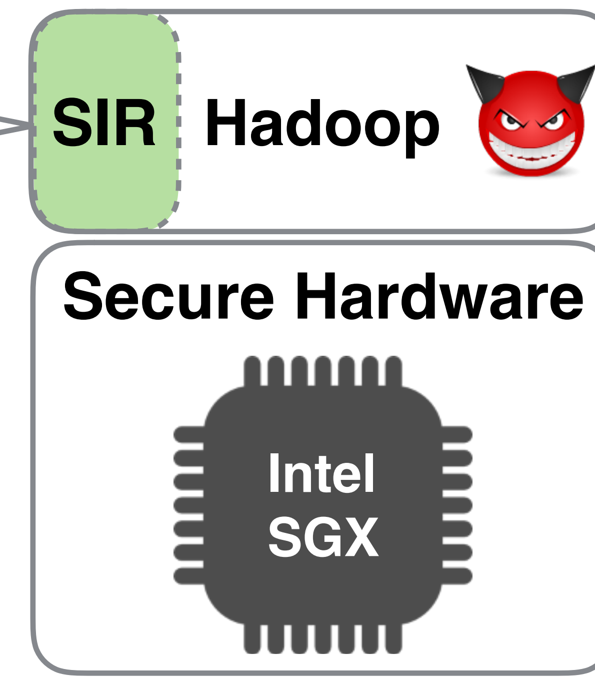
Adversary can exploit SIRs using I/O interactions

Our goal: ensure that secrets are not leaked (confidentiality) in the presence of programming errors and compiler bugs

Challenges in Proving Confidentiality

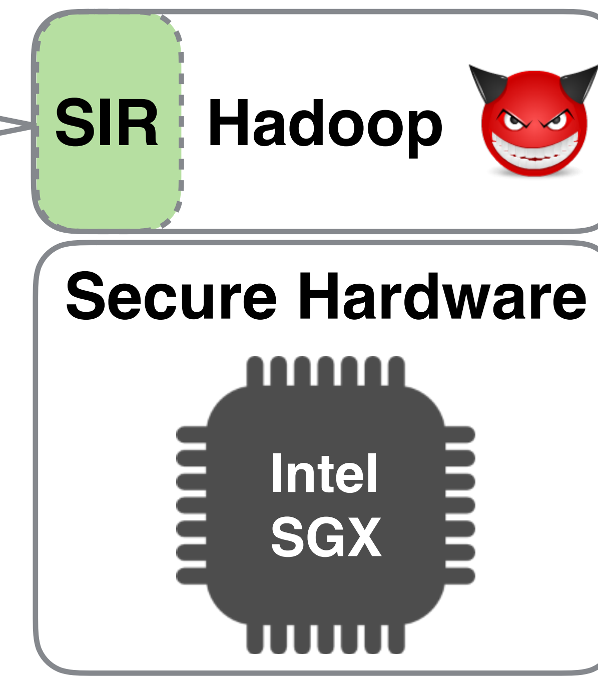
Challenges in Proving Confidentiality

```
void Reduce(byte *kEnc, byte *vEnc)
{
```



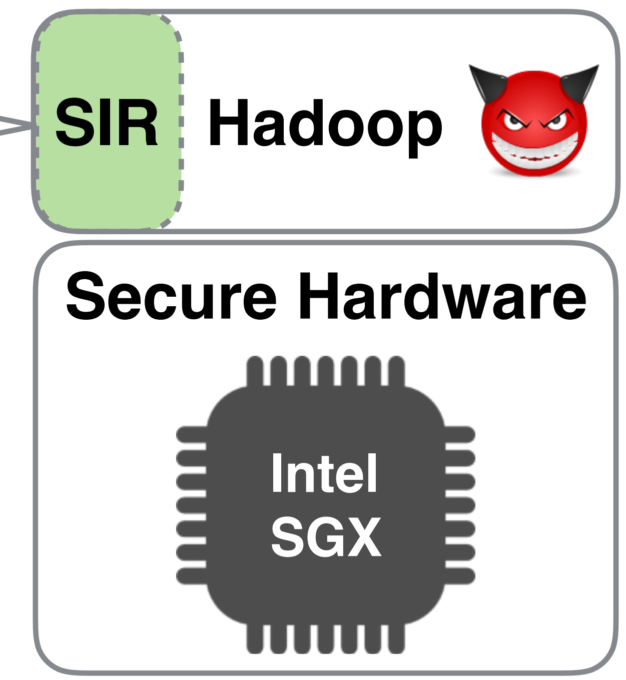
Challenges in Proving Confidentiality

```
void Reduce(byte *kEnc, byte *vEnc)
{
  KeyAesGcm *aesKey = ProvisionKey();
```



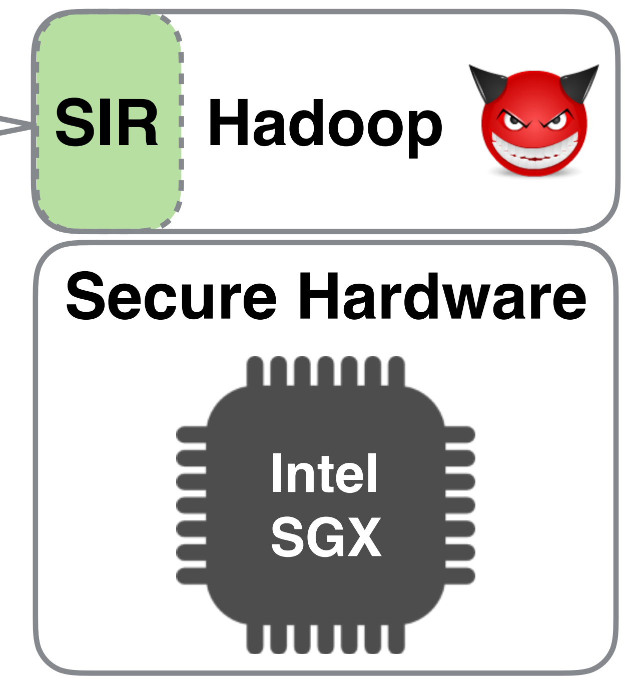
Challenges in Proving Confidentiality

```
void Reduce(byte *kEnc, byte *vEnc)
{
  KeyAesGcm *aesKey = ProvisionKey();

  char k[..];
  aesKey->Decrypt(kEnc, k);
  char v[..];
  aesKey->Decrypt(vEnc, v);
```



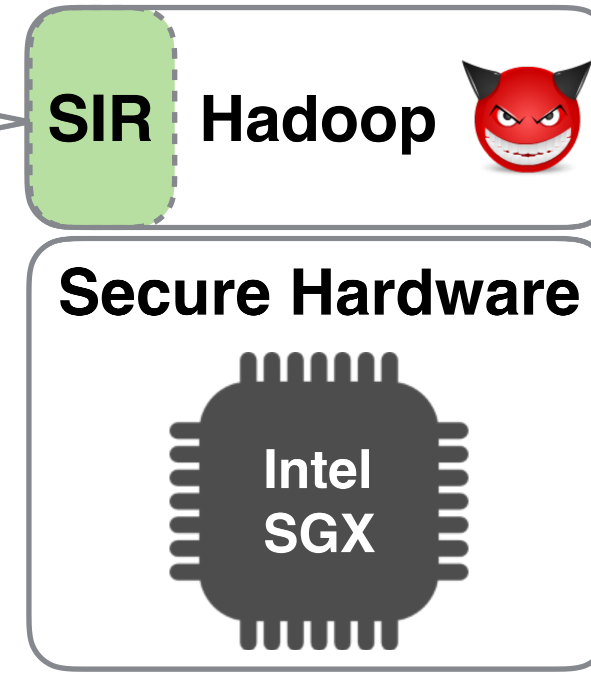
Challenges in Proving Confidentiality

```
void Reduce(byte *kEnc, byte *vEnc)
{
  KeyAesGcm *aesKey = ProvisionKey();

  char k[..];
  aesKey->Decrypt(kEnc, k);
  char v[..];
  aesKey->Decrypt(vEnc, v);
  long sum = compute_sum(v);
```



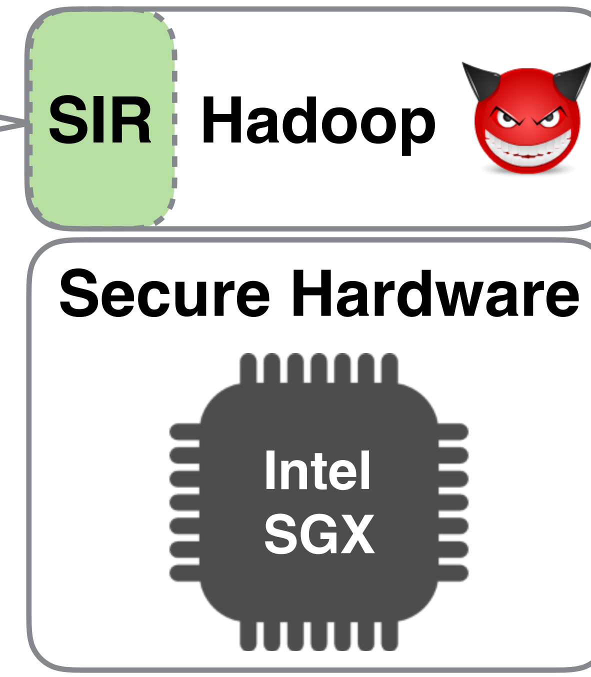
Challenges in Proving Confidentiality

```
void Reduce(byte *kEnc, byte *vEnc)
{
  KeyAesGcm *aesKey = ProvisionKey();

  char k[..];
  aesKey->Decrypt(kEnc, k);
  char v[..];
  aesKey->Decrypt(vEnc, v);
  long sum = compute_sum(v);

  char cleartext[..];
  sprintf(cleartext, “%s %lld”, k, sum);
```



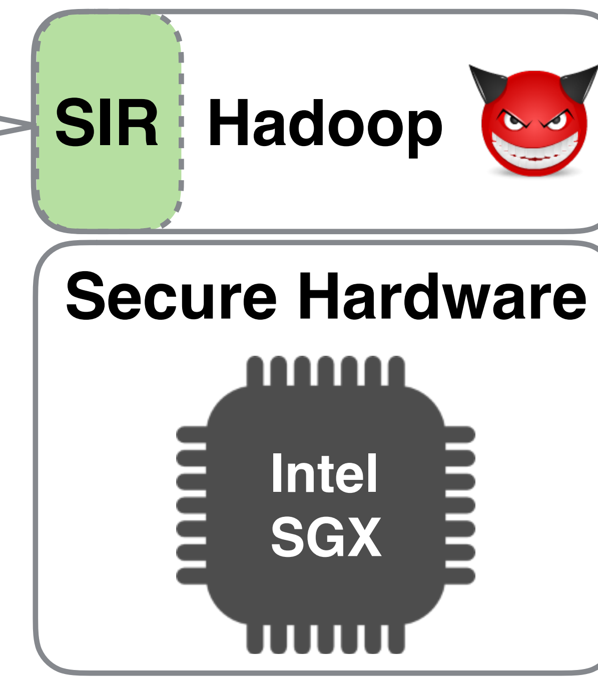
Challenges in Proving Confidentiality

```
void Reduce(byte *kEnc, byte *vEnc)
{
  KeyAesGcm *aesKey = ProvisionKey();

  char k[..];
  aesKey->Decrypt(kEnc, k);
  char v[..];
  aesKey->Decrypt(vEnc, v);
  long sum = compute_sum(v);

  char cleartext[..];
  sprintf(cleartext, "%s %lld", k, sum);
  aesKey->Encrypt(cleartext,
                  untrusted_memory,
                  BUF_SIZE);
}
```



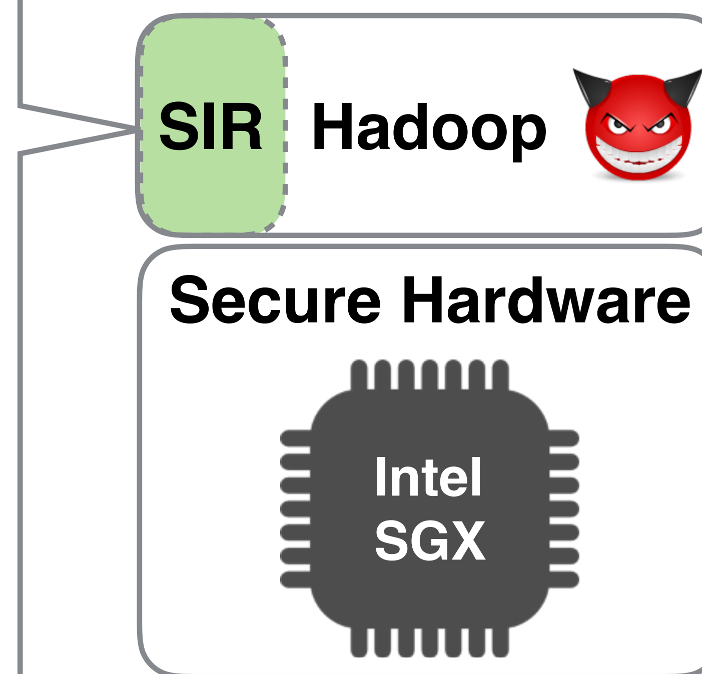
Challenges in Proving Confidentiality

```
void Reduce(byte *kEnc, byte *vEnc)
{
  KeyAesGcm *aesKey = ProvisionKey();

  char k[..];
  aesKey->Decrypt(kEnc, k);
  char v[..];
  aesKey->Decrypt(vEnc, v);
  long sum = compute_sum(v);

  char cleartext[..];
  sprintf(cleartext, "%s %lld", k, sum);
  aesKey->Encrypt(cleartext,
                  untrusted_memory,
                  BUF_SIZE);
}
```



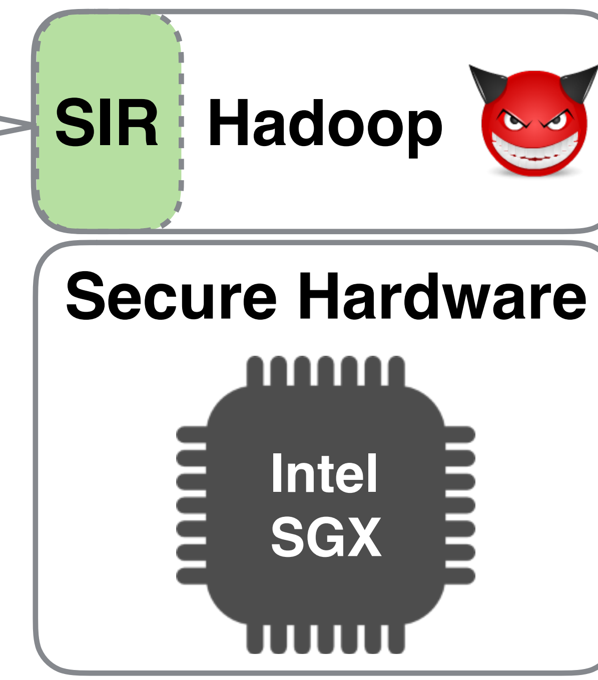
Challenges in Proving Confidentiality

```
void Reduce(byte *kEnc, byte *vEnc)
{
  KeyAesGcm *aesKey = ProvisionKey();

  char k[..];
  aesKey->Decrypt(kEnc, k);
  char v[..];
  aesKey->Decrypt(vEnc, v);
  long sum = compute_sum(v);

  char cleartext[..];
  sprintf(cleartext, “%s %lld”, k, sum);
  aesKey->Encrypt(cleartext,
                  untrusted_memory,
                  BUF_SIZE);
}
```



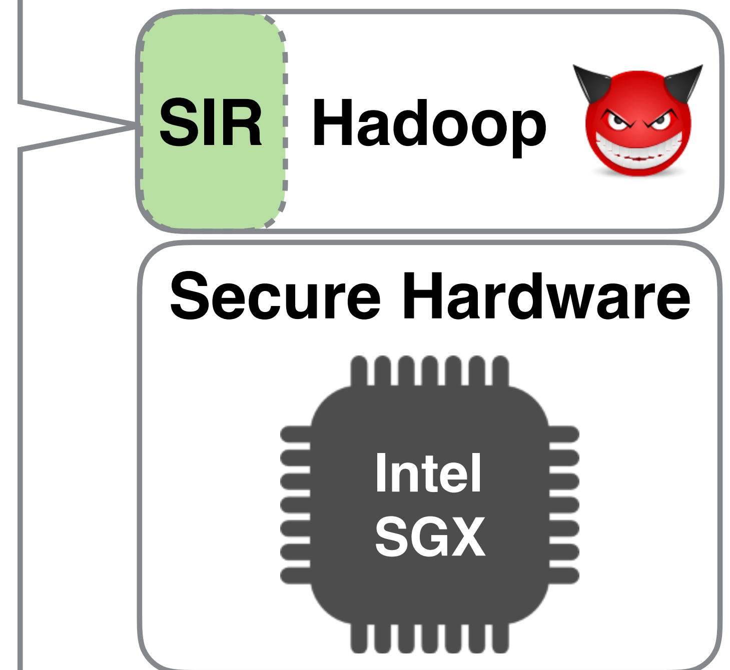
Challenges in Proving Confidentiality

```
void Reduce(byte *kEnc, byte *vEnc)
{
  KeyAesGcm *aesKey = ProvisionKey();

  char k[..];
  aesKey->Decrypt(kEnc, k);
  char v[..];
  aesKey->Decrypt(vEnc, v);
  long sum = compute_sum(v);

  char cleartext[..];
  sprintf(cleartext, "%s %lld", k, sum);
  aesKey->Encrypt(cleartext,
                  untrusted_memory,
                  BUF_SIZE);
}
```



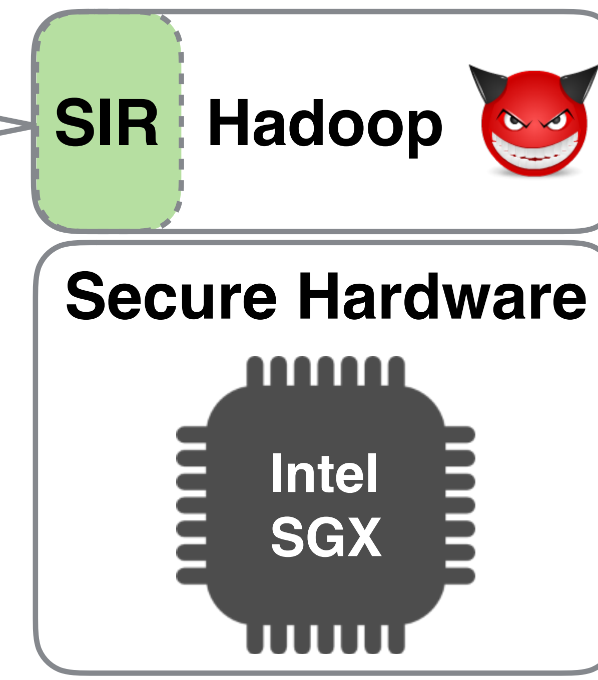
Challenges in Proving Confidentiality

```
void Reduce(byte *kEnc, byte *vEnc)
{
  KeyAesGcm *aesKey = ProvisionKey();

  char k[..];
  aesKey->Decrypt(kEnc, k);
  char v[..];
  aesKey->Decrypt(vEnc, v);
  long sum = compute_sum(v);

  char cleartext[..];
  sprintf(cleartext, "%s %lld", k, sum);
  aesKey->Encrypt(cleartext,
                  untrusted_memory,
                  BUF_SIZE);
}
```



