A Design and Verification Methodology for Secure Isolated Regions

Rohit Sinha¹, Manuel Costa², Akash Lal², Nuno P. Lopes², Sriram Rajamani², Sanjit A. Seshia¹, Kapil Vaswani²

¹Berkeley, UNIVERSITY OF CALIFORNIA
²Microsoft Research

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 */ }

SIR memory is protected: only SIR code can access it

Trusted Computing Base includes the SIR and CPU hardware

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

SIR
Hadoop

Secure Hardware
Intel
SGX
Challenges in Proving Confidentiality

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

- SIR
- Hadoop
- Secure Hardware
  - Intel SGX
void Reduce(byte *kEnc, byte *vEnc) {
    KeyAesGcm *aesKey = ProvisionKey();
}
void Reduce(byte *kEnc, byte *vEnc)
{
    KeyAesGcm *aesKey = ProvisionKey();

    char k[..];
    aesKey->Decrypt(kEnc, k);
    char v[..];
    aesKey->Decrypt(vEnc, v);
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);
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);
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);
}
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);
}
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);
}
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
Information Release Confinement

SIR  Hadoop

5
Information Release Confinement

L implements `send`, `recv`, malloc, free
Information Release Confinement

L implements `send`, `recv`, `malloc`, `free`
Information Release Confinement

L implements \texttt{send}, \texttt{recv}, \texttt{malloc}, \texttt{free}

...f12e6dafa9d3b

\texttt{ciphertext}

...c96a8ebbd4a7e

\texttt{messages}
Information Release Confinement

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

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

```c
void Reduce(byte *kEnc, byte *vEnc) {
    char *k = recv(..);
    char *v = recv(..);
}
```
Information Release Confinement

IRC: All updates to non-SIR memory via L’s **send** API

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

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

```c
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

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

```c
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

IRC: All updates to non-SIR memory via L’s \texttt{send} API

Hadoop

\begin{center}
\begin{tikzpicture}
\node (SIR) at (0,0) {
\begin{minipage}{0.5\textwidth}
\begin{itemize}
\item \texttt{send(secret)}
\end{itemize}
\end{minipage}};
\node (L) at (SIR.east) {
\begin{minipage}{0.3\textwidth}
\textbf{Runtime Library (L)}
\end{minipage}};
\node (U) at (SIR.east) {
\begin{minipage}{0.2\textwidth}
\textbf{User Code (U)}
\end{minipage}};
\node (Hadoop) at (SIR.east) {
\begin{minipage}{0.3\textwidth}
L implements \texttt{send, recv, malloc, free}
\end{minipage}};
\end{tikzpicture}
\end{center}
Information Release Confinement

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

✓ Prevents explicit information leaks: side channels outside scope

L implements send, recv, malloc, free
Information Release Confinement

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

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

\textit{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

send, recv implement a cryptographically secure channel

Correctness of U

U calls into L at API entrypoints
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
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
send, recv
implement a
cryptographically secure channel

U calls into L
at API entrypoints

Correctness of U
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

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**
- SIR satisfies IRC
- Correctness of L
- Correctness of U
- send, recv implement a channel with secrecy and integrity
- 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 that U calls into L at API entrypoints
This Talk: Automatically Proving IRC

Verifying that U calls into L at API entrypoints
Verifying that U does not modify non-SIR memory
This Talk: Automatically Proving IRC

- Verifying that U calls into L at API entrypoints
- Verifying that U does not modify non-SIR memory
- Correctness properties of L
This Talk: Automatically Proving IRC

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

Potential Code in U

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

<table>
<thead>
<tr>
<th>SIR memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
</tr>
<tr>
<td>return addr</td>
</tr>
</tbody>
</table>
Verifying U: Calls to L via API Entrypoints

Potential Code in U

*\( q = buf + \text{input}; \)
*\( q = \text{input2}; \)
...
\( \text{return;} \)

SIR memory

\( q \rightarrow \)

