Linear capabilities for modular fully-abstract compilation of verified code

Thomas Van Strydonck Dominique Devriese Frank Piessens

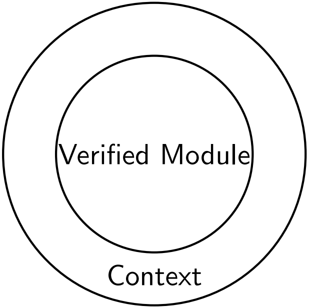
KU Leuven

thomas.vanstrydonck@cs.kuleuven.be

May 2, 2018

Preserving sound modular verification

- Separation logic in verification tools
  - Sound
  - Modular
- **Problem**: Guarantees *lost* in untrusted context
- **Solution**: Compiler enforces separation logic contracts



The compiler

Source language

- Regular verified C code
- Separation logic annotated
  - e.g. VeriFast syntax for concreteness

Target language

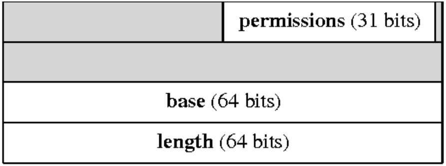
- Regular unverified C code
- Support for *capabilities* (next slide)
  - CHERI-inspired
  - Linear capabilities

No assembly hassle in C, but still unsafe (powerful attacker).

(Linear) Capabilities

Capability:

- Unforgeable memory pointer
- Grants permissions on memory region
- Fine-grained memory protection
- Capability machines (ex CHERI)



Linear Capability:

- Linearity = one-use! cfr e.g. Linear Logic
- Non-copyable $\Rightarrow$ callers/callees cannot keep copies
- Intuitive: separation logic is linear

Relation to *full abstraction*

Full abstraction

$=$ reflection and preservation of contextual equivalence

$$s \simeq_{ctx} s' \Leftrightarrow [[s]] \simeq_{ctx} [[s']]$$
$$\text{where } x \simeq_{ctx} x' \equiv \forall C : C[x] \Downarrow \Leftrightarrow C[x'] \Downarrow$$

$\supseteq$ preservation of integrity and confidentiality

Relation to *full abstraction*

Full abstraction

= reflection and preservation of contextual equivalence

$$s \simeq_{ctx} s' \Leftrightarrow [[s]] \simeq_{ctx} [[s']]$$
$$\text{where } x \simeq_{ctx} x' \equiv \forall C : C[x] \Downarrow \Leftrightarrow C[x'] \Downarrow$$

$\supseteq$ preservation of integrity and confidentiality

Importance

Fully abstract compiler $\Rightarrow$ compiled code upholds contracts

Related work (Agten et al.)

- Different hardware primitives
  $\Rightarrow$ Less fine-grained
- Integrity, *not* confidentiality

Example program

Illustrates approach

Based on *separation logic derivation* (next slide)

```
void array_map(int n[], int *data
    , int L)
//@pre n ↦ [_]_L * data ↦ _;
//@post n ↦ [_]_L * data ↦ _;
{
  if (L == 0) {
    skip;
  } else {
    //@split n[0];
    int newVal = p(n[0], data);
    n[0] = newVal;
    array_map(n+1, data, L−1);
    //@join n (n+1);
  }
  return;
}
```

```
int p(int x, int *data)
//@pre data ↦ _;
//@post data ↦ _;
{...}
```

Elements

- $*$, $\mapsto$
- @pre/post: contract
- array chunk notation: $[\cdot]$
- @split/join: manipulate array chunks

Separation logic derivation

= tree-shaped proof of function contract

Root = the Hoare triple: $\{pre\}\ BODY\ \{post\}$

Used as input $\Rightarrow$ *separation-logic-proof-directed compilation*

Example program: else derivation

```
void array_map(int n[], int *data, int L)
//@pre n ↦ [_]_L * data ↦ _;
//@post n ↦ [_]_L * data ↦ _;
{
  //{c1: n ↦ [_]_L * c2: data ↦ _}
  if (L == 0) {
    (...)
  } else {
    //{c1: n ↦ [_]_L * c2: data ↦ _ * L != 0}
    //{c1: n ↦ [d,_]_L * c2: data ↦ _}
    //@split n[0];
    //{c1: n ↦ [d] * c3: n+1 ↦ [_]_L−1 * c2: data ↦ _}
    int newVal = p(n[0], data);
    //{c1: n ↦ [d] * c3: n+1 ↦ [_]_L−1 * c2: data ↦ _ *
   newVal = _}
    n[0] = newVal;
    //{c1: n ↦ [newVal] * c3: n+1 ↦ [_]_L−1 * c2: data ↦ _}
    (...)
```