Challenges in Proving Confidentiality

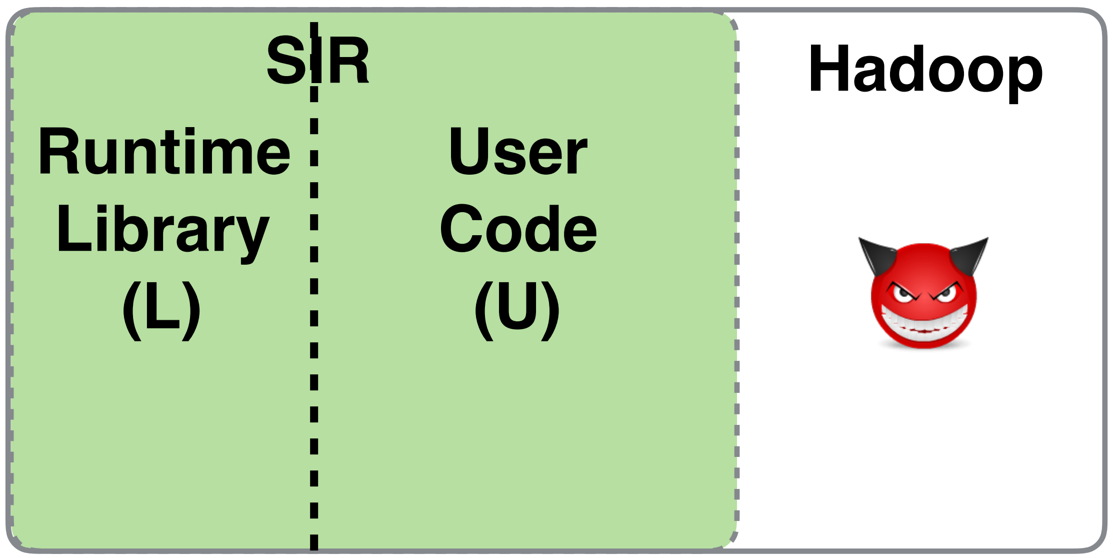
```
void Reduce(byte *kEnc, byte *vEnc)
{
  KeyAesGcm *aesKey = ProvisionKey();

  char k[..];
  aesKey->Decrypt(kEnc, k);
  char v[..];
  aesKey->Decrypt(vEnc, v);
  long sum = compute_sum(v);

  char cleartext[..];
  sprintf(cleartext, “%s %lld”, k, sum);
  aesKey->Encrypt(cleartext,
                  untrusted_memory,
                  BUF_SIZE);
}
```



compiler

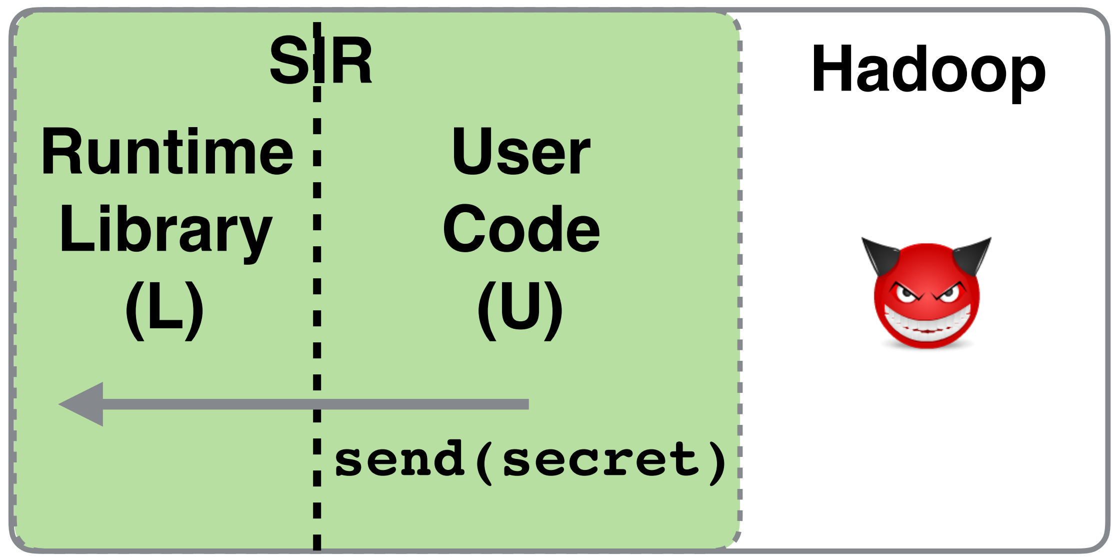
Information Release Confinement

Information Release Confinement



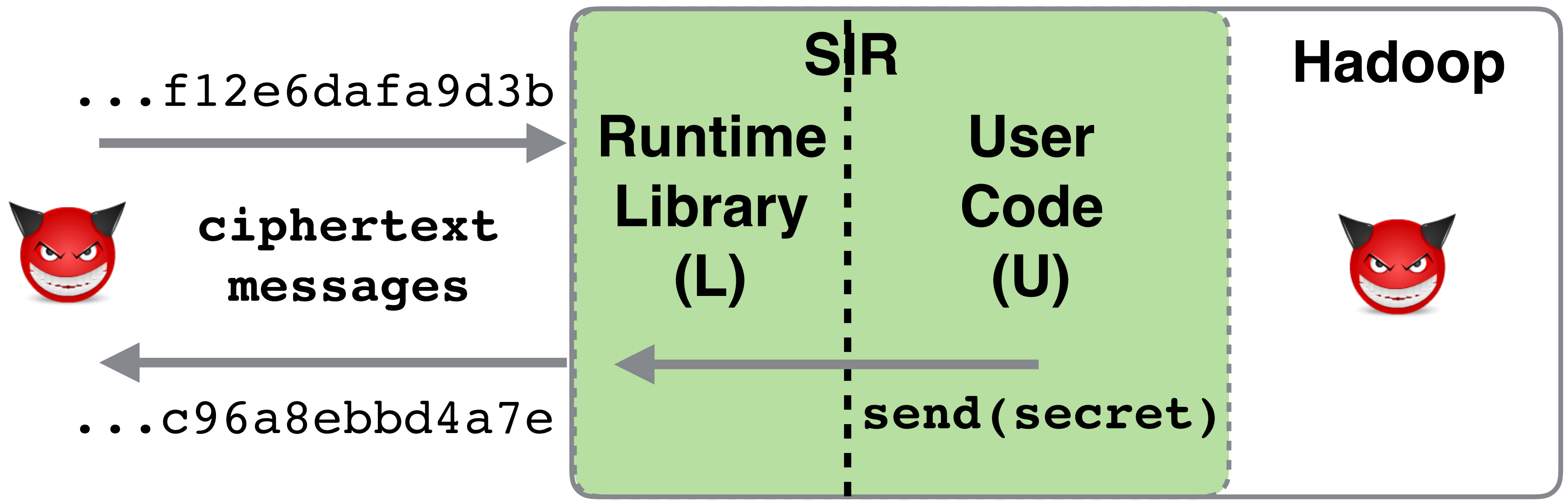
L implements **send**, **recv**, malloc, free

Information Release Confinement



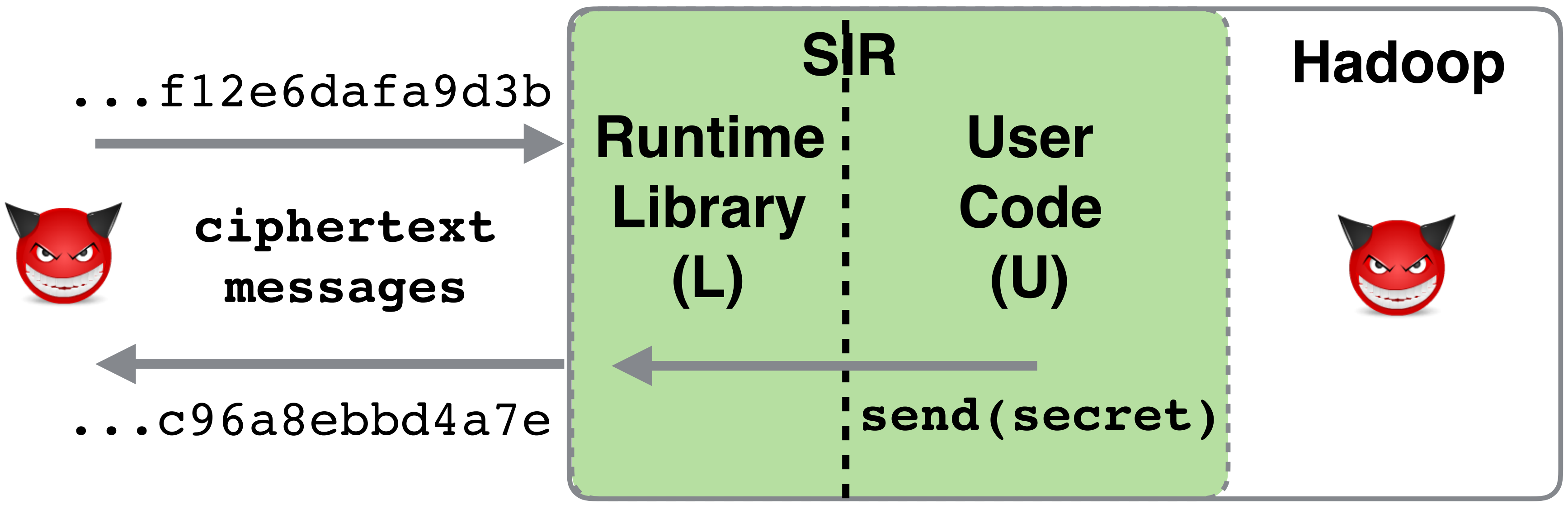
L implements **send**, **recv**, malloc, free

Information Release Confinement



L implements **send**, **recv**, malloc, free

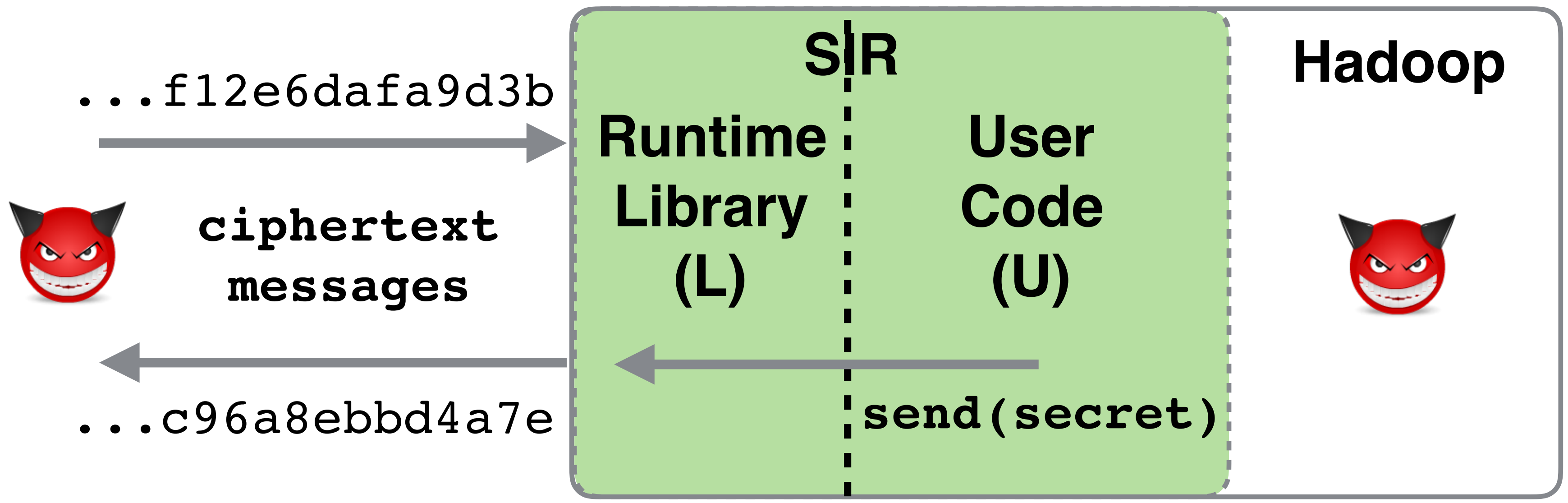
Information Release Confinement



L implements **send**, **recv**, malloc, free

IRC: All updates to non-SIR memory via L's `send` API

Information Release Confinement

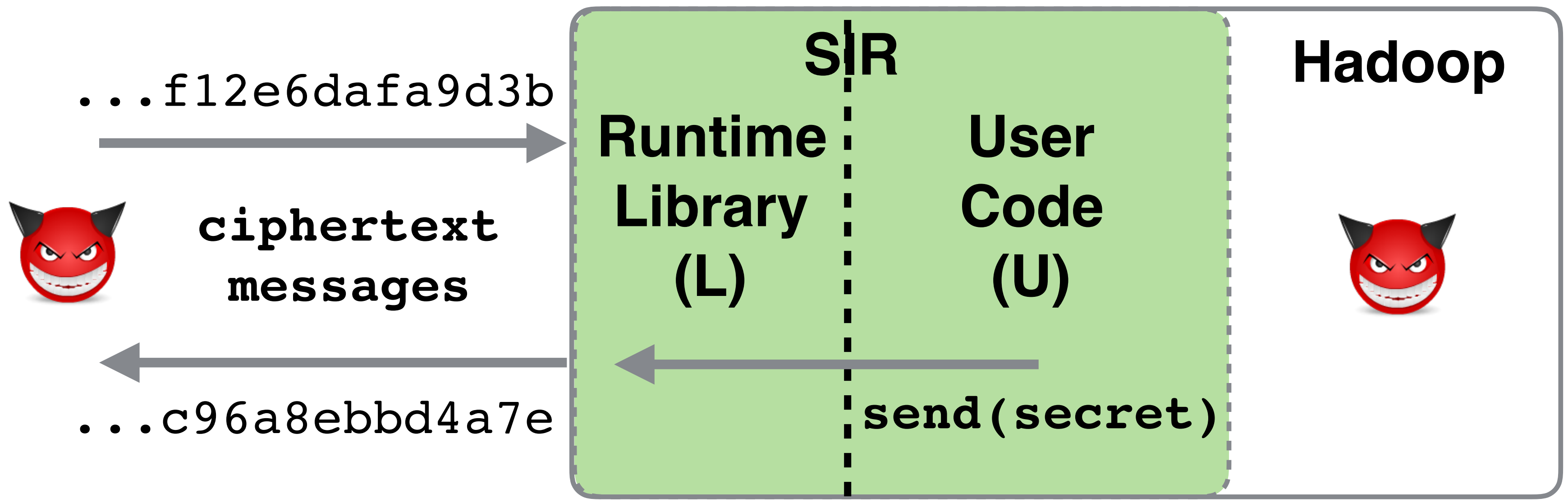


L implements **send**, **recv**, malloc, free

IRC: All updates to non-SIR memory via L's `send` API

```
void Reduce(byte *kEnc, byte *vEnc) {
  char *k = recv(..);
  char *v = recv(..);
```

Information Release Confinement

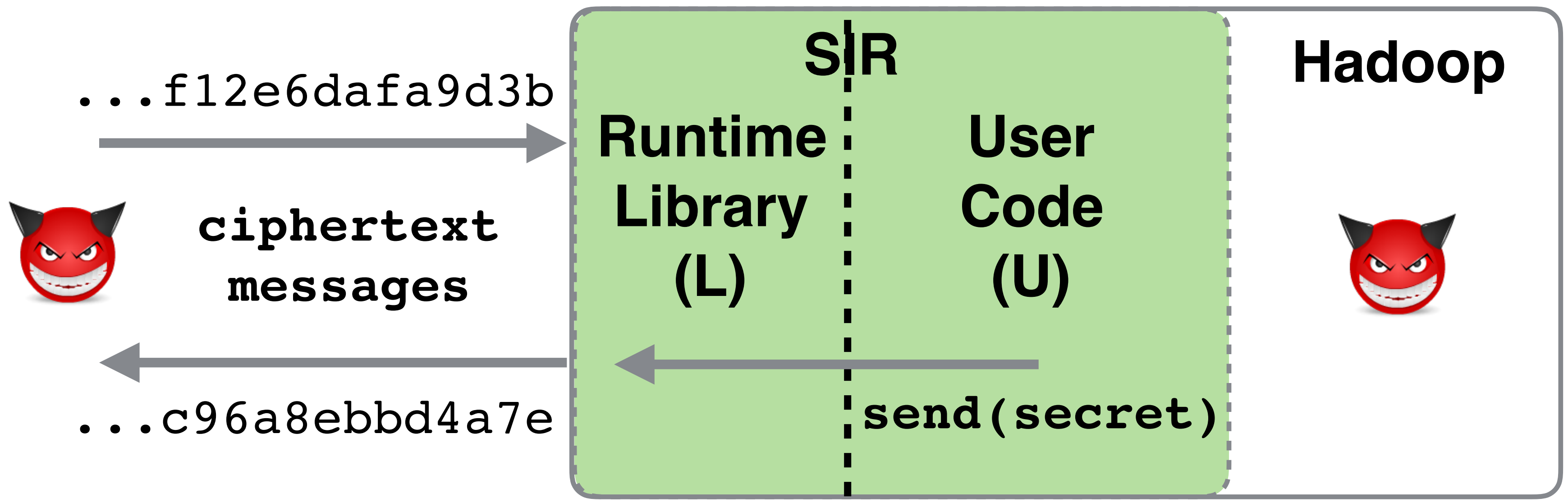


L implements **send**, **recv**, malloc, free

IRC: All updates to non-SIR memory via L's `send` API

```
void Reduce(byte *kEnc, byte *vEnc) {
  char *k = recv(..);
  char *v = recv(..);
  long sum = compute_sum(v);
```

Information Release Confinement

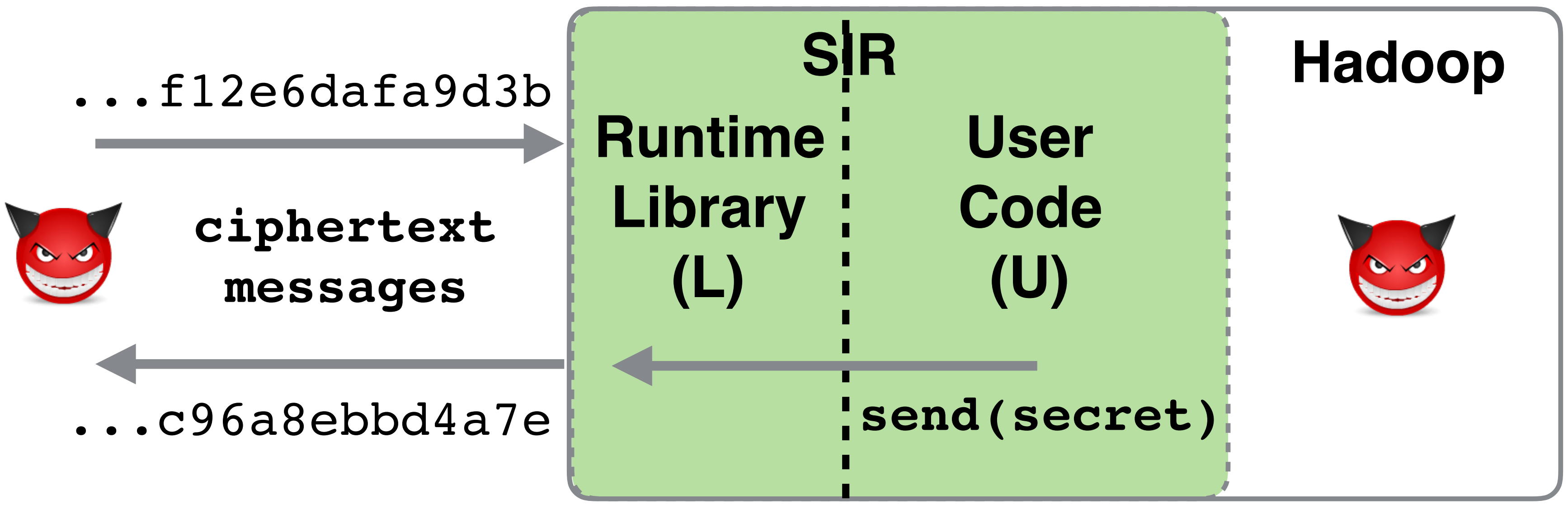


L implements **send**, **recv**, malloc, free

IRC: All updates to non-SIR memory via L's `send` API

```
void Reduce(byte *kEnc, byte *vEnc) {
  char *k = recv(..);
  char *v = recv(..);
  long sum = compute_sum(v);
  char cleartext[..];
  sprintf(cleartext, "%s %lld", k, sum);
  send(cleartext, ..);
}
```