\( \begin{array}{c}
\text{buf} \\
\text{return addr}
\end{array} \)
Verifying U: Calls to L via API Entrypoints

Potential Code in U

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

SIR memory

```
buf

return addr
```

Verification

- Verifying Calls in U
- Verifying Writes in U
- Verifying L
Verifying U: Calls to L via API Entrypoints

Potential Code in U

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

SIR memory

buf

q

return addr
Verifying U: Calls to L via API Entrypoints

Potential Code in U

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

SIR memory

buf

return addr

arbitrary code

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;
```

SIR memory

```
buf
return addr
arbitrary code
```

- 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;

SIR memory

buf
return addr
arbitrary code

arbitrary code
- middle of x86 instructions
- arbitrary instructions in L
Verifying U: Calls to L via API Entrypoints

### Potential Code in U

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

### Evaluation

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

### Diagram

- **SIR memory**
  -,buf
  - arbitrary code
  - arbitrary code
  - return addr

- **arbitrary code**
  - 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;

SIR memory

buf

return addr

arbitrary code

Control Flow Integrity

- middle of x86 instructions
- arbitrary instructions in L

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;

SIR memory

buf
return addr
arbitrary code

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**

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

**SIR memory**

- buf
- return addr
- arbitrary code

**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

- 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;

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

SIR memory

buf

return addr

arbitrary code

arbitrary code
- 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;

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

\[
\begin{align*}
*q &= \text{buf} + \text{input}; \\
*q &= \text{input2}; \\
... \\
\text{return;}
\end{align*}
\]

Weak Control Flow Integrity (WCFI)

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

SIR memory

\[
\begin{array}{c}
\text{buf} \\
\text{return addr} \\
\text{arbitrary code}
\end{array}
\]

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

WCFI \Rightarrow
U calls into L via APIs
Verifying U: Runtime Checks for WCFI

Potential Code in U

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

SIR memory

buf
return addr
Verifying U: Runtime Checks for WCFI

Potential Code in U

*\(q = buf + \text{input};\)

*\(q = \text{input2};\)

... return;

Runtime check using VC3 compiler:
is address of \(q\) marked writable in bitmap?
Verifying U: Runtime Checks for WCFI

Potential Code in U

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

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

- not-writable
- return addresses
Verifying U: Runtime Checks for WCFI

Potential Code in U

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

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

- not-writable
- writable
- return addresses
- local vars
- heap objects

SIR memory

<table>
<thead>
<tr>
<th>buf</th>
</tr>
</thead>
<tbody>
<tr>
<td>return addr</td>
</tr>
</tbody>
</table>

bitmap
Verifying U: Runtime Checks for WCFI

Potential Code in U

*\(q = \text{buf} + \text{input};\)

*\(q = \text{input2};\)

\(...\)

return;

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

- not-writable
- writable

- return addresses
- local vars
- heap objects
Verifying U: Runtime Checks for WCFI

Potential Code in U

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

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

- not-writable
  - return addresses
- writable
  - local vars
  - heap objects
Verifying U: Runtime Checks for WCFI

Potential Code in U

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

Runtime check using VC3 compiler:

is address of q marked writable in bitmap?

- not-writable
- writable
- return addresses
- local vars
- heap objects

SIR memory

- buf
- return addr
- bitmap
Potential Code in U

*\(q = buf + \text{input};\)

*\(q = \text{input}2;\)

... Software TRAP

return;

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

- **not-writable**: return addresses
- **writable**: local vars, heap objects
Verifying U: Modeling U’s Binary
Verifying U: Modeling U’s Binary

Avoid trusting the compiler:
Long history of bugs
Verifying U: Modeling U’s Binary

**Avoid trusting the compiler:**
- Long history of bugs
- Compiler optimizes away many runtime checks
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

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

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

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 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

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 U: Modeling the Adversary

load from untrusted memory returns arbitrary value *
load(mem,a) = ITE(SIR(a), mem[a], *);
Verifying U: Modeling the Adversary

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

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

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 \Psi;
    mem := store(mem, rbx, rdx);
...
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 \Psi;
    mem := store(mem, rbx, rdx);
...
assert \phi;
return;
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 $\Psi$;
    mem := store(mem, rbx, rdx);
...
assert $\varphi$;
return;

Proof Obligations guarantee WCFI \Rightarrow
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 \Psi;  
    mem := store(mem, rbx, rdx);  
...
assert \varphi;  
return;

Proof Obligations guarantee WCFI ⇒  
U calls into L via APIs
Verifying U: Proof Obligations

Boogie model