Example program: else derivation

```
void array_map(int n[], int *data, int L)
//@pre n ↦ [_]_L * data ↦ _;
//@post n ↦ [_]_L * data ↦ _;
{
  //{c1: n ↦ [_]_L * c2: data ↦ _}
  if (L == 0) {
    (...)
  } else {
    //{c1: n ↦ [_]_L * c2: data ↦ _ * L != 0}
    //{c1: n ↦ [d,_]_L * c2: data ↦ _}
    //@split n[0];
    //{c1: n ↦ [d] * c3: n+1 ↦ [_]_{L-1} * c2: data ↦ _}
    int newVal = p(n[0], data);
    //{c1: n ↦ [d] * c3: n+1 ↦ [_]_{L-1} * c2: data ↦ _ *
   newVal = _}
    n[0] = newVal;
    //{c1: n ↦ [newVal] * c3: n+1 ↦ [_]_{L-1} * c2: data ↦ _}
    (...)
```

Example program: else derivation

```
void array_map(int n[], int *data, int L)
//@pre n ↦ [_]_L * data ↦ _;
//@post n ↦ [_]_L * data ↦ _;
{
  //{c1: n ↦ [_]_L * c2: data ↦ _}
  if (L == 0) {
    (...)
  } else {
    //{c1: n ↦ [_]_L * c2: data ↦ _ * L != 0}
    //{c1: n ↦ [d,_]_L * c2: data ↦ _}
    //@split n[0];
    //{c1: n ↦ [d] * c3: n+1 ↦ [_]_{L−1} * c2: data ↦ _}
    int newVal = p(n[0], data);
    //{c1: n ↦ [d] * c3: n+1 ↦ [_]_{L−1} * c2: data ↦ _ * newVal = _}
    n[0] = newVal;
    //{c1: n ↦ [newVal] * c3: n+1 ↦ [_]_{L−1} * c2: data ↦ _}
    (...)
```

Example program: else derivation

```
void array_map(int n[], int *data, int L)
//@pre n ↦ [_]_L * data ↦ _;
//@post n ↦ [_]_L * data ↦ _;
{
  //{c1: n ↦ [_]_L * c2: data ↦ _}
  if (L == 0) {
    (...)
  } else {
    //{c1: n ↦ [_]_L * c2: data ↦ _ * L != 0}
    //{c1: n ↦ [d,_]_L * c2: data ↦ _}
    //@split n[0];
    //{c1: n ↦ [d] * c3: n+1 ↦ [_]_{L-1} * c2: data ↦ _}
    int newVal = p(n[0], data);
    //{c1: n ↦ [d] * c3: n+1 ↦ [_]_{L-1} * c2: data ↦ _ * newVal = _}
    n[0] = newVal;
    //{c1: n ↦ [newVal] * c3: n+1 ↦ [_]_{L-1} * c2: data ↦ _}
    (...)
```

Example program: else derivation

```
void array_map(int n[], int *data, int L)
//@pre n ↦ [_]_L * data ↦ _;
//@post n ↦ [_]_L * data ↦ _;
{
  //{c1: n ↦ [_]_L * c2: data ↦ _}
  if (L == 0) {
    (...)
  } else {
    //{c1: n ↦ [_]_L * c2: data ↦ _ * L != 0}
    //{c1: n ↦ [d,_]_L * c2: data ↦ _}
    //@split n[0];
    //{c1: n ↦ [d] * c3: n+1 ↦ [_]_{L-1} * c2: data ↦ _}
    int newVal = p(n[0], data);
    //{c1: n ↦ [d] * c3: n+1 ↦ [_]_{L-1} * c2: data ↦ _ * newVal = _}
    n[0] = newVal;
    //{c1: n ↦ [newVal] * c3: n+1 ↦ [_]_{L-1} * c2: data ↦ _}
    (...)
```

Example program: else derivation

```
void array_map(int n[], int *data, int L)
//@pre n ↦ [_]_L * data ↦ _;
//@post n ↦ [_]_L * data ↦ _;
{
  //{c1: n ↦ [_]_L * c2: data ↦ _}
  if (L == 0) {
    (...)
  } else {
    //{c1: n ↦ [_]_L * c2: data ↦ _ * L != 0}
    //{c1: n ↦ [d,_]_L * c2: data ↦ _}
    //@split n[0];
    //{c1: n ↦ [d] * c3: n+1 ↦ [_]_L−1 * c2: data ↦ _}
    int newVal = p(n[0], data);
    //{c1: n ↦ [d] * c3: n+1 ↦ [_]_L−1 * c2: data ↦ _ * newVal = _}
    n[0] = newVal;
    //{c1: n ↦ [newVal] * c3: n+1 ↦ [_]_L−1 * c2: data ↦ _}
    (...)
```