Information Release Confinement



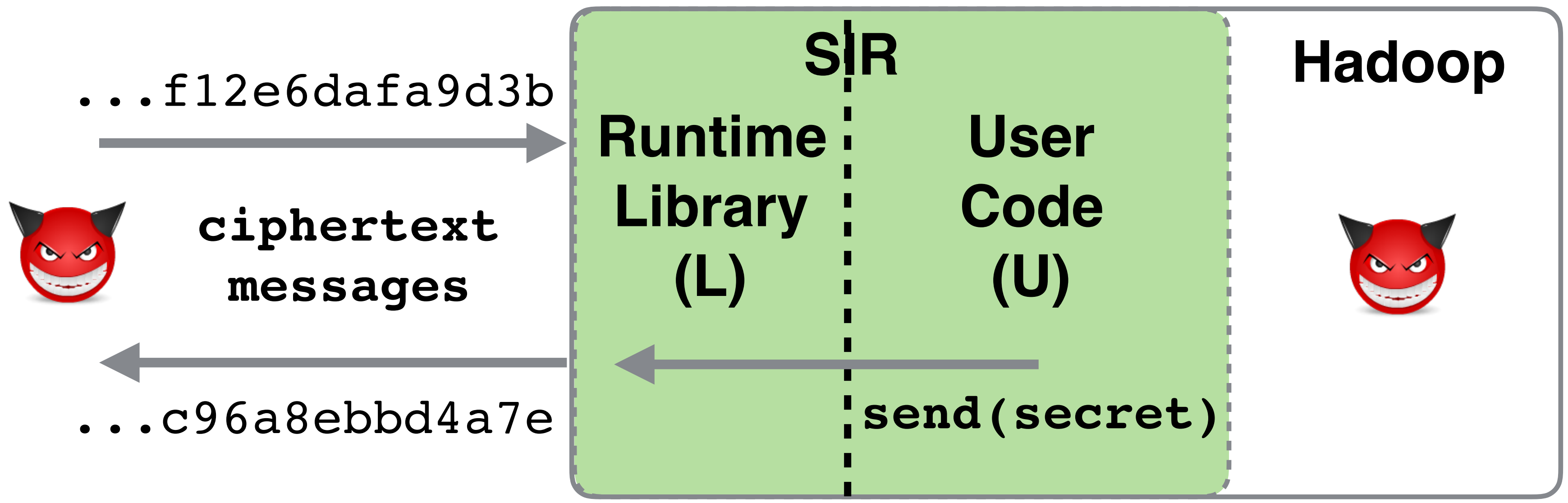
L implements **send**, **recv**, malloc, free

IRC: All updates to non-SIR memory via L's `send` API

```
void Reduce(byte *kEnc, byte *vEnc) {
  char *k = recv(..);
  char *v = recv(..);
  long sum = compute_sum(v);
  char cleartext[..];
  sprintf(cleartext, "%s %lld", k, sum);
  send(cleartext, ..);
}
```

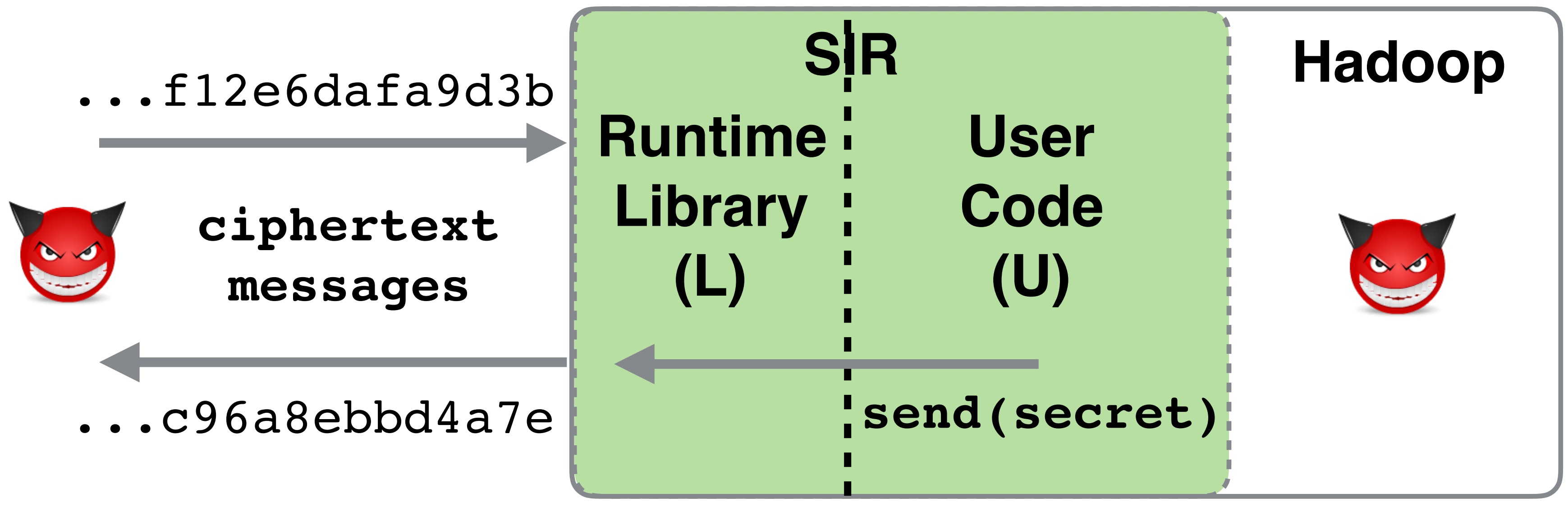
Separation of concerns:
U does not manage crypto keys or write to untrusted memory

Information Release Confinement



L implements **send**, **recv**, malloc, free

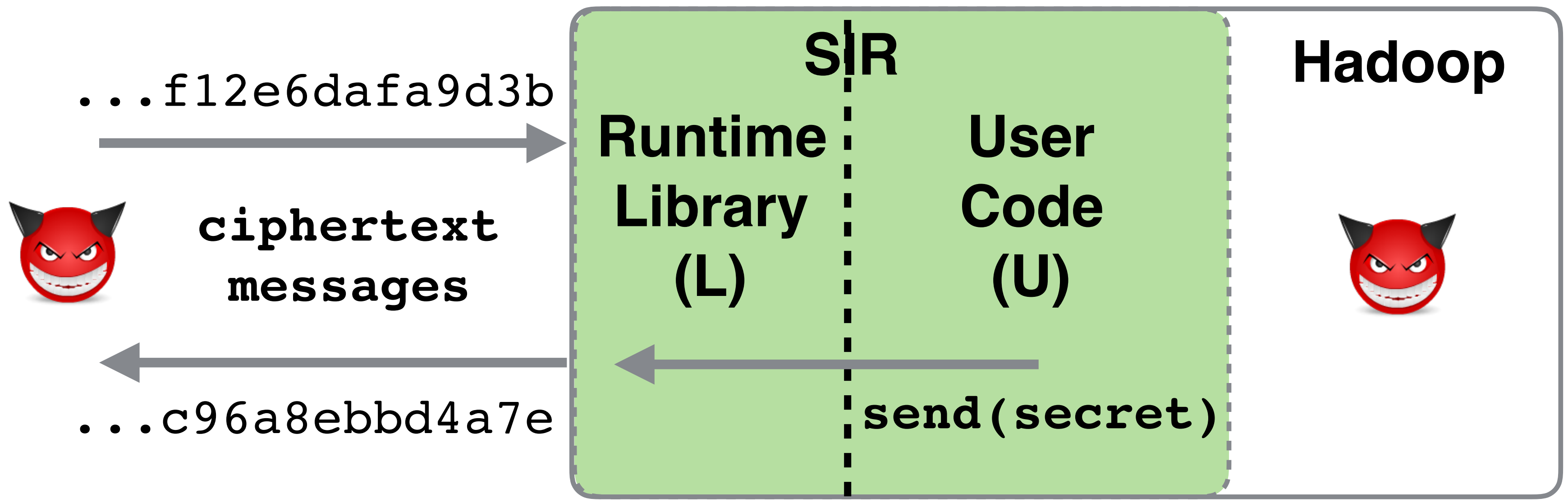
Information Release Confinement



L implements **send**, **recv**, malloc, free

IRC: All updates to non-SIR memory via L's `send` API

Information Release Confinement

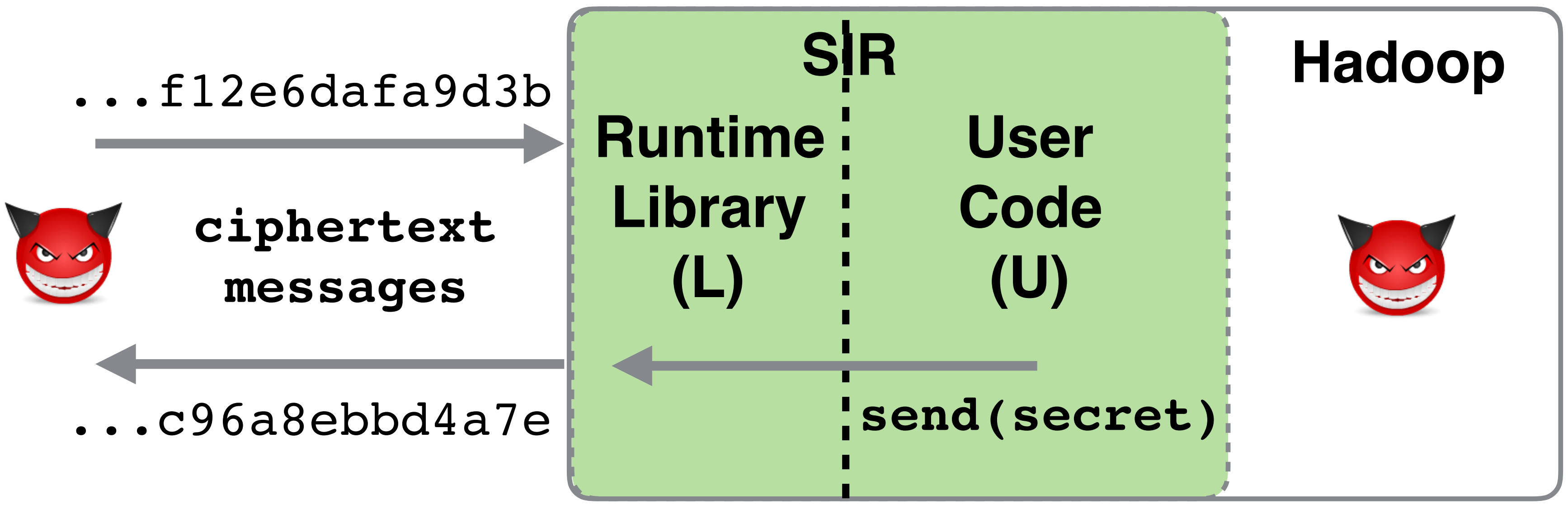


L implements **send**, **recv**, malloc, free

IRC: All updates to non-SIR memory via L's `send` API

✓ **Prevents explicit information leaks**: side channels outside scope

Information Release Confinement

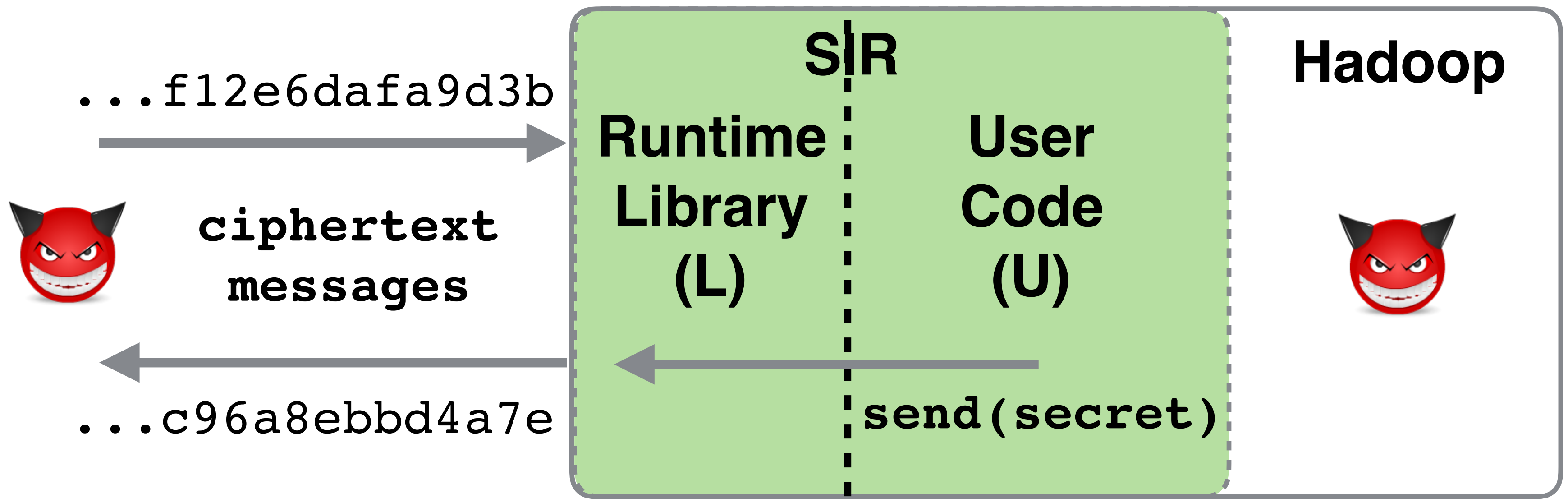


L implements
send, **recv**,
malloc, free

IRC: All updates to non-SIR memory via L's `send` API

- ✓ **Prevents explicit information leaks**: side channels outside scope
- ✓ Even if U is buggy, an **adversary only sees encrypted values** in an exploit

Information Release Confinement



L implements **send**, **recv**, malloc, free

IRC: All updates to non-SIR memory via L's `send` API

- ✓ **Prevents explicit information leaks**: side channels outside scope
- ✓ Even if U is buggy, an **adversary only sees encrypted values** in an exploit
- ✓ **Avoids fine-grained tracking of secrets** in U's memory: all of U is secret

Proving Information Release Confinement

SIR satisfies IRC

Proving Information Release Confinement

SIR satisfies IRC

Correctness of L

Proving Information Release Confinement

SIR satisfies IRC

Correctness of L

Correctness of U

Proving Information Release Confinement

SIR satisfies IRC

Correctness of L

Correctness of U

`send, recv`
implement a
cryptographically
secure channel

Proving Information Release Confinement

SIR satisfies IRC

Correctness of L

Correctness of U

`send, recv` implement a cryptographically secure channel

U calls into L at API entrypoints

Proving Information Release Confinement

SIR satisfies IRC

Correctness of L

Correctness of U

`send, recv` implement a cryptographically secure channel

U calls into L at API entrypoints

U does not write to non-SIR memory

Proving Information Release Confinement

SIR satisfies IRC

Correctness of L

`send, recv` implement a cryptographically secure channel

Correctness of U

U calls into L at API entrypoints

U does not write to non-SIR memory

✓ **We don’t require full functional correctness of U**

Proving Information Release Confinement

SIR satisfies IRC

Correctness of L

Correctness of U

`send, recv` implement a cryptographically secure channel

U calls into L at API entrypoints

U does not write to non-SIR memory

- ✓ **We don't require full functional correctness of U**
- ✓ **Proof strategy requires no annotations from the developer**

Contributions

Contributions

Formal Specification of IRC

IRC as a design methodology for programming SIRs

Contributions

Formal Specification of IRC

IRC as a design methodology for programming SIRs

Sound Decomposition of IRC proof into Contracts on U and L

SIR satisfies IRC

Correctness of L

`send, recv` implement a channel with secrecy and integrity

Correctness of U

U calls into L at API entrypoints

U does not write to non-SIR memory

Contributions

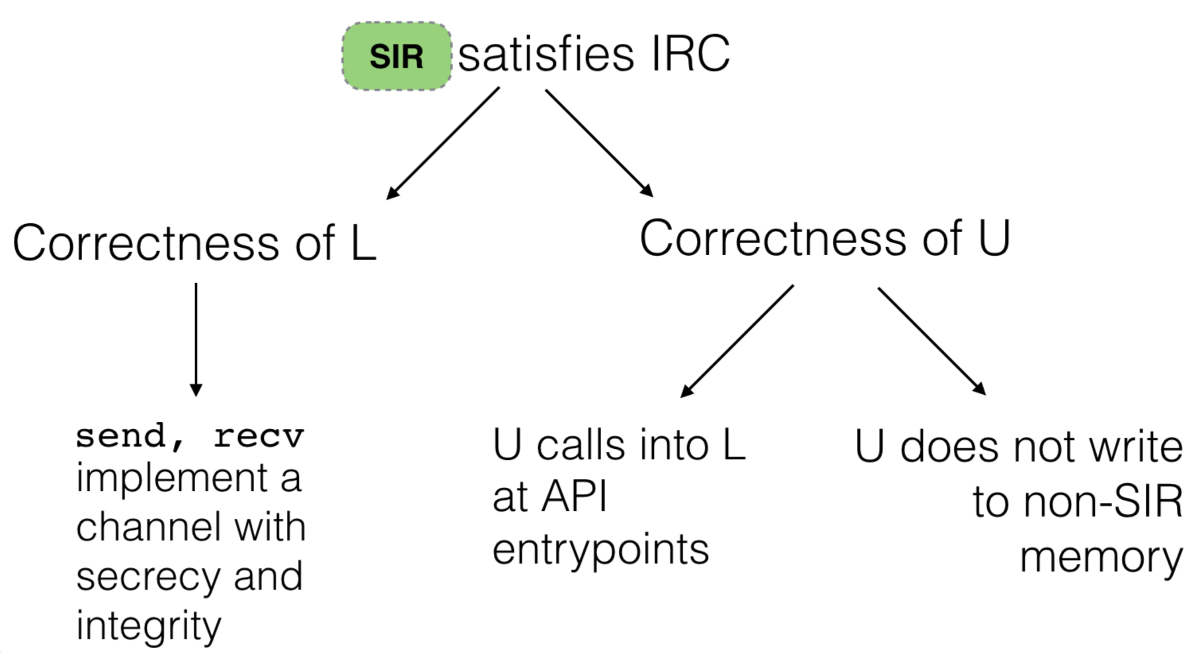
Formal Specification of IRC

IRC as a design methodology for programming SIRs

Automatic, Modular Verifier for proving IRC on U's binary

Verifier checks against a privileged OS-level adversary

Sound Decomposition of IRC proof into Contracts on U and L



Contributions

Formal Specification of IRC

IRC as a design methodology for programming SIRs

Automatic, Modular Verifier for proving IRC on U's binary

Verifier checks against a privileged OS-level adversary

Sound Decomposition of IRC proof into Contracts on U and L

SIR satisfies IRC

Correctness of L

`send, recv` implement a channel with secrecy and integrity

Correctness of U

U calls into L at API entrypoints

U does not write to non-SIR memory

Evaluation on several SIRs

Map-Reduce benchmarks from VC3

SPEC benchmarks

This Talk: Automatically Proving IRC

This Talk: Automatically Proving IRC

Verifying Calls in U

Verifying that U calls into L at API entrypoints

This Talk: Automatically Proving IRC

Verifying Calls in U Verifying that U calls into L at API entrypoints

Verifying Writes in U Verifying that U does not modify non-SIR memory

This Talk: Automatically Proving IRC

Verifying Calls in U	Verifying that U calls into L at API entrypoints
Verifying Writes in U	Verifying that U does not modify non-SIR memory
Verifying L	Correctness properties of L