```plaintext
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 \Psi;
mem := store(mem, rbx, rdx);
... assert \phi;
return;
```

Proof Obligations guarantee WCFI ⇒ U calls into L via APIs

Presence of runtime checks helps SMT solver to prove \textbf{assert } \Psi
Verifying U: Proof Obligations on store

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

SIR memory

- bitmap
- stack
- cleartext
void Reduce(...) {
    ...
    sprintf(cleartext, "%s %lld", ...);
    ...
}

void sprintf(char *cleartext, ...) {
    /* write to cleartext */
}
Verifying U: Proof Obligations on \textit{store}

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

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

[SIR memory]

- \textit{bitmap}
- \textit{stack}
- \textit{cleartext}
- \textit{return address}
- \textit{sprintf local variables}
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 \textit{store}

\begin{verbatim}
void Reduce(...) {
    ...
    sprintf(cleartext, "\%s \%lld", ...);
    ...
}
void sprintf(char *cleartext, ...) {
    /* write to cleartext */
}
\end{verbatim}

\texttt{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

SIR memory
- bitmap
- stack
  - cleartext
  - return address
  - sprintf local variables
Verifying U: Proof Obligations on store

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

void sprintf(char *cleartext, ...) {
    /* write to cleartext */
}
```
Verifying U: Proof Obligations on \texttt{store}

```c
void Reduce(...) {
    ...
    \texttt{<compiler makes cleartext writable>}  \texttt{store}\rightarrow  \texttt{bitmap}
    sprintf(cleartext, "%s %lld", ...);
    ...
}

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

- **SIR memory**
  - `store` → `bitmap`
  - `stack`
    - `cleartext`
    - `return address`
    - `sprintf local variables`
Verifying U: Proof Obligations on \texttt{store}

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

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

\texttt{verifier asserts that \texttt{bitmap} is updated safely}
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 U: Soundness

\textbf{call}:
assert \text{policy}(e) \land (\forall i. (\text{AddrInStack}(i) \land i < \text{rsp}) \Rightarrow \neg \text{writable}(\text{mem}, i))

\textbf{ret}:
assert (\text{rsp} = \text{old(rsp)}) \land (\forall i. (\text{AddrInStack}(i) \land i < \text{old(rsp)}) \Rightarrow \neg \text{writable}(\text{mem}, i));

\textbf{jmp}:
assert (\text{start}(p) \leq e < \text{end}(p)) \rightarrow \text{legal}(e);
Verifying U: Soundness

\textbf{call:}
\begin{align*}
\text{assert } \text{policy}(e) \land (Y_i \land \text{AddlStock}(i) \land i < \text{esp}) \Rightarrow \neg \text{writable}(\text{mem}, i))
\end{align*}

\textbf{ret:}
\begin{align*}
\text{assert } (\text{rsp} = \text{old}) \land (\text{old} < \text{esp}) \land (\text{esp} < \text{esp}) \Rightarrow \neg \text{writable}(\text{mem}, i));
\end{align*}

\textbf{jmp:}
\begin{align*}
\text{assert } (\text{start}(p) \leq e < \text{end}(p)) \Rightarrow \text{legal}(e);
\end{align*}

\begin{center}
\textbf{Proof obligations imply WCFI} \Rightarrow \\
\textbf{U calls into L via APIs}
\end{center}
Verifying U: Preventing Writes Outside SIR
Verifying U: Preventing Writes Outside SIR

*\( q = buf + input; \)
*\( q = secret; \)
...

SIR memory

<table>
<thead>
<tr>
<th>buf</th>
</tr>
</thead>
</table>

Verifying Calls in U
Verifying Writes in U
Verifying L
Evaluation
Verifying U: Preventing Writes Outside SIR

*\(q = \text{buf} + \text{input};\)
*\(q = \text{secret};\)

...
Verifying U: Preventing Writes Outside SIR

*\( q = buf + \text{input}; \)
*\( q = \text{secret}; \)
...

SIR memory

buf

q ->

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;
...
```