Example program: else derivation

```
void array_map(int n[], int *data, int L)
//@pre n ↦ [_]_L ∗ data ↦ _;
//@post n ↦ [_]_L ∗ data ↦ _;
{
  //{c1: n ↦ [_]_L ∗ c2: data ↦ _}
  if (L == 0) {
    (...)
  } else {
    //{c1: n ↦ [_]_L ∗ c2: data ↦ _ ∗ L != 0}
    //{c1: n ↦ [d,_]_L ∗ c2: data ↦ _}
    //@split n[0];
    //{c1: n ↦ [d] ∗ c3: n+1 ↦ [_]_{L−1} ∗ c2: data ↦ _}
    int newVal = p(n[0], data);
    //{c1: n ↦ [d] ∗ c3: n+1 ↦ [_]_{L−1} ∗ c2: data ↦ _ ∗ newVal = _}
    n[0] = newVal;
    //{c1: n ↦ [newVal] ∗ c3: n+1 ↦ [_]_{L−1} ∗ c2: data ↦ _}
    (...)
```

Example program: else derivation

```
void array_map(int n[], int *data, int L)
//@pre n ↦ [_]_L * data ↦ _;
//@post n ↦ [_]_L * data ↦ _;
{
    (...)
    //{c1: n ↦ [newVal] * c3: n+1 ↦ [_]_{L−1} * c2: data ↦ _}
    array_map(n+1, data, L−1);
    //{c1: n ↦ [newVal] * c3: n+1 ↦ [_]_{L−1} * c2: data ↦ _}
    //@join n (n+1);
    //{c1: n ↦ [newVal,_]_L * c2: data ↦ _}
    //{c1: n ↦ [_]_L * c2: data ↦ _}
  }
  return;
  //{c1: n ↦ [_]_L * c2: data ↦ _ * result == ()}
  //{c1: n ↦ [_]_L * c2: data ↦ _ }
}
```

Example program: else derivation

```
void array_map(int n[], int *data, int L)
//@pre n ↦ [_]_L * data ↦ _;
//@post n ↦ [_]_L * data ↦ _;
{
    (...)
    //{c1: n ↦ [newVal] * c3: n+1 ↦ [_]_{L−1} * c2: data ↦ _}
    array_map(n+1, data, L−1);
    //{c1: n ↦ [newVal] * c3: n+1 ↦ [_]_{L−1} * c2: data ↦ _}
    //@join n (n+1);
    //{c1: n ↦ [newVal,_]_L * c2: data ↦ _}
    //{c1: n ↦ [_]_L * c2: data ↦ _}
  }
  return;
  //{c1: n ↦ [_]_L * c2: data ↦ _ * result == ()}
  //{c1: n ↦ [_]_L * c2: data ↦ _ }
}
```

Example program: else derivation

```
void array_map(int n[], int *data, int L)
//@pre n ↦ [_]_L * data ↦ _;
//@post n ↦ [_]_L * data ↦ _;
{
    (...)
    //{c1: n ↦ [newVal] * c3: n+1 ↦ [_]_{L−1} * c2: data ↦ _}
    array_map(n+1, data, L−1);
    //{c1: n ↦ [newVal] * c3: n+1 ↦ [_]_{L−1} * c2: data ↦ _}
    //@join n (n+1);
    //{c1: n ↦ [newVal,_]_L * c2: data ↦ _}
    //{c1: n ↦ [_]_L * c2: data ↦ _}
  }
  return;
  //{c1: n ↦ [_]_L * c2: data ↦ _ * result == ()}
  //{c1: n ↦ [_]_L * c2: data ↦ _ }
}
```

Example program: else derivation

```
void array_map(int n[], int *data, int L)
//@pre n ↦ [_]_L * data ↦ _;
//@post n ↦ [_]_L * data ↦ _;
{
    (...)
    //{c1: n ↦ [newVal] * c3: n+1 ↦ [_]_{L−1} * c2: data ↦ _}
    array_map(n+1, data, L−1);
    //{c1: n ↦ [newVal] * c3: n+1 ↦ [_]_{L−1} * c2: data ↦ _}
    //@join n (n+1);
    //{c1: n ↦ [newVal,_]_L * c2: data ↦ _}
    //{c1: n ↦ [_]_L * c2: data ↦ _}
  }
  return;
  //{c1: n ↦ [_]_L * c2: data ↦ _ * result == ()}
  //{c1: n ↦ [_]_L * c2: data ↦ _ }
}
```