This Talk: Automatically Proving IRC

Verifying Calls in U	Verifying that U calls into L at API entrypoints
Verifying Writes in U	Verifying that U does not modify non-SIR memory
Verifying L	Correctness properties of L
Evaluation	Evaluation on VC3 and SPEC

Verifying U: Calls to L via API Entrypoints

Verifying Calls in U

Verifying Writes in U

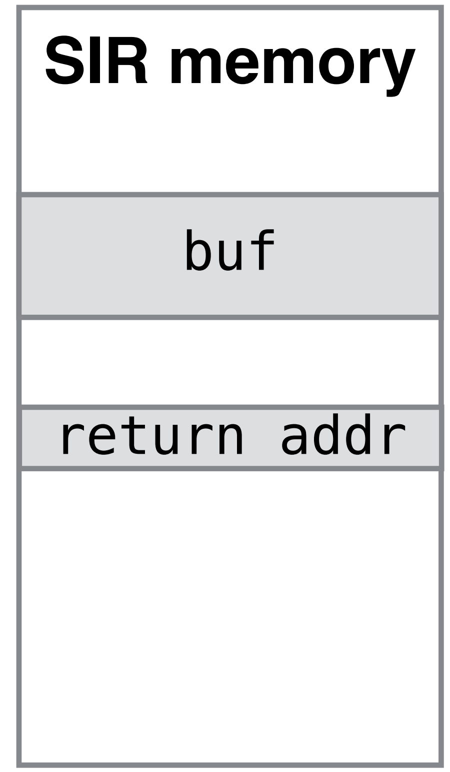
Verifying L

Evaluation

Verifying U: Calls to L via API Entrypoints

Potential Code in U

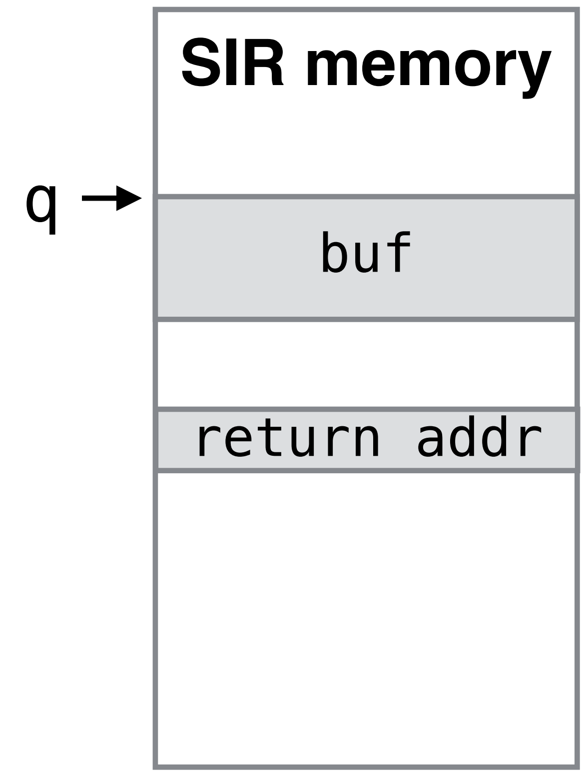
```
*q = buf + input;
*q = input2;
...
return;
```



Verifying U: Calls to L via API Entrypoints

Potential Code in U

```
*q = buf + input;
*q = input2;
...
return;
```



Verifying U: Calls to L via API Entrypoints

Verifying Calls in U

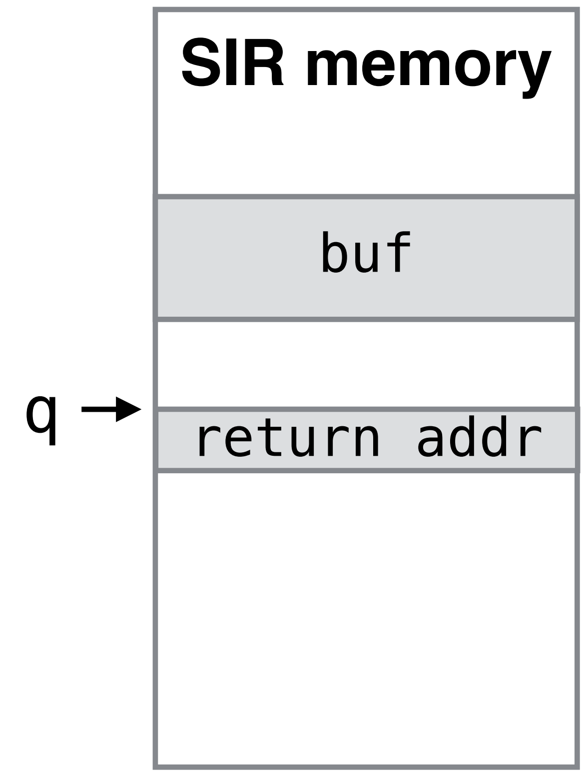
Verifying Writes in U

Verifying L

Evaluation

Potential Code in U

```
*q = buf + input;
*q = input2;
...
return;
```



Verifying U: Calls to L via API Entrypoints

Verifying Calls in U

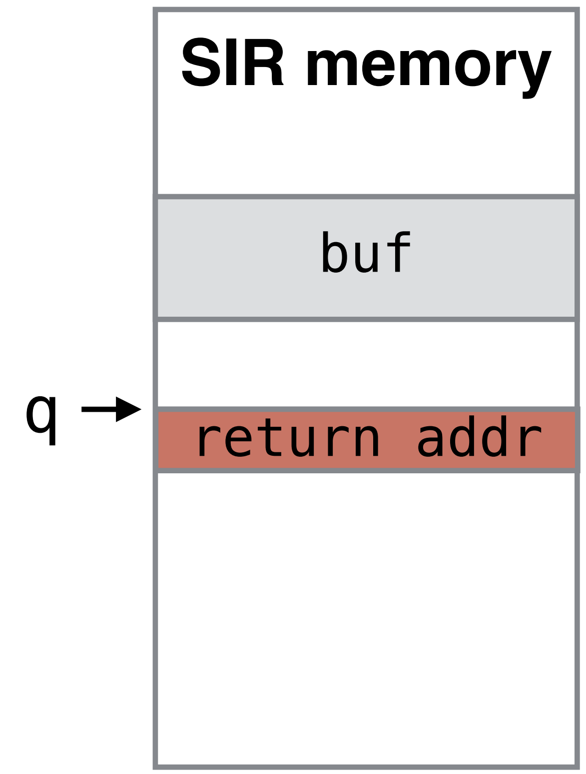
Verifying Writes in U

Verifying L

Evaluation

Potential Code in U

```
*q = buf + input;
*q = input2;
...
return;
```



Verifying U: Calls to L via API Entrypoints

Verifying Calls in U

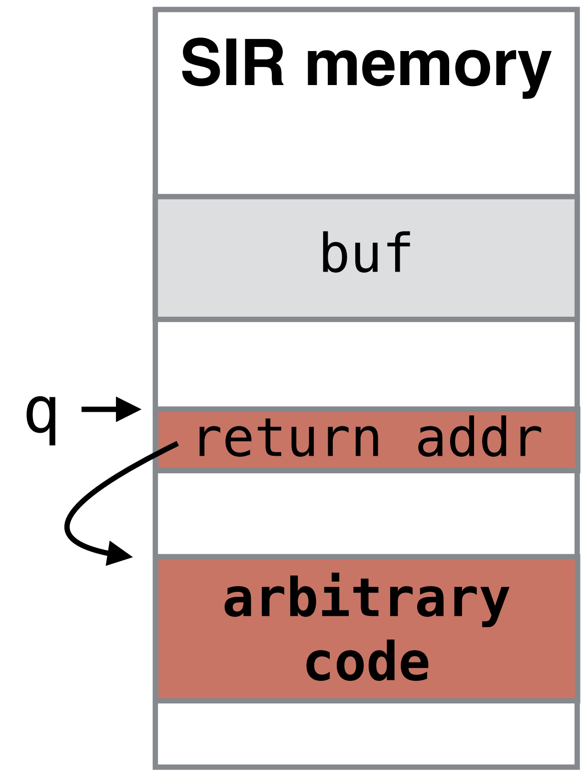
Verifying Writes in U

Verifying L

Evaluation

Potential Code in U

```
*q = buf + input;
*q = input2;
...
return;
```



Verifying U: Calls to L via API Entrypoints

Verifying Calls in U

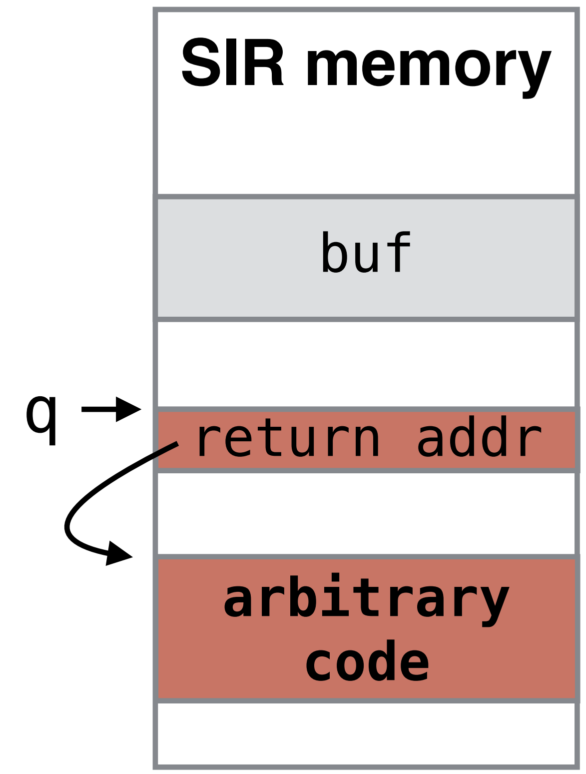
Verifying Writes in U

Verifying L

Evaluation

Potential Code in U

```
*q = buf + input;
*q = input2;
...
return;
```



<u>arbitrary code</u>

- middle of x86 instructions
- arbitrary instructions in L

Verifying U: Calls to L via API Entrypoints

Verifying Calls in U

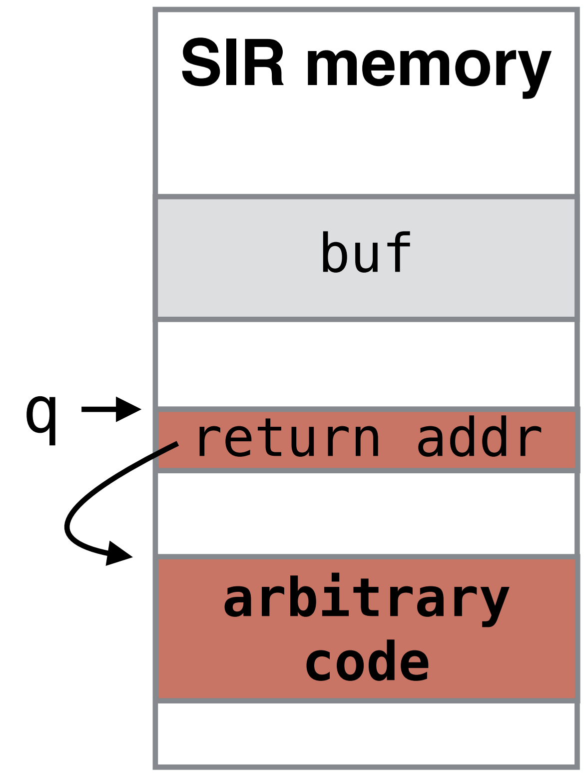
Verifying Writes in U

Verifying L

Evaluation

Potential Code in U

```
*q = buf + input;
*q = input2;
...
return;
```

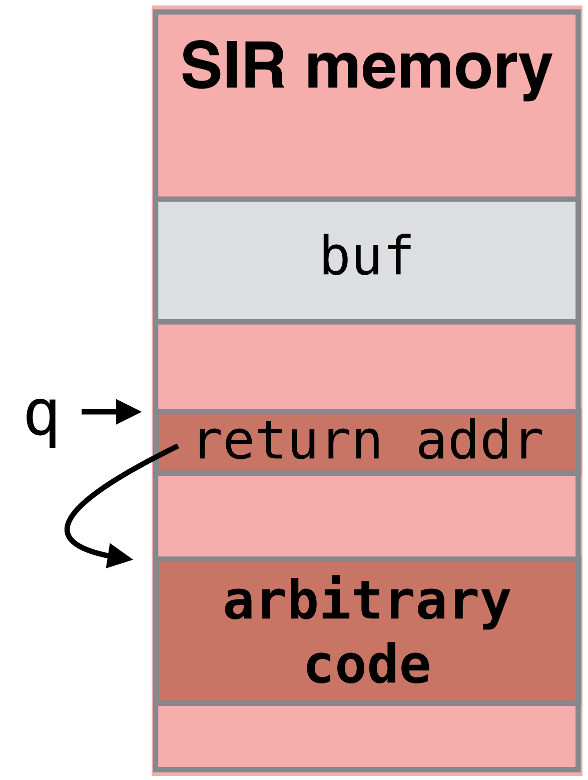


<u>arbitrary code</u>
- middle of x86 instructions
- arbitrary instructions in L

Verifying U: Calls to L via API Entrypoints

Potential Code in U

```
*q = buf + input;
*q = input2;
...
return;
```



<u>arbitrary code</u>
- middle of x86 instructions
- arbitrary instructions in L

Verifying U: Calls to L via API Entrypoints

Verifying Calls in U

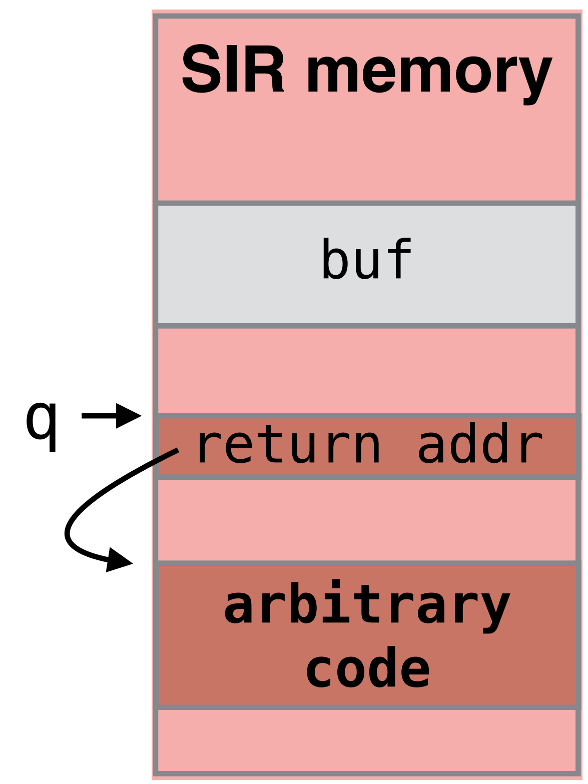
Verifying Writes in U

Verifying L

Evaluation

Potential Code in U

```
*q = buf + input;
*q = input2;
...
return;
```



arbitrary code
- middle of x86 instructions
- arbitrary instructions in L

Control Flow Integrity

Verifying U: Calls to L via API Entrypoints

Verifying Calls in U

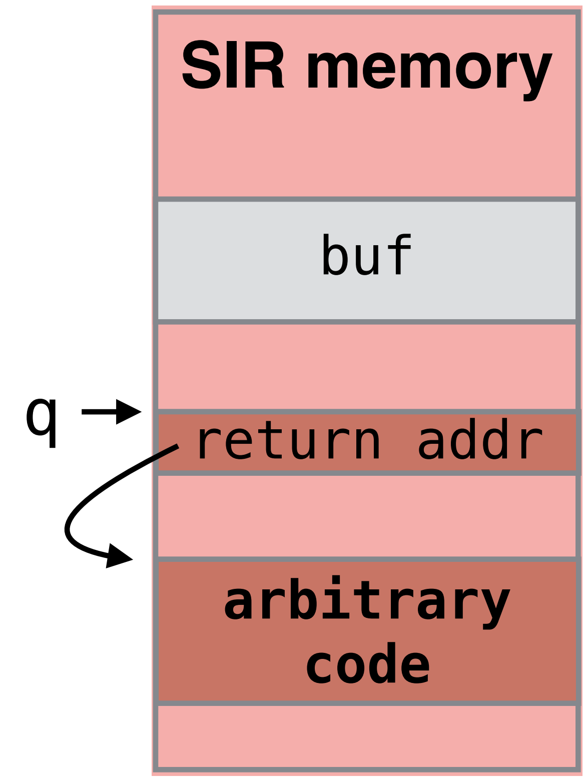
Verifying Writes in U

Verifying L

Evaluation

Potential Code in U

```
*q = buf + input;
*q = input2;
...
return;
```



<u>arbitrary code</u>

- middle of x86 instructions
- arbitrary instructions in L

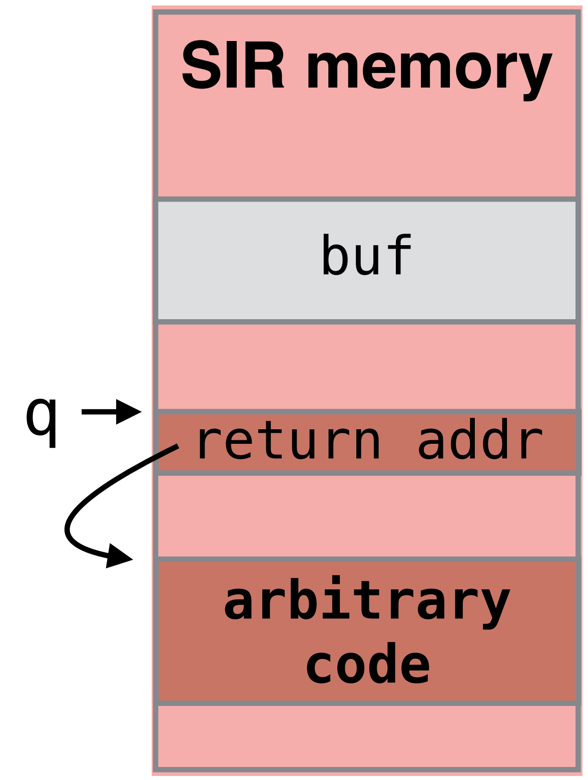
Control Flow Integrity

✓ A `call` instruction targets the starting address of a procedure in U or API of L