SIR memory

<table>
<thead>
<tr>
<th>buf</th>
</tr>
</thead>
</table>

q →
Verifying U: Preventing Writes Outside SIR

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

```c
*q = buf + input;
*q = secret;
...
```
Verifying U: Preventing Writes Outside SIR

Software TRAP

Runtime check using VC3 compiler:
is q within SIR range?
Verifying U: Preventing Writes Outside SIR

Since we don’t trust the compiler:

```c
verifier asserts addrInSIR(addr) for each store
```
Verifying U: Soundness

Boogie model

...  
$L2$: assume CF == 1;  
  
  assert $\Psi$;  
  mem := store(mem, rbx, rdx);  

...  
assert $\varphi$;  
return;
Boogie model

... $L2$: assume $CF == 1$;
  assert $\Psi$;
  mem := store(mem, rbx, rdx);
...
  assert $\varphi$;
  return;

Proof Obligations for WCFI
Verifying U: Soundness

Boogie model

... 
$L2$: assume $CF == 1$;

assert $\Psi$;

mem := store(mem, rbx, rdx);

... 
assert $\varphi$;

return;

Proof Obligations for WCFI

Proof Obligations for writes within SIR
Verifying U: Soundness

Boogie model

... $L2$: assume \( CF = 1 \);
    assert \( \Psi \);
    \text{mem} := \text{store} (\text{mem}, \text{rbx}, \text{rdx});

...
    assert \( \phi \);
    return;

Proof Obligations for WCFI + Proof Obligations for writes within SIR = IRC
Verifying U: Modular Verification
Verifying U: Modular Verification

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

```c
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.

```c
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

\[
\text{ret: } \text{assert}(\forall i : i < \text{rsp} \Rightarrow \neg \text{writable}(\text{mem}, i))
\]
Verification Optimizations

\[ \text{ret: assert } (\forall i : i < \text{rsp} \Rightarrow \neg \text{writable}(\text{mem}, i)) \]

sound strategy for quantifier instantiation
Verification Optimizations

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

- sound strategy for quantifier instantiation
- split into disjoint arrays for bitmap and stack
Verification Optimizations

\[ \text{ret: assert} \left( \forall i : i < \text{rsp} \Rightarrow \neg \text{writable(mem, } i) \right) \]

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

- 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
Manual, One-time Verification of L

```c
void send(void *buf, size_t size)
void recv(void *buf, size_t size)
```
Manual, One-time Verification of L

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

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

```c
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

```c
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

```c
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

```c
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 ...
```
Manually, One-time Verification of L