Verifying U: Calls to L via API Entrypoints

Potential Code in U

```
*q = buf + input;
*q = input2;
...
return;
```



<u>arbitrary code</u>
- middle of x86 instructions
- arbitrary instructions in L

Control Flow Integrity

✓ A `call` instruction targets the starting address of a procedure in U or API of L

✓ A `ret` instruction returns back to the caller

Verifying U: Calls to L via API Entrypoints

Verifying Calls in U

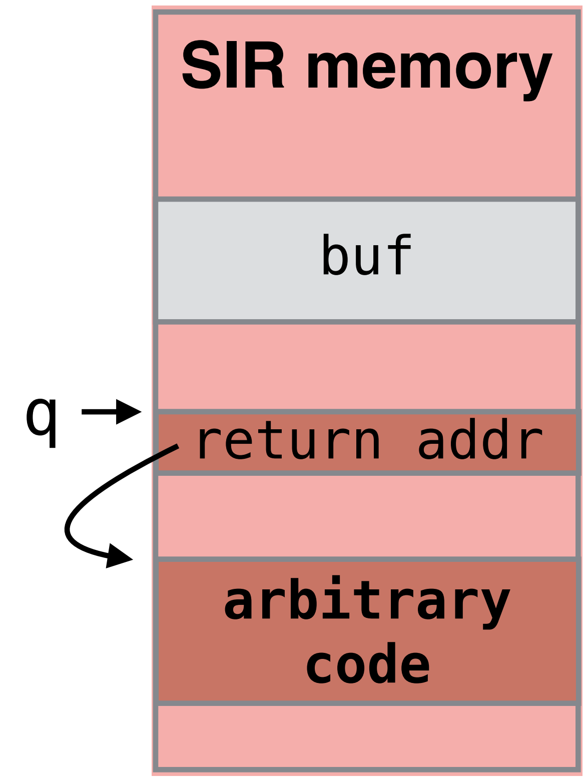
Verifying Writes in U

Verifying L

Evaluation

Potential Code in U

```
*q = buf + input;
*q = input2;
...
return;
```



<u>arbitrary code</u>

- middle of x86 instructions
- arbitrary instructions in L

Control Flow Integrity

✓ A `call` instruction targets the starting address of a procedure in U or API of L

✓ A `ret` instruction returns back to the caller

✓ A `jmp` instruction targets a legal instruction within the procedure

Verifying U: Calls to L via API Entrypoints

Verifying Calls in U

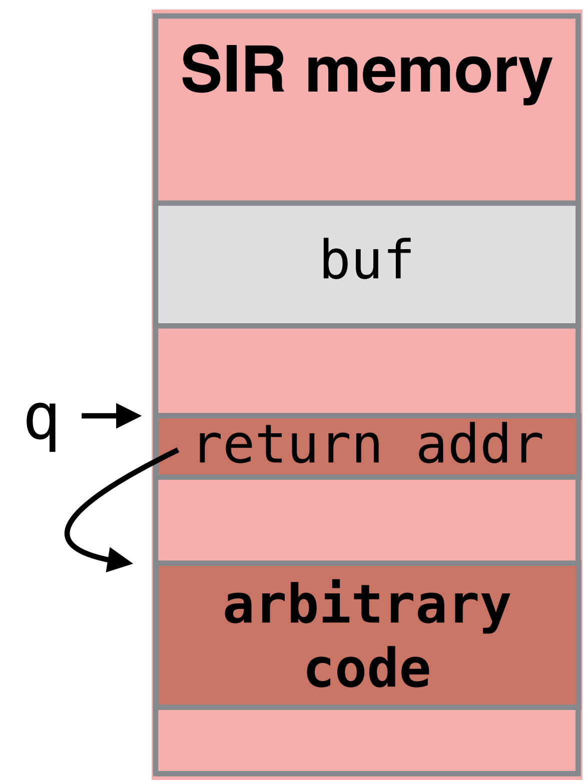
Verifying Writes in U

Verifying L

Evaluation

Potential Code in U

```
*q = buf + input;
*q = input2;
...
return;
```



arbitrary code
- middle of x86 instructions
- arbitrary instructions in L

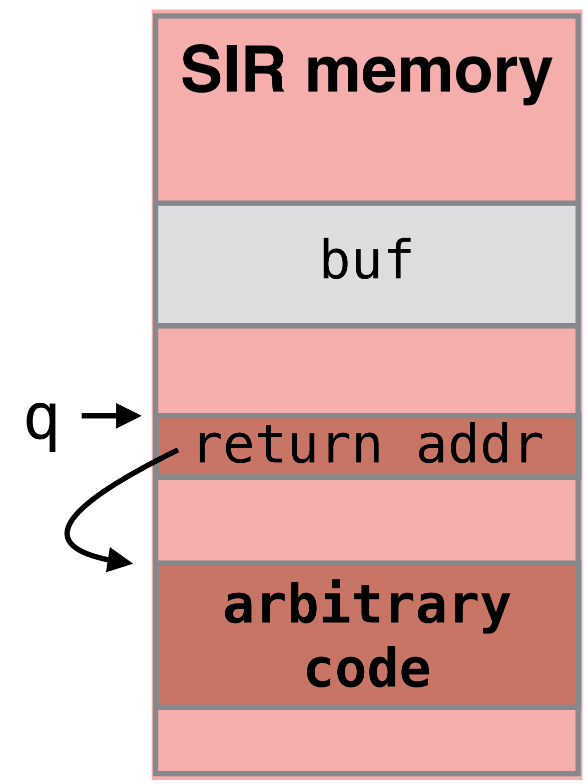
Weak Control Flow Integrity (WCFI)

- ✓ A `call` instruction targets the starting address of a procedure in U or API of L
- ✓ A `ret` instruction returns back to the caller
- ✓ A `jmp` instruction targets a legal instruction within the procedure

Verifying U: Calls to L via API Entrypoints

Potential Code in U

```
*q = buf + input;
*q = input2;
...
return;
```



arbitrary code
- middle of x86 instructions
- arbitrary instructions in L

Weak Control Flow Integrity (WCFI)

- ✓ A `call` instruction targets the starting address of a procedure in U or API of L
- ✓ A `ret` instruction returns back to the caller
- ✓ A `jmp` instruction targets a legal instruction within the procedure

WCFI ⇒

U calls into L via APIs

Verifying U: Runtime Checks for WCFI

Verifying Calls in U

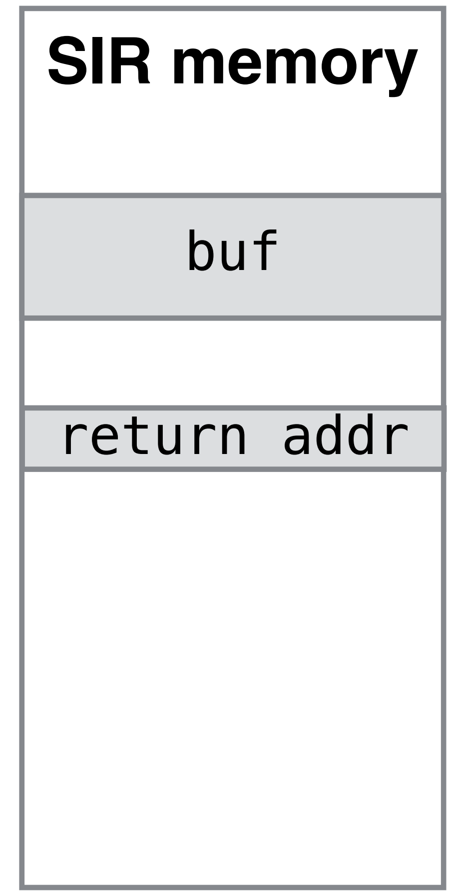
Verifying Writes in U

Verifying L

Evaluation

Potential Code in U

```
*q = buf + input;
*q = input2;
...
return;
```



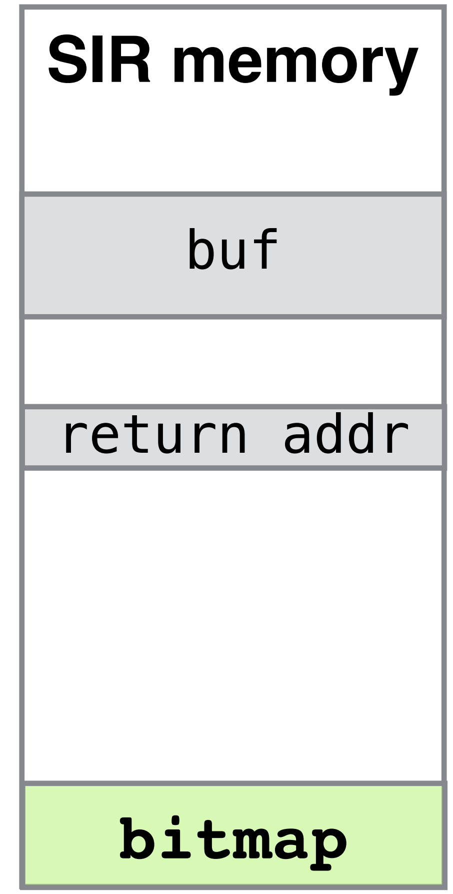
Verifying U: Runtime Checks for WCFI

- Verifying Calls in U
- Verifying Writes in U
- Verifying L
- Evaluation

Potential Code in U

```
*q = buf + input;
*q = input2;
...
return;
```

`Runtime check using VC3 compiler:`
`is address of q marked writable in bitmap?`

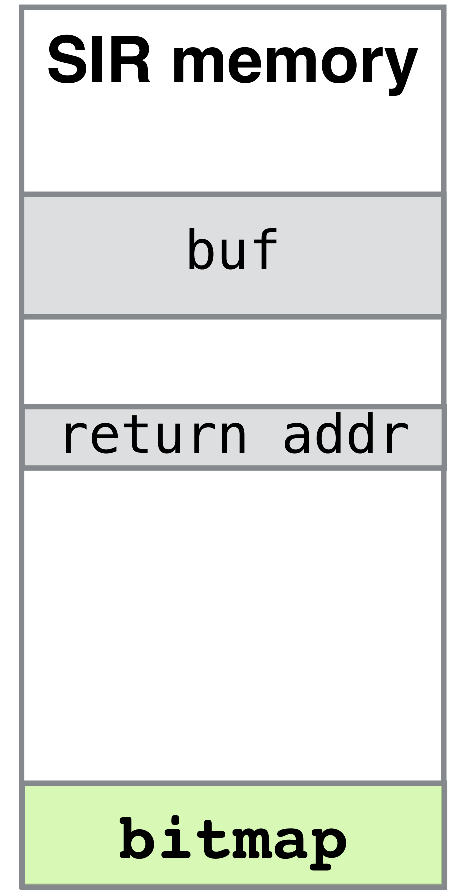


Verifying U: Runtime Checks for WCFI

- **Verifying Calls in U**
- Verifying Writes in U
- Verifying L
- Evaluation

Potential Code in U

```
*q = buf + input;
*q = input2;
...
return;
```



Runtime check using VC3 compiler:
is address of q marked writable
in bitmap?

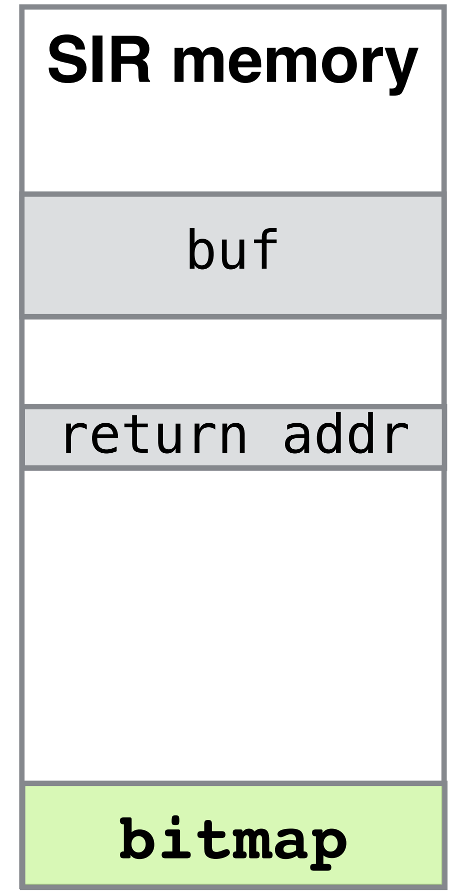
<u>not-writable</u>
return addresses

Verifying U: Runtime Checks for WCFI

- Verifying Calls in U
- Verifying Writes in U
- Verifying L
- Evaluation

Potential Code in U

```
*q = buf + input;
*q = input2;
...
return;
```



Runtime check using VC3 compiler:
is address of q marked writable in bitmap?

<u>not-writable</u>
return addresses

<u>writable</u>
local vars
heap objects

Verifying Calls in U

Verifying Writes in U

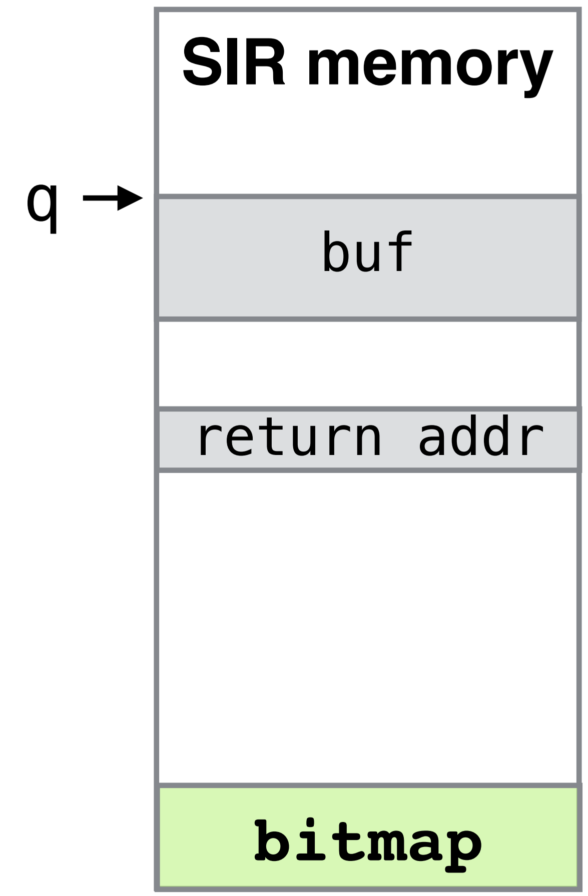
Verifying L

Evaluation

Verifying U: Runtime Checks for WCFI

Potential Code in U

```
*q = buf + input;
*q = input2;
...
return;
```



Runtime check using VC3 compiler:
is address of q marked writable
in bitmap?

<u>not-writable</u>
return addresses

<u>writable</u>
local vars
heap objects

Verifying U: Runtime Checks for WCFI

Verifying Calls in U

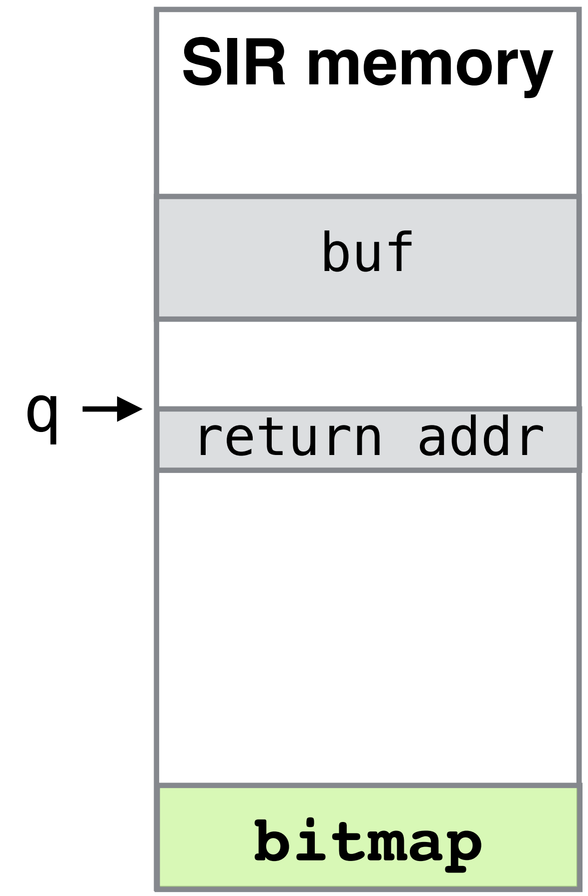
Verifying Writes in U

Verifying L

Evaluation

Potential Code in U

```
*q = buf + input;
*q = input2;
...
return;
```



Runtime check using VC3 compiler:
is address of q marked writable
in bitmap?

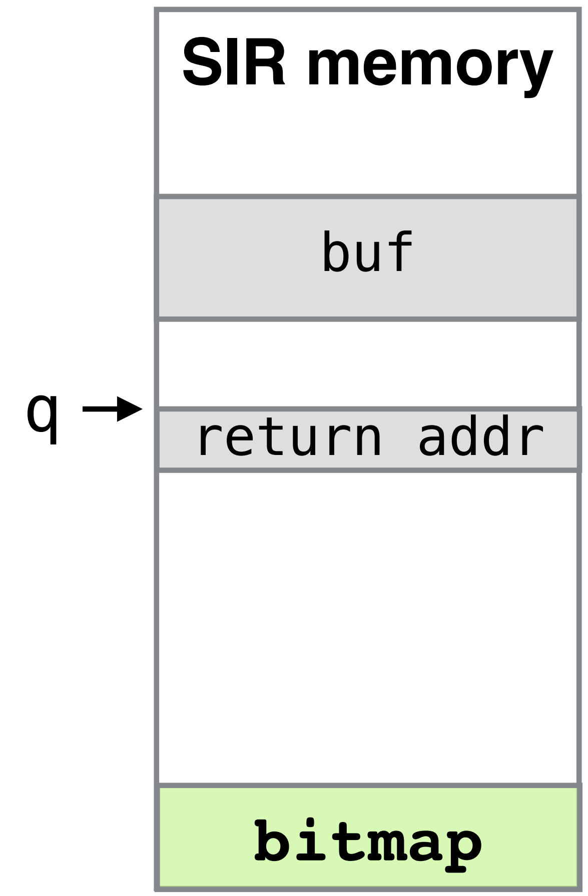
<u>not-writable</u>
return addresses

<u>writable</u>
local vars
heap objects

Verifying U: Runtime Checks for WCFI

Potential Code in U

```
*q = buf + input;
*q = input2;
...
return;
```



Runtime check using VC3 compiler:
is address of q marked writable in bitmap?

<u>not-writable</u>
return addresses

<u>writable</u>
local vars
heap objects

Verifying U: Runtime Checks for WCFI

Verifying Calls in U

Verifying Writes in U

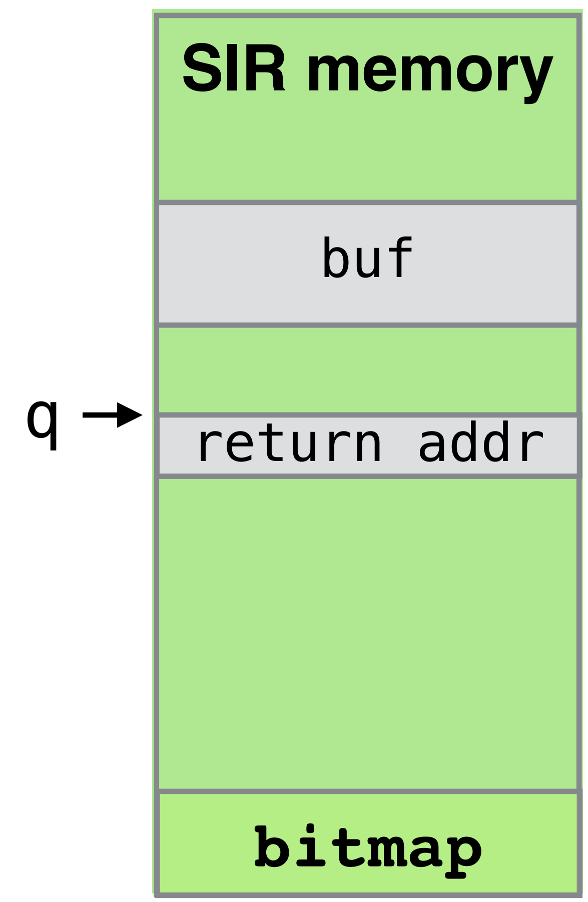
Verifying L

Evaluation

Potential Code in U

```
*q = buf + input;
*q = input2;
...
return;
```

Software TRAP



Runtime check using VC3 compiler:
is address of q marked writable in bitmap?

<u>not-writable</u>
return addresses

<u>writable</u>
local vars
heap objects

Verifying U: Modeling U's Binary

- **Verifying Calls in U**
- Verifying Writes in U
- Verifying L
- Evaluation

Verifying U: Modeling U's Binary

Avoid trusting the compiler:

Long history of bugs

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

Verifying U: Modeling U's Binary

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

Avoid trusting the compiler:

Long history of bugs

Compiler optimizes away many runtime checks

Verifying U: Modeling U's Binary

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

Avoid trusting the compiler:

Long history of bugs

Compiler optimizes away many runtime checks

x64 code produced by compiler

```
...
mov rcx, [rax+rbx]
bt rcx, rbx
jb $L2
int 3
$L2: mov [rbx],rdx
...
ret
```

Verifying U: Modeling U's Binary

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

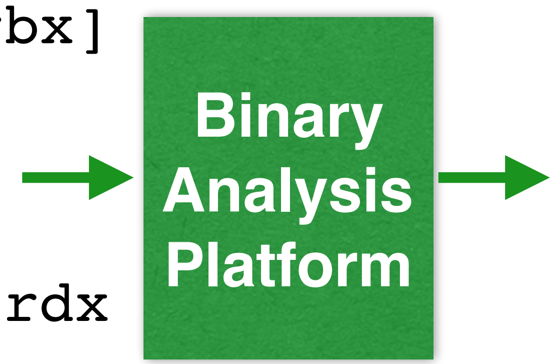
Avoid trusting the compiler:

Long history of bugs

Compiler optimizes away many runtime checks

x64 code produced by compiler

```
...
mov rcx, [rax+rbx]
bt rcx, rbx
jb $L2
int 3
$L2: mov [rbx],rdx
...
ret
```



Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

Verifying U: Modeling U's Binary

Avoid trusting the compiler:

Long history of bugs

Compiler optimizes away many runtime checks

x64 code produced by compiler

```
...
mov rcx, [rax+rbx]
bt rcx, rbx
jb $L2
int 3
$L2: mov [rbx],rdx
...
ret
```

→ **Binary Analysis Platform** →

Boogie model

```
...
rcx := load(mem, rax +_64 rbx);
CF := (rcx >>_64 rbx)[1:0];
goto $L1, $L2;
$L1: assume CF == 0;
     assume false;
$L2: assume CF == 1;
     mem := store(mem, rbx, rdx);
...
return;
```

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

Verifying U: Modeling U's Binary

Avoid trusting the compiler:

Long history of bugs

Compiler optimizes away many runtime checks

x64 code produced by compiler

```
...
mov rcx, [rax+rbx]
bt rcx, rbx
jb $L2
int 3
$L2: mov [rbx],rdx
...
ret
```

→ **Binary Analysis Platform** →

Boogie model

```
...
rcx := load(mem, rax +_64 rbx);
CF := (rcx >>_64 rbx)[1:0];
goto $L1, $L2;
$L1: assume CF == 0;
     assume false;
$L2: assume CF == 1;
     mem := store(mem, rbx, rdx);
...
return;
```

Verifying U: Modeling the Adversary

Verifying Calls in U

Verifying Writes in U

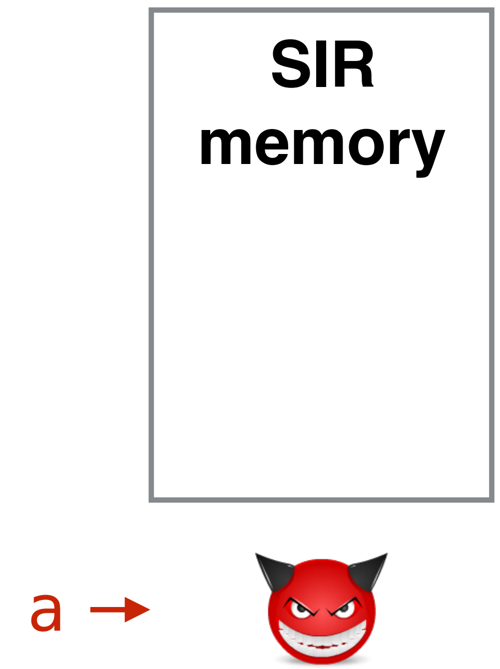
Verifying L

Evaluation

Verifying U: Modeling the Adversary

load from untrusted memory returns arbitrary value *

```
load(mem,a) = ITE(SIR(a), mem[a], *);
```



Verifying U: Modeling the Adversary

Verifying Calls in U

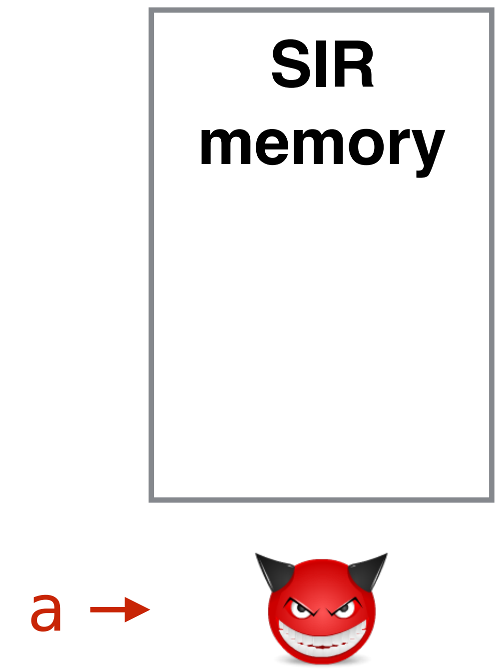
Verifying Writes in U

Verifying L

Evaluation

load from untrusted memory returns arbitrary value *

```
load(mem,a) = ITE(SIR(a), mem[a], *);
```



[Moat *CCS'15*]: **models all operations by a malicious OS** e.g. generate interrupts, modify page tables, launch other SIRs, etc.

Verifying U: Proof Obligations

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

Boogie model

```
...
rcx := load(mem, rax +64 rbx);
CF := (rcx >>64 rbx)[1:0];
goto $L1, $L2;
$L1: assume CF == 0;
     assume false;
```

Verifying U: Proof Obligations

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

Boogie model

```
...
rcx := load(mem, rax +64 rbx);
CF := (rcx >>64 rbx)[1:0];
goto $L1, $L2;
$L1: assume CF == 0;
     assume false;
$L2: assume CF == 1;
     assert Ψ;
     mem := store(mem, rbx, rdx);
...
```

Verifying U: Proof Obligations

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

Boogie model

```
...
rcx := load(mem, rax +64 rbx);
CF := (rcx >>64 rbx)[1:0];
goto $L1, $L2;
$L1: assume CF == 0;
     assume false;
$L2: assume CF == 1;
     assert Ψ;
     mem := store(mem, rbx, rdx);
...
assert φ;
return;
```

Verifying U: Proof Obligations

- Verifying Calls in U
- Verifying Writes in U
- Verifying L
- Evaluation

Boogie model

```
...
rcx := load(mem, rax +64 rbx);
CF := (rcx >>64 rbx)[1:0];
goto $L1, $L2;
$L1: assume CF == 0;
     assume false;
$L2: assume CF == 1;
     assert Ψ;
     mem := store(mem, rbx, rdx);
...
assert φ;
return;
```

Proof Obligations guarantee WCFI ⇒

U calls into L via APIs

Verifying U: Proof Obligations

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

Boogie model

```
...
rcx := load(mem, rax +64 rbx);
CF := (rcx >>64 rbx)[1:0];
goto $L1, $L2;
$L1: assume CF == 0;
     assume false;
$L2: assume CF == 1;
     assert Ψ;
     mem := store(mem, rbx, rdx);
...
assert φ;
return;
```

VC-gen + SMT Solving

Proof Obligations guarantee WCFI ⇒

U calls into L via APIs

Verifying U: Proof Obligations

Boogie model

```
...
rcx := load(mem, rax +_64 rbx);
CF := (rcx >>_64 rbx)[1:0];
goto $L1, $L2;
$L1: assume CF == 0;
     assume false;
$L2: assume CF == 1;
     assert Ψ;
     mem := store(mem, rbx, rdx);
...
assert φ;
return;
```

VC-gen + SMT Solving

Proof Obligations guarantee WCFI ⇒ U calls into L via APIs

Presence of runtime checks helps SMT solver to prove `assert Ψ`

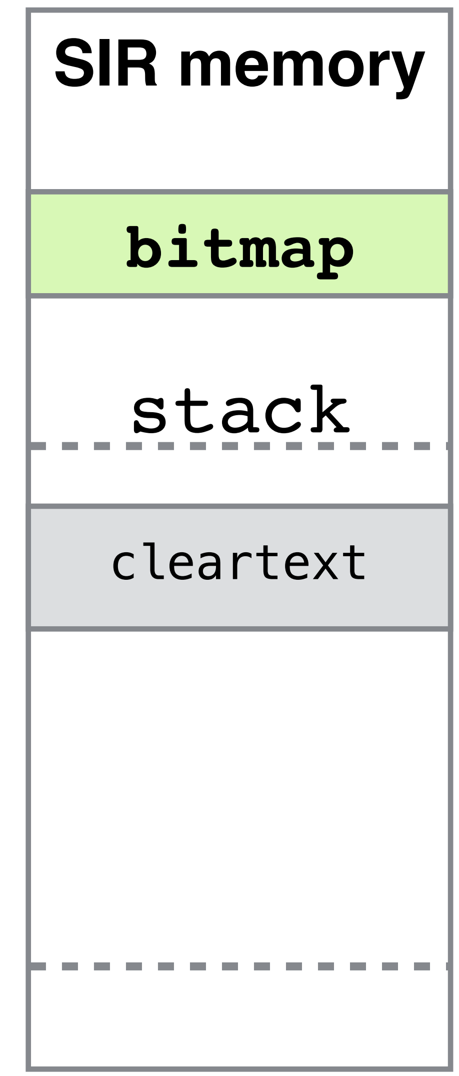
Verifying U: Proof Obligations on `store`

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation



Verifying U: Proof Obligations on `store`

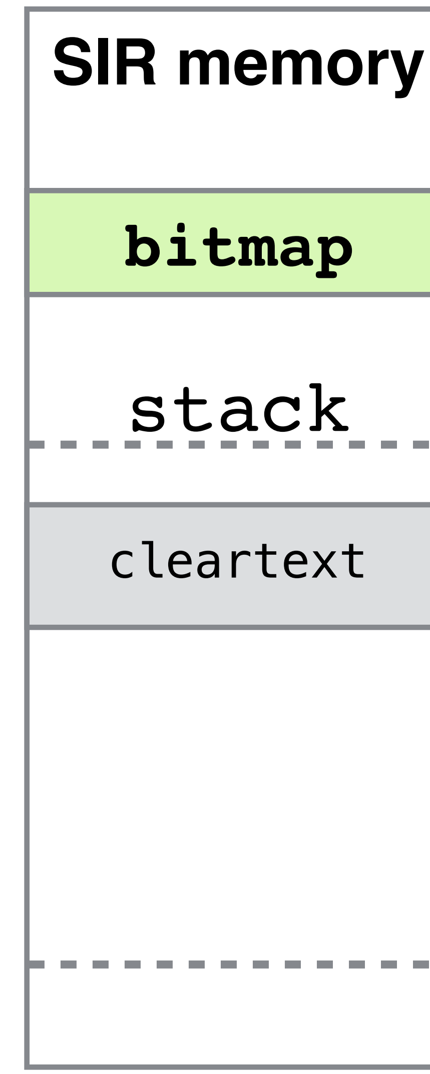
Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

```
void Reduce(...) {
   ...
   sprintf(cleartext, “%s %lld”,...);
   ...
}

void sprintf(char *cleartext,...) {
   /* write to cleartext */
}
```



Verifying U: Proof Obligations on `store`

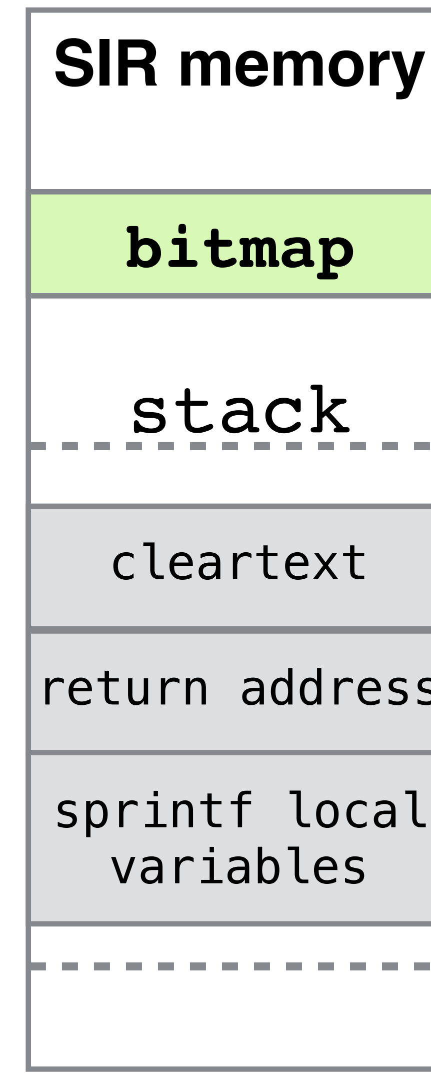
Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

```
void Reduce(...) {
   ...
   sprintf(cleartext, “%s %lld”,...);
   ...
}

void sprintf(char *cleartext,...) {
   /* write to cleartext */
}
```



Verifying U: Proof Obligations on `store`

Verifying Calls in U

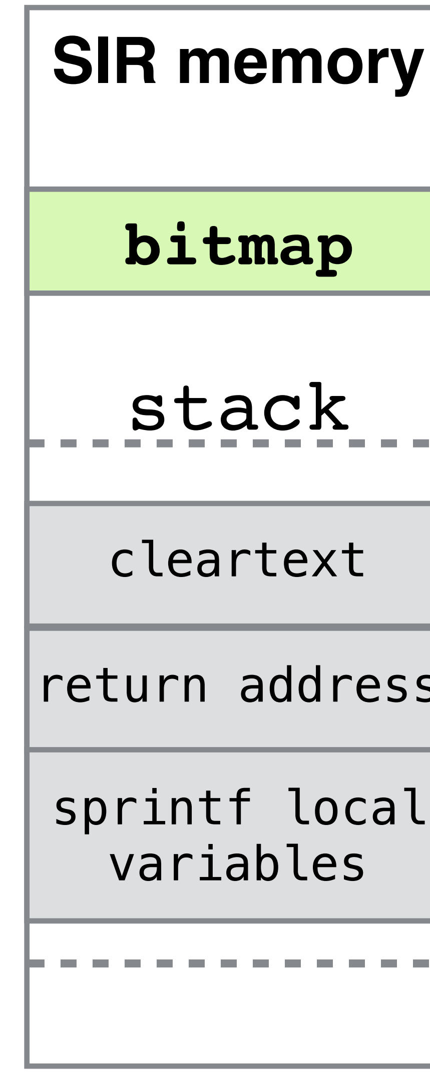
Verifying Writes in U

Verifying L

Evaluation

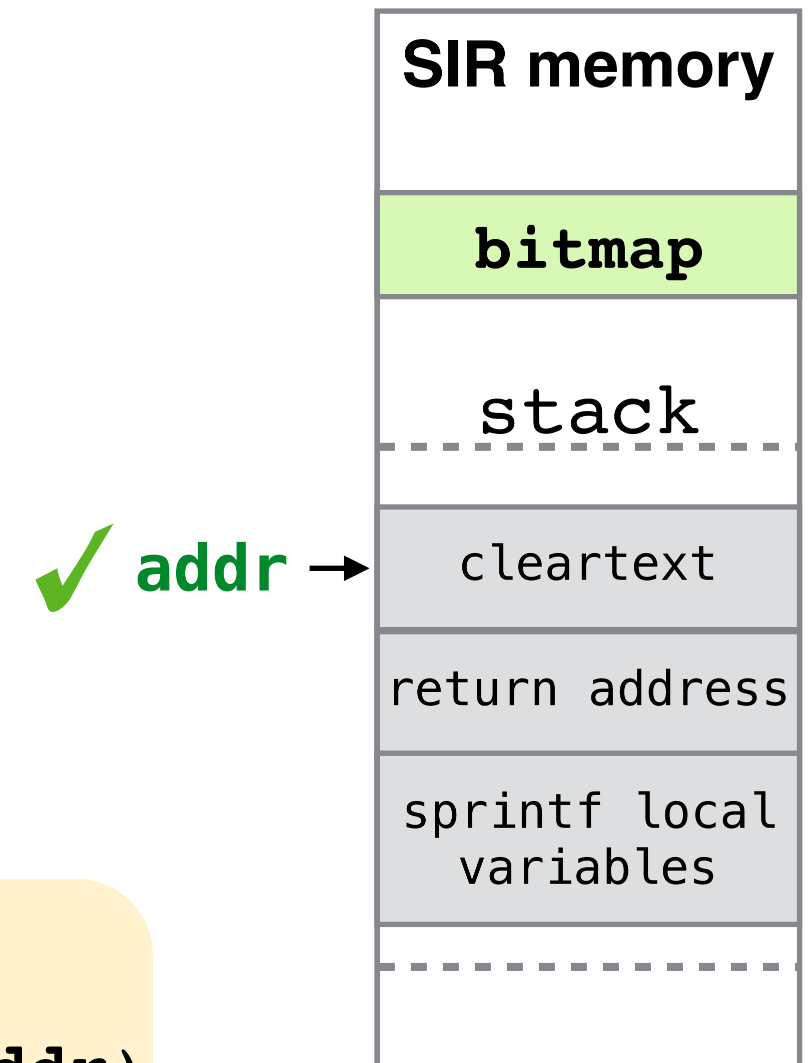
```
void Reduce(...) {
   ...
   sprintf(cleartext, "%s %lld",...);
   ...
}

void sprintf(char *cleartext,...) {
   /* write to cleartext */
}
```