```c
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

```c
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
Evaluation

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

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Code Size</th>
<th>Verified Asserts</th>
<th>Timed out Asserts</th>
<th>False Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserUsage</td>
<td>14 KB</td>
<td>2125</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>IoVolumes</td>
<td>17 KB</td>
<td>2391</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Revenue</td>
<td>18 KB</td>
<td>1534</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>lbm</td>
<td>38 KB</td>
<td>1192</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>astar</td>
<td>115 KB</td>
<td>6468</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>bzip2</td>
<td>155 KB</td>
<td>10287</td>
<td>36</td>
<td>0</td>
</tr>
</tbody>
</table>

timeout: 30 mins
## Evaluation

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Code Size</th>
<th>Verified Asserts</th>
<th>Timed out Asserts</th>
<th>False Positives</th>
<th>timeout: 30 mins</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserUsage</td>
<td>14 KB</td>
<td>2125</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>IoVolumes</td>
<td>17 KB</td>
<td>2391</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Revenue</td>
<td>18 KB</td>
<td>1534</td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>ibm</td>
<td>38 KB</td>
<td>1192</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>astar</td>
<td>115 KB</td>
<td>6468</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>bzip2</td>
<td>155 KB</td>
<td>10287</td>
<td>36</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Evaluation

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Code Size</th>
<th>Verified Asserts</th>
<th>Timed out Asserts</th>
<th>False Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserUsage</td>
<td>14 KB</td>
<td>2125</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>IoVolumes</td>
<td>17 KB</td>
<td>2391</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Revenue</td>
<td>18 KB</td>
<td>1534</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>ibm</td>
<td>38 KB</td>
<td>1192</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>astar</td>
<td>115 KB</td>
<td>6468</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>bzip2</td>
<td>155 KB</td>
<td>10287</td>
<td>36</td>
<td>0</td>
</tr>
</tbody>
</table>
Evaluation

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Code Size</th>
<th>Verified Asserts</th>
<th>Timed out Asserts</th>
<th>False Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserUsage</td>
<td>14 KB</td>
<td>2125</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>IoVolumes</td>
<td>17 KB</td>
<td>2391</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Revenue</td>
<td>18 KB</td>
<td>1534</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>ibm</td>
<td>38 KB</td>
<td>1192</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>astar</td>
<td>115 KB</td>
<td>6468</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>bzip2</td>
<td>155 KB</td>
<td>10287</td>
<td>36</td>
<td>0</td>
</tr>
</tbody>
</table>

timeout: 30 mins
verified in 4 hours
Related Work
## Related Work

<table>
<thead>
<tr>
<th>Work</th>
<th>Application</th>
<th>Technique</th>
<th>Comparison</th>
</tr>
</thead>
</table>


<table>
<thead>
<tr>
<th>Work</th>
<th>Application</th>
<th>Technique</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sinha et al: Moat</td>
<td>Verifying Confidentiality of SGX Programs</td>
<td>Refinement type system for non-interference</td>
<td>Requires global, precise tracking of secrets in machine code</td>
</tr>
<tr>
<td>Work</td>
<td>Application</td>
<td>Technique</td>
<td>Comparison</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------------------------</td>
<td>----------------------------------------</td>
<td>------------------------------------------------------</td>
</tr>
<tr>
<td>Sinha et al: Moat</td>
<td>Verifying Confidentiality of</td>
<td>Refinement type system for non-interference</td>
<td>Requires global, precise tracking of secrets in machine code</td>
</tr>
<tr>
<td>CCS’15</td>
<td>SGX Programs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Myers et al: JIF</td>
<td>Information Flow for Java</td>
<td>Type system for non-interference</td>
<td>Requires annotations, and trust in language runtime</td>
</tr>
<tr>
<td>SOSP’97</td>
<td>applications</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Related Work

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

<table>
<thead>
<tr>
<th>Work</th>
<th>Application</th>
<th>Technique</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sinha et al: Moat</td>
<td>Verifying Confidentiality of SGX Programs</td>
<td>Refinement type system for non-interference</td>
<td>Requires global, precise tracking of secrets in machine code</td>
</tr>
<tr>
<td>Myers et al: JIF</td>
<td>Information Flow for Java applications</td>
<td>Type system for non-interference</td>
<td>Requires annotations, and trust in language runtime</td>
</tr>
<tr>
<td>Hawblitzel et al: IronClad Apps</td>
<td>Functional correctness</td>
<td>Deductive verification</td>
<td>Requires manual effort in writing invariants</td>
</tr>
<tr>
<td>Morisett et al: Rocksalt</td>
<td>Software Fault Isolation</td>
<td>Verified Machine Code Checker extracted from Coq</td>
<td>different goal 64-bit version requires 100GB address space</td>
</tr>
</tbody>
</table>

**CCS’15**, **SOSP’97**, **OSDI’14**, **PLDI’12**
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

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