verifier **asserts writable(bitmap, addr)** for each store



Verifying U: Proof Obligations on `store`


```
void Reduce(...) {
  ...
  sprintf(cleartext, “%s %lld”,...);
  ...
}

void sprintf(char *cleartext,...) {
  /* write to cleartext */
}
```



verifier **asserts writable(bitmap, addr)** for each store

Verifying U: Proof Obligations on `store`

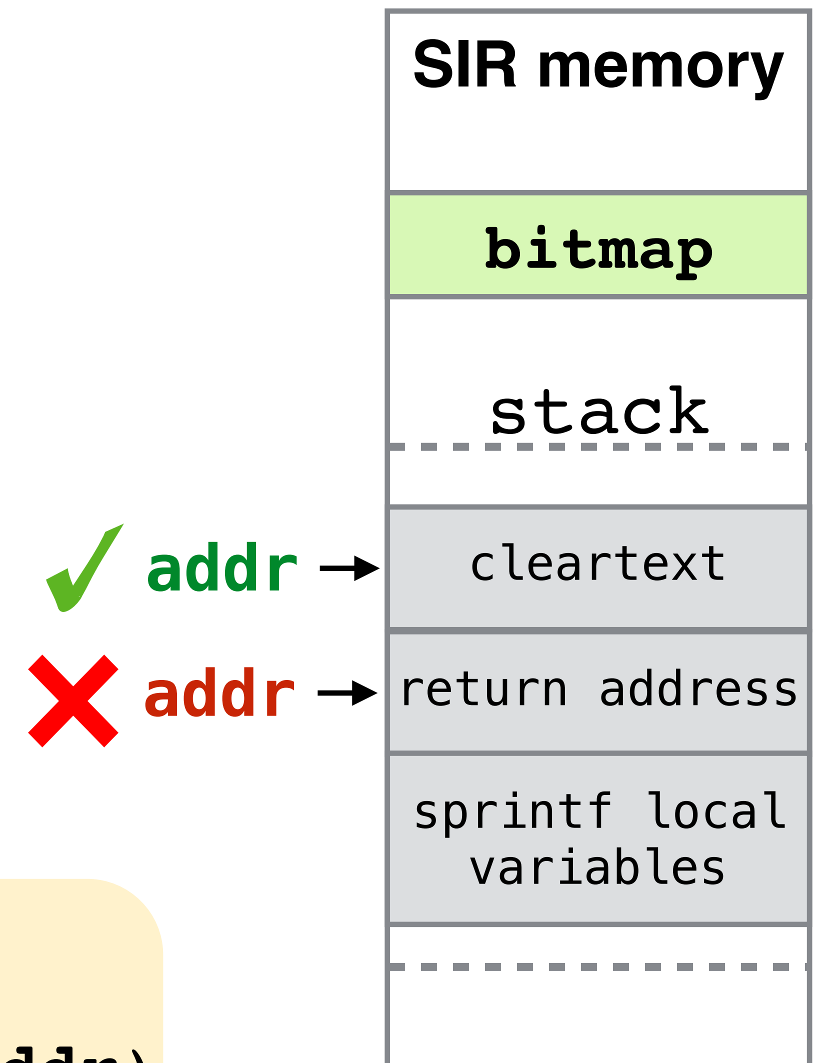
Verifying Calls in U

Verifying Writes in U

Verifying L

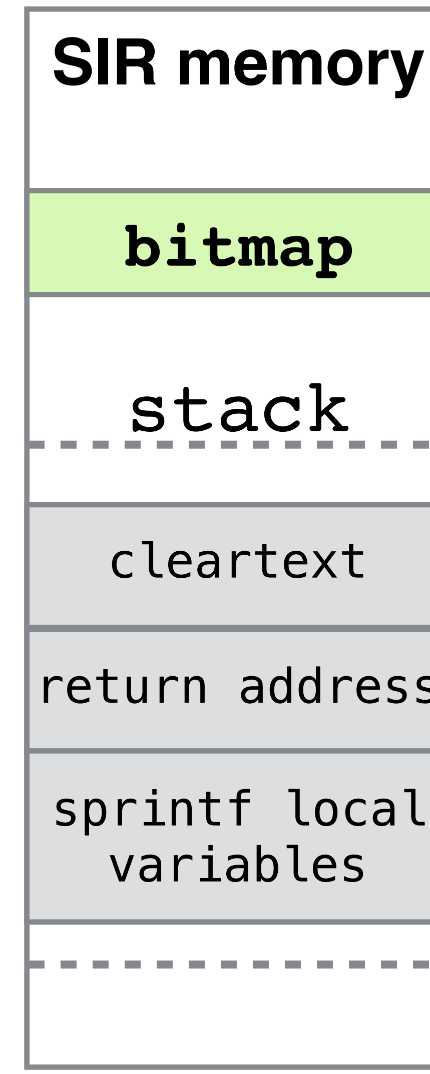
Evaluation

```
void Reduce(...) {
   ...
   sprintf(cleartext, “%s %lld”,...);
   ...
}

void sprintf(char *cleartext,...) {
   /* write to cleartext */
}
```



verifier **asserts writable(bitmap, addr)** for each store

Verifying U: Proof Obligations on `store`



Verifying U: Proof Obligations on `store`

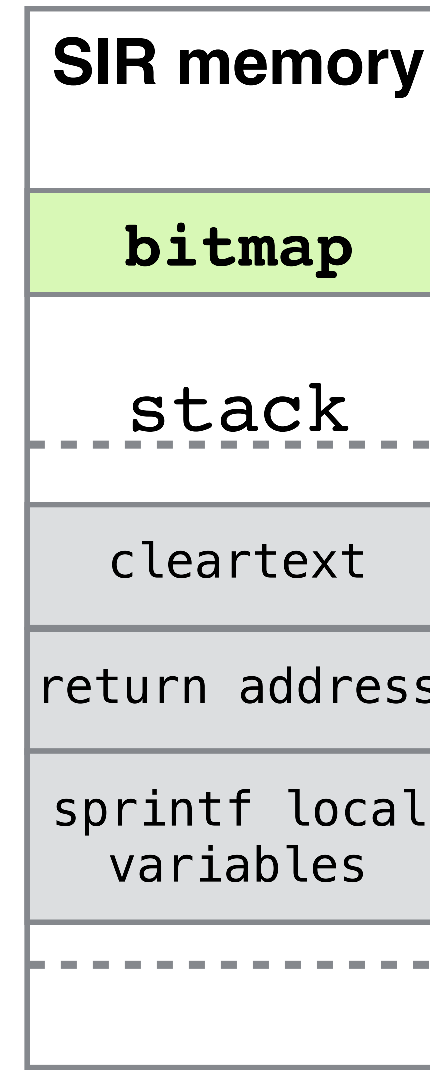
Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

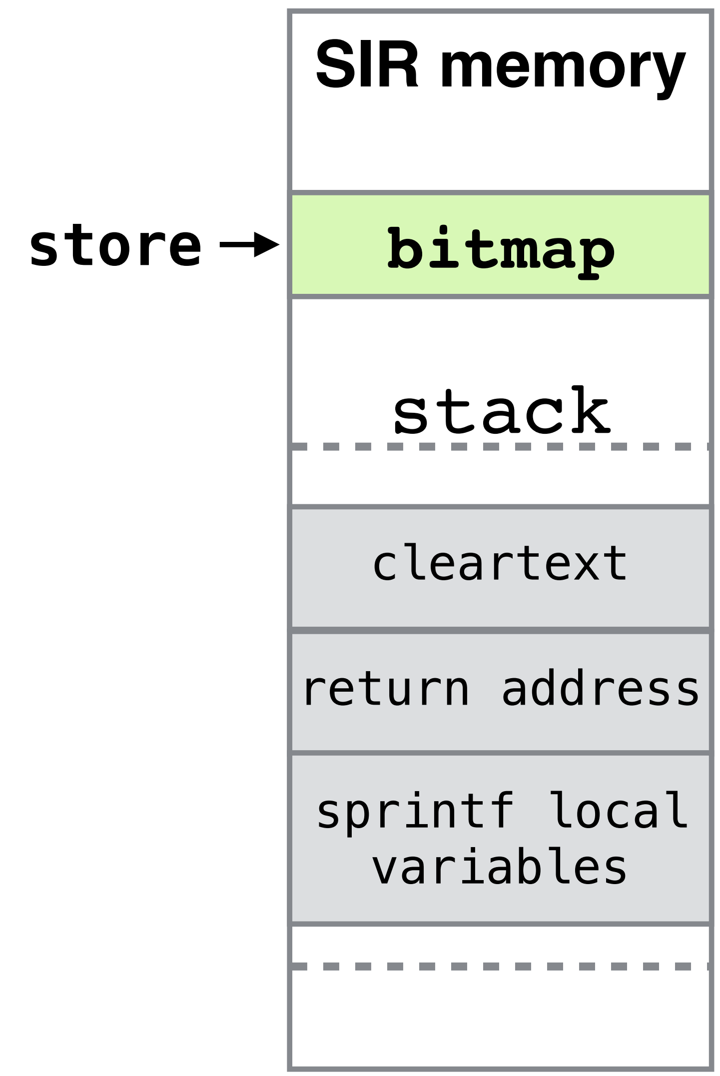
```
void Reduce(...) {
 ...

 sprintf(cleartext, “%s %lld”, ...);
 ...
}

void sprintf(char *cleartext, ...) {
  /* write to cleartext */
}
```



Verifying U: Proof Obligations on `store`

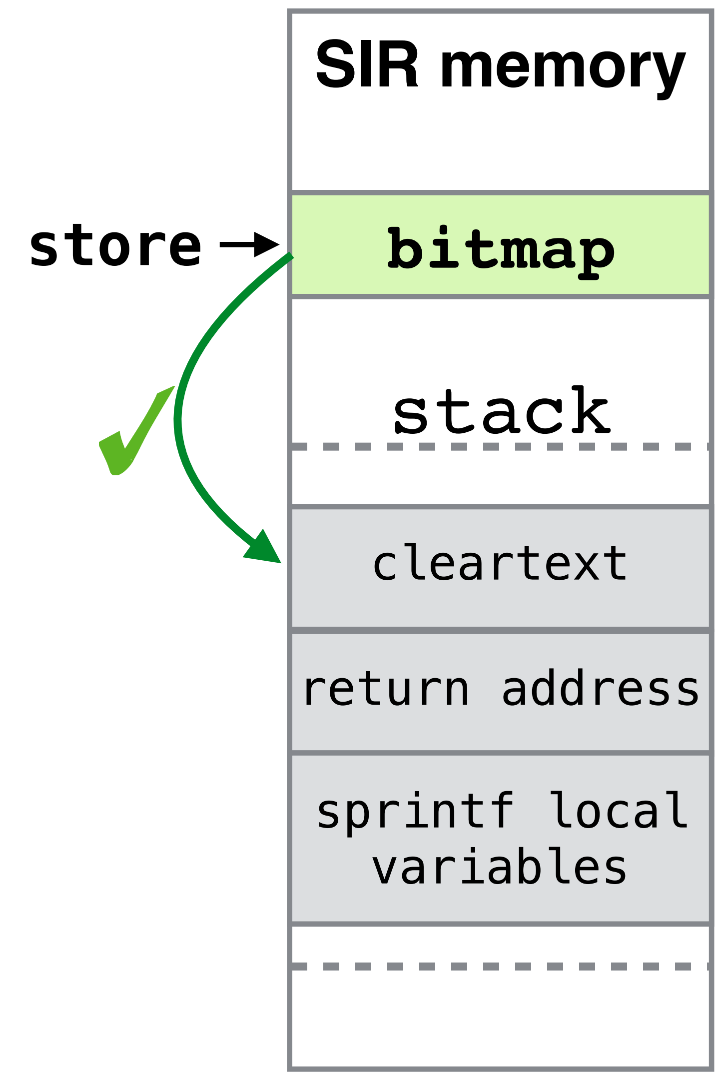
Verifying Calls in U
Verifying Writes in U
Verifying L
Evaluation

```
void Reduce(...) {
 ...
 <compiler makes cleartext writable>
 sprintf(cleartext, "%s %lld", ...);
 ...
}

void sprintf(char *cleartext, ...) {
  /* write to cleartext */
}
```



Verifying U: Proof Obligations on `store`

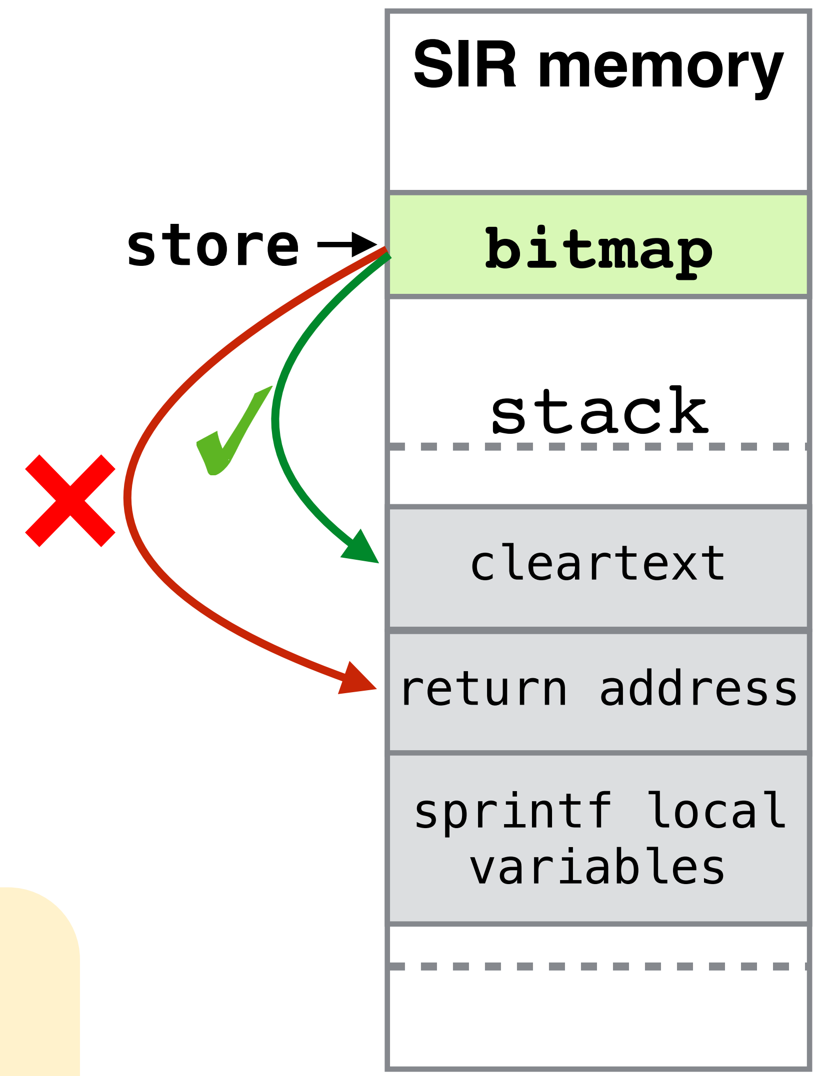

```
void Reduce(...) {
 ...
 <compiler makes cleartext writable>
 sprintf(cleartext, "%s %lld", ...);
 ...
}

void sprintf(char *cleartext, ...) {
  /* write to cleartext */
}
```



verifier **asserts** that
bitmap is updated safely

Verifying U: Proof Obligations on `store`

```
void Reduce(...) {
  ...
 <compiler makes cleartext writable>
  sprintf(cleartext, “%s %lld”, ...);
  ...
}

void sprintf(char *cleartext, ...) {
   /* write to cleartext */
}
```



verifier **asserts** that
bitmap is updated safely

Verifying U: Soundness

- **Verifying Calls in U**
- Verifying Writes in U
- Verifying L
- Evaluation

Verifying U: Soundness

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

`call:`

assert $\text{policy}(e) \;\wedge\; (\forall i.\ (\text{AddrInStack}(i) \wedge i < \text{rsp}) \Rightarrow \neg\text{writable}(\texttt{mem}, i))$

`ret:`

assert $(\text{rsp} = \text{old}(\text{rsp})) \wedge (\forall i.\ (\text{AddrInStack}(\text{i}) \wedge i < \text{old}(\text{rsp})) \Rightarrow \neg\text{writable}(\texttt{mem}, i));$

`jmp:`

assert $(\text{start}(p) \leq e < \text{end}(p)) \rightarrow \text{legal}(e);$

Verifying U: Soundness

- **Verifying Calls in U**
- Verifying Writes in U
- Verifying L
- Evaluation

`call:`

assert $\mathsf{policy}(e)$ … $\Rightarrow \neg\mathsf{writable}(\mathtt{mem}, i))$

`ret:`

assert $(\mathsf{rsp} = \mathsf{old}$ … $(\mathsf{rsp})) \Rightarrow \neg\mathsf{writable}(\mathtt{mem}, i));$

`jmp:`

assert $(\mathsf{start}(p) \leq e < \mathsf{end}(p)) \to \mathsf{legal}(e);$

Proof obligations imply WCFI ⇒

U calls into L via APIs

Verifying U: Preventing Writes Outside SIR

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

Verifying U: Preventing Writes Outside SIR

- Verifying Calls in U
- **Verifying Writes in U**
- Verifying L
- Evaluation

```
*q = buf + input;
*q = secret;
...
```

Verifying U: Preventing Writes Outside SIR

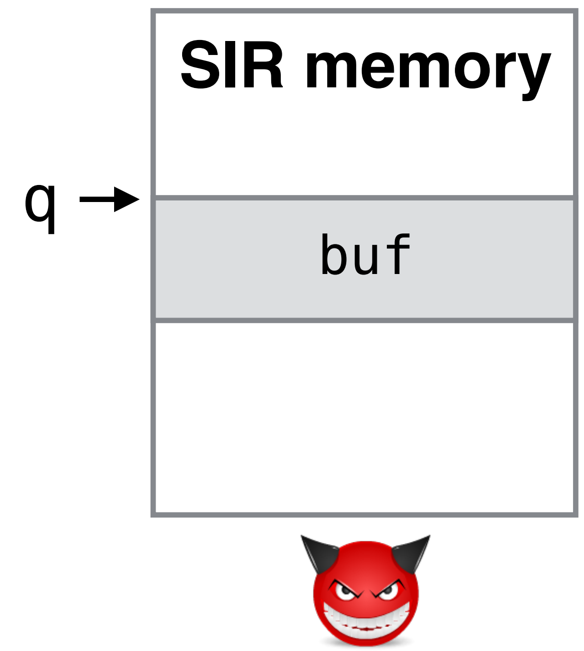
Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

```
*q = buf + input;
*q = secret;
...
```



Verifying U: Preventing Writes Outside SIR

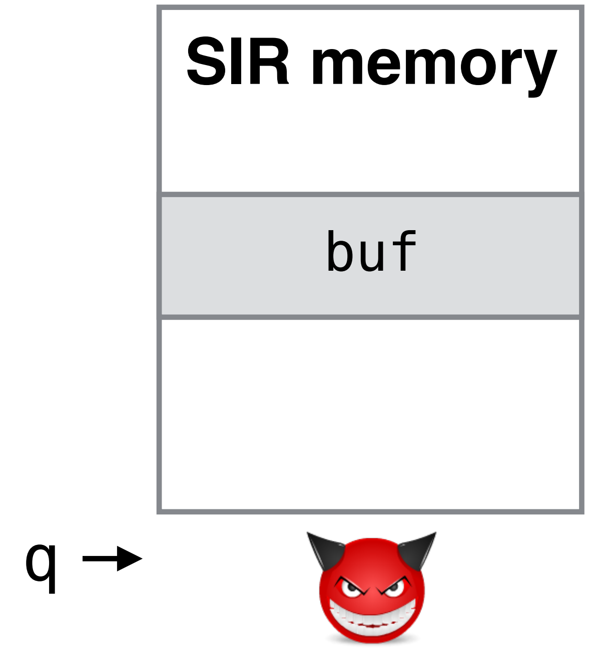
Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

```
*q = buf + input;
*q = secret;
...
```



Verifying U: Preventing Writes Outside SIR

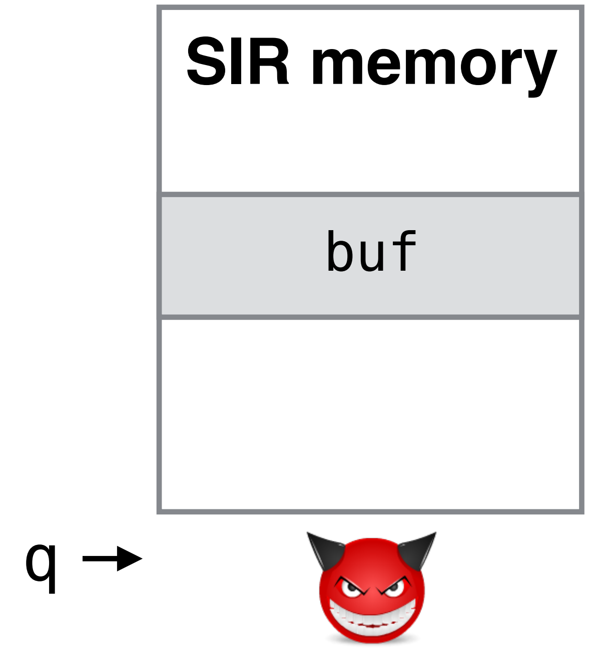
Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

```
*q = buf + input;
*q = secret;
...
```



Verifying U: Preventing Writes Outside SIR

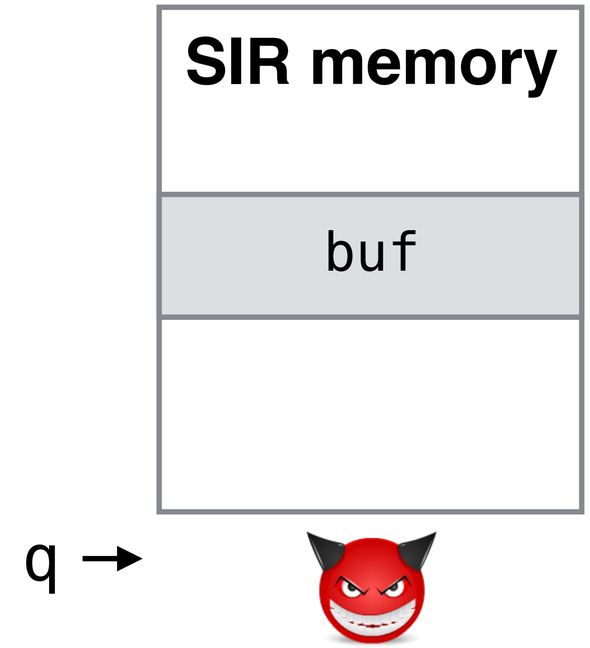
Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

```
*q = buf + input;
*q = secret;
...
```

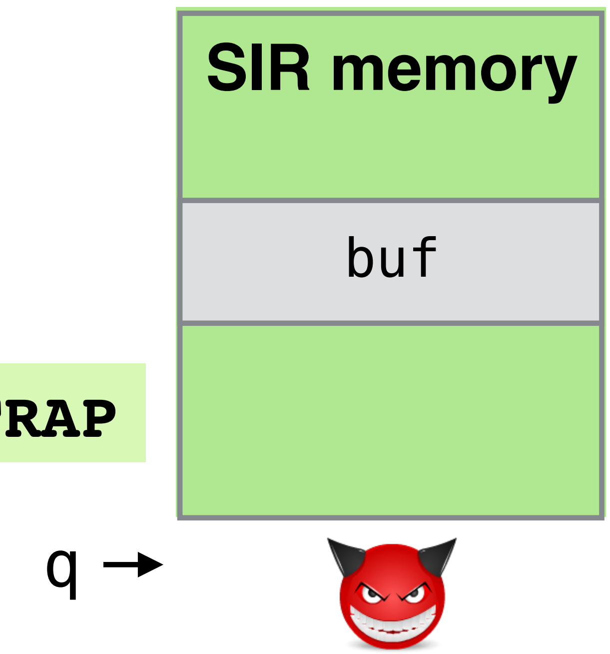


Runtime check using VC3 compiler:
is q within SIR range?

Verifying U: Preventing Writes Outside SIR


```
*q = buf + input;
*q = secret;
...
```

Software TRAP



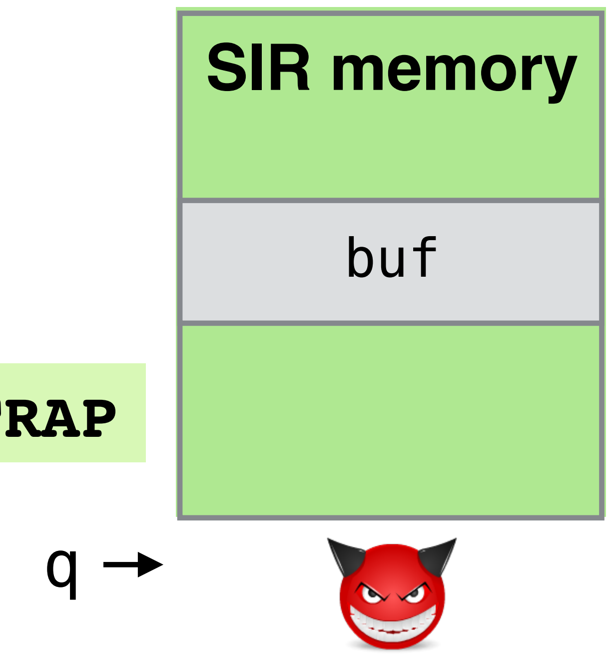
Runtime check using VC3 compiler:
is q within SIR range?

Verifying U: Preventing Writes Outside SIR

- Verifying Calls in U
- **Verifying Writes in U**
- Verifying L
- Evaluation

```
*q = buf + input;
*q = secret;
...
```

`Software TRAP`



`Runtime check using VC3 compiler:`
`is q within SIR range?`

Since we don't trust the compiler:

`verifier asserts addrInSIR(addr)`
`for each store`

Verifying U: Soundness

- Verifying Calls in U
- Verifying Writes in U
- Verifying L
- Evaluation

Boogie model

```
...
$L2: assume CF == 1;
     assert Ψ;
     mem := store(mem, rbx, rdx);
...
assert φ;
return;
```

Verifying U: Soundness

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

Boogie model

```
...
$L2: assume CF == 1;
     assert Ψ;
     mem := store(mem, rbx, rdx);
...
assert φ;
return;
```

Proof Obligations for WCFI

Verifying U: Soundness

- Verifying Calls in U
- Verifying Writes in U
- Verifying L
- Evaluation

Boogie model

```
...
$L2: assume CF == 1;
     assert Ψ;
     mem := store(mem, rbx, rdx);
...
assert φ;
return;
```

Proof Obligations for WCFI

Proof Obligations for writes within SIR

Verifying U: Soundness

- **Verifying Calls in U**
- **Verifying Writes in U**
- Verifying L
- Evaluation

Boogie model

```
...
$L2: assume CF == 1;
     assert Ψ;
     mem := store(mem, rbx, rdx);
...
assert φ;
return;
```

Proof Obligations for WCFI + **Proof Obligations for writes within SIR** = **IRC**

Verifying U: Modular Verification

- Verifying Calls in U
- Verifying Writes in U
- Verifying L
- Evaluation

Verifying U: Modular Verification

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

We perform modular reasoning of U's binary without false positives.

Verifying U: Modular Verification

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

We perform modular reasoning of U's binary without false positives.

The VC3 compiler generates enough runtime checks to allow this.

Verifying U: Modular Verification

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

We perform modular reasoning of U's binary without false positives.

The VC3 compiler generates enough runtime checks to allow this.

```
void Reduce(...) {
   ...
   sprintf(cleartext, “%s %lld”, …);
   ...
}
void sprintf(char *cleartext, …) {

   //write to cleartext, which is stack-allocated
}
```

Verifying U: Modular Verification

We perform modular reasoning of U's binary without false positives.

The VC3 compiler generates enough runtime checks to allow this.

```
void Reduce(...) {
   ...
   sprintf(cleartext, "%s %lld", …);
   ...
}
void sprintf(char *cleartext, …) {
    <runtime check that buf is in SIR memory>
   //write to cleartext, which is stack-allocated
}
```

Verification Optimizations

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

Verification Optimizations

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

`ret:` `assert`$(\forall i \ : i < \mathsf{rsp} \Rightarrow \neg \mathsf{writable}(\mathsf{mem}, \mathsf{i}))$

Verification Optimizations

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

`ret:` `assert`$(\forall i : i < \mathsf{rsp} \Rightarrow \neg\mathsf{writable}(\mathsf{mem}, \mathsf{i}))$

sound strategy for quantifier instantiation

Verification Optimizations

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

`ret:` `assert`$(\forall i : i < \mathsf{rsp} \Rightarrow \neg\mathsf{writable}(\mathsf{mem}, i))$

sound strategy for quantifier instantiation

split into disjoint arrays for bitmap and stack

Verification Optimizations

`ret:` `assert` $(\forall i : i < \mathsf{rsp} \Rightarrow \neg \mathsf{writable}(\mathsf{mem}, i))$

sound strategy for quantifier instantiation

split into disjoint arrays for bitmap and stack

Removed hundreds of Z3 timeouts in our experiments

Manual, One-time Verification of L

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

Manual, One-time Verification of L

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

```
void send(void *buf, size_t size)
void recv(void *buf, size_t size)
```

Manual, One-time Verification of L

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

```
void send(void *buf, size_t size)
void recv(void *buf, size_t size)
// ensures no unsafe modification to U
```

Manual, One-time Verification of L

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

```
void send(void *buf, size_t size)
void recv(void *buf, size_t size)
// ensures no unsafe modification to U
// ensures channel key is not modified
```

Manual, One-time Verification of L

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

```
void send(void *buf, size_t size)
void recv(void *buf, size_t size)
// ensures no unsafe modification to U
// ensures channel key is not modified
// ensures ...
```

Manual, One-time Verification of L

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

```
void send(void *buf, size_t size)
void recv(void *buf, size_t size)
// ensures no unsafe modification to U
// ensures channel key is not modified
// ensures ...
```

Manual, One-time Verification of L

- Verifying Calls in U
- Verifying Writes in U
- **Verifying L**
- Evaluation

```
void send(void *buf, size_t size)
void recv(void *buf, size_t size)
// ensures no unsafe modification to U
// ensures channel key is not modified
// ensures ...

void *malloc(size_t size)
void free(void *buf)
```

Manual, One-time Verification of L

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

```
void send(void *buf, size_t size)
void recv(void *buf, size_t size)
// ensures no unsafe modification to U
// ensures channel key is not modified
// ensures ...

void *malloc(size_t size)
void free(void *buf)
// ensures ...
```

Manual, One-time Verification of L

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

```
void send(void *buf, size_t size)
void recv(void *buf, size_t size)
// ensures no unsafe modification to U
// ensures channel key is not modified
// ensures ...

void *malloc(size_t size)
void free(void *buf)
// ensures ...
```

Manual, One-time Verification of L

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

```
void send(void *buf, size_t size)
void recv(void *buf, size_t size)
// ensures no unsafe modification to U
// ensures channel key is not modified
// ensures ...

void *malloc(size_t size)
void free(void *buf)
// ensures ...
```

No requires clause on U

Evaluation

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

Evaluation

`Runtime checks incur 15% performance hit [Schuster et al.: VC3]`

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

Evaluation

Runtime checks incur 15% performance hit [Schuster et al.: VC3]

Benchmark	Code Size	Verified Asserts	Timed out Asserts	False Positives
UserUsage	14 KB	2125	2	4
IoVolumes	17 KB	2391	2	0
Revenue	18 KB	1534	3	0
lbm	38 KB	1192	0	0
astar	115 KB	6468	2	0
bzip2	155 KB	10287	36	0

timeout:
30 mins

Evaluation

`Runtime checks incur 15% performance hit [Schuster et al.: VC3]`

Benchmark	Code Size	Verified Asserts	Timed out Asserts	False Positives
UserUsage	14 KB	2125	2	4
IoVolumes	17 KB	2391	2	0
Revenue	18 KB	1534	3	0
lbm	38 KB	1192	0	0
astar	115 KB	6468	2	0
bzip2	155 KB	10287	36	0

`timeout: 30 mins`

Evaluation

Runtime checks incur 15% performance hit [Schuster et al.: VC3]

Benchmark	Code Size	Verified Asserts	Timed out Asserts	False Positives
UserUsage	14 KB	2125	2	4
IoVolumes	17 KB	2391	2	0
Revenue	18 KB	1534	3	0
lbm	38 KB	1192	0	0
astar	115 KB	6468	2	0
bzip2	155 KB	10287	36	0

timeout:
30 mins

verified
in 4 hours

Verifying Calls in U

Verifying Writes in U

Verifying L

Evaluation

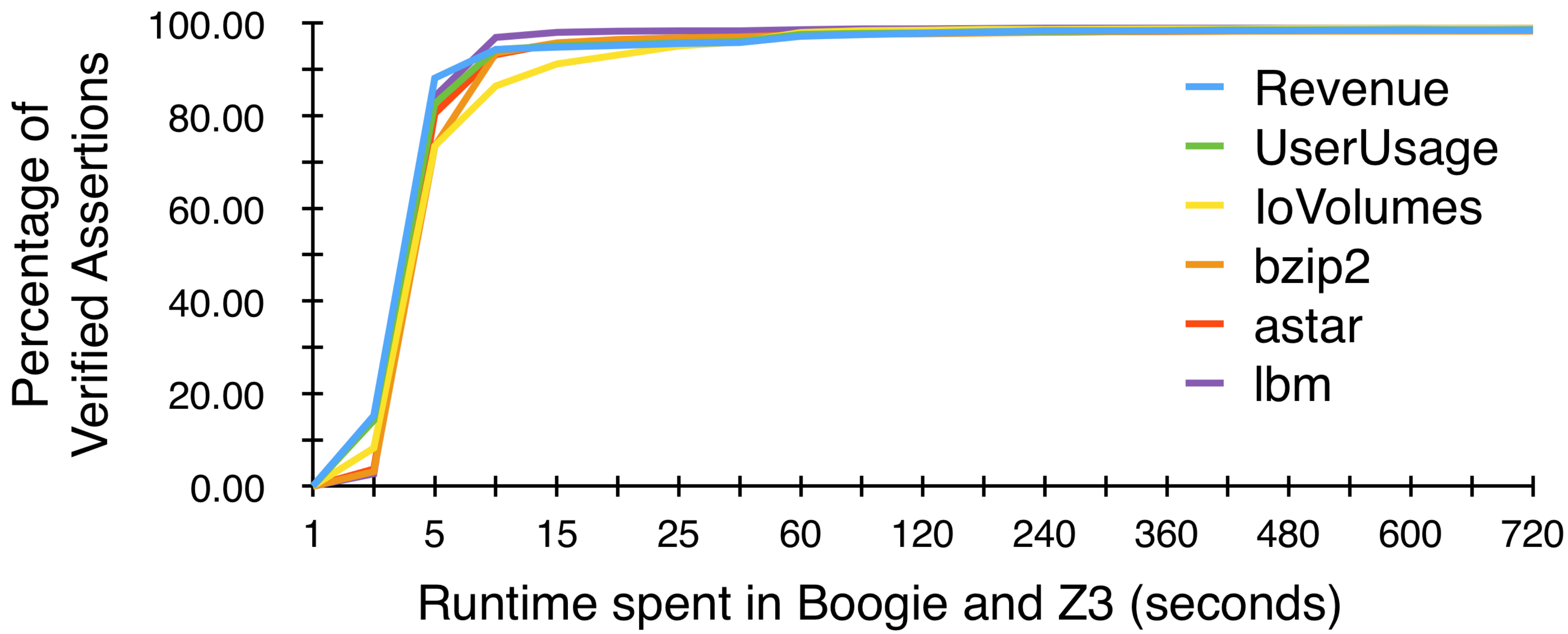
Evaluation

Runtime checks incur 15% performance hit [Schuster et al.: VC3]

Benchmark	Code Size	Verified Asserts	Timed out Asserts	False Positives
UserUsage	14 KB	2125	2	$\boxed{4}$
IoVolumes	17 KB	2391	2	0
Revenue	18 KB	1534	3	0
lbm	38 KB	1192	0	0
astar	115 KB	6468	2	0
bzip2	155 KB	10287	36	0

timeout: 30 mins

verified in 4 hours



Related Work

Related Work

Work	Application	Technique	Comparison

Related Work

Work	Application	Technique	Comparison
Sinha et al: Moat *CCS'15*	Verifying Confidentiality of SGX Programs	Refinement type system for non-interference	Requires global, precise tracking of secrets in machine code

Related Work

Work	Application	Technique	Comparison
Sinha et al: Moat *CCS'15*	Verifying Confidentiality of SGX Programs	Refinement type system for non-interference	Requires global, precise tracking of secrets in machine code
Myers et al: JIF *SOSP'97*	Information Flow for Java applications	Type system for non-interference	Requires annotations, and trust in language runtime

Related Work

Work	Application	Technique	Comparison
Sinha et al: Moat *CCS'15*	Verifying Confidentiality of SGX Programs	Refinement type system for non-interference	Requires global, precise tracking of secrets in machine code
Myers et al: JIF *SOSP'97*	Information Flow for Java applications	Type system for non-interference	Requires annotations, and trust in language runtime
Hawblitzel et al: IronClad Apps *OSDI'14*	Functional correctness	Deductive verification	Requires manual effort in writing invariants

Related Work

Work	Application	Technique	Comparison
Sinha et al: Moat *CCS'15*	Verifying Confidentiality of SGX Programs	Refinement type system for non-interference	Requires global, precise tracking of secrets in machine code
Myers et al: JIF *SOSP'97*	Information Flow for Java applications	Type system for non-interference	Requires annotations, and trust in language runtime
Hawblitzel et al: IronClad Apps *OSDI'14*	Functional correctness	Deductive verification	Requires manual effort in writing invariants
Morisett et al: Rocksalt *PLDI'12*	Software Fault Isolation	Verified Machine Code Checker extracted from Coq	different goal 64-bit version requires 100GB address space

Takeaway Points

Takeaway Points

IRC as a design principle for SIRs:

Takeaway Points

IRC as a design principle for SIRs:

- easier to verify than full functional correctness

Takeaway Points

IRC as a design principle for SIRs:

- easier to verify than full functional correctness
- avoids tracking of secrets in SIR's memory

Takeaway Points

IRC as a design principle for SIRs:

- easier to verify than full functional correctness
- avoids tracking of secrets in SIR’s memory

Automatic, modular verification of IRC on SIR binaries, with a small trusted computing base

Takeaway Points

Fork me on GitHub

IRC as a design principle for SIRs:

- easier to verify than full functional correctness
- avoids tracking of secrets in SIR's memory

Automatic, modular verification of IRC on SIR binaries, with a small trusted computing base

https://github.com/TrustedCloud/slashconfidential