Example program: else derivation

```
void array_map(int n[], int *data, int L)
//@pre n ↦ [_]_L * data ↦ _;
//@post n ↦ [_]_L * data ↦ _;
{
    (...)
    //{c1: n ↦ [newVal] * c3: n+1 ↦ [_]_{L−1} * c2: data ↦ _}

    array_map(n+1, data, L−1);
    //{c1: n ↦ [newVal] * c3: n+1 ↦ [_]_{L−1} * c2: data ↦ _}

    //@join n (n+1);
    //{c1: n ↦ [newVal,_]_L * c2: data ↦ _}
    //{c1: n ↦ [_]_L * c2: data ↦ _}
  }
  return;
  //{c1: n ↦ [_]_L * c2: data ↦ _ * result == ()}
  //{c1: n ↦ [_]_L * c2: data ↦ _ }
}
```

Example program: else derivation

```
void array_map(int n[], int *data, int L)
//@pre n ↦ [_]_L * data ↦ _;
//@post n ↦ [_]_L * data ↦ _;
{
    (...)
    //{c1: n ↦ [newVal] * c3: n+1 ↦ [_]_{L−1} * c2: data ↦ _}

    array_map(n+1, data, L−1);
    //{c1: n ↦ [newVal] * c3: n+1 ↦ [_]_{L−1} * c2: data ↦ _}

    //@join n (n+1);
    //{c1: n ↦ [newVal,_]_L * c2: data ↦ _}
    //{c1: n ↦ [_]_L * c2: data ↦ _}
  }
  return;
  //{c1: n ↦ [_]_L * c2: data ↦ _ * result == ()}
  //{c1: n ↦ [_]_L * c2: data ↦ _ }
}
```

Example program: else derivation

```
void array_map(int n[], int *data, int L)
//@pre n ↦ [_]_L * data ↦ _;
//@post n ↦ [_]_L * data ↦ _;
{
    (...)
    //{c1: n ↦ [newVal] * c3: n+1 ↦ [_]_L−1 * c2: data ↦ _}

    array_map(n+1, data, L−1);
    //{c1: n ↦ [newVal] * c3: n+1 ↦ [_]_L−1 * c2: data ↦ _}

    //@join n (n+1);
    //{c1: n ↦ [newVal,_]_L * c2: data ↦ _}
    //{c1: n ↦ [_]_L * c2: data ↦ _}
  }
  return;
  //{c1: n ↦ [_]_L * c2: data ↦ _ * result == ()}
  //{c1: n ↦ [_]_L * c2: data ↦ _ }
}
```

Translation: Intuition

Separation-logic-proof-directed

- Chunks become linear capabilities
  - Contain all permissions
  - This is why we **name** heap chunks!
- Original pointers become addresses
  - Regular ints
  - Lose all permission
  - Kept for address operations

```
//{c1: n ↦ [_]_L}
n: int*
```

```
c1: int* (linear)
 n: int
```

Translation: Split/Join

Performed by target language built-in functions *split* and *join*.
Split creates 2 linear array capabilities out of one. Join does the opposite.

```
//{c1: n ↦ [_]_L * c2: data ↦ _ * L != 0}
//{c1: n ↦ [d,_]_L * c2: data ↦ _}
//@split n[0];
//{c1: n ↦ [d] * c3: n+1 ↦ [_]_{L−1} * c2: data ↦ _}
int newVal = p(n[0], data);
```

```
{c1,c3} = split(c1,0);
```

Translation: Array operations

Source language array mutation/lookup
⇒ Target language mutation/lookup on the corresponding capability

```
int newVal = p(n[0], data);
//{c1: n ↦ [d] * c3: n+1 ↦ [_]L−1 * c2: data ↦ _ *
    newVal = _}

n[0] = newVal;
//{c1: n ↦ [newVal] * c3: n+1 ↦ [_]L−1 * c2: data ↦ _}

array_map(n+1, data);
```

```
c1[0] = newVal;
```

Translation: Function call

Add arguments/return values to calls.
Corresponding to heap chunks.

```
int p(int x,int *data)
//@pre data ↦ _;
//@post data ↦ _;
{...}
```

```
//@split n[0];
//{c1: n ↦ [d] * c3: n+1 ↦ [_]L−1 * c2: data ↦ _}
int newVal = p(n[0], data);
//{c1: n ↦ [d] * c3: n+1 ↦ [_]L−1 * c2: data ↦ _ *
    newVal = _}

n[0] = newVal;
```

```
{int,int*} p(int x, int data, int* data_cap){...}
           {newVal,c2} = p(c1[0],data,c2);
```

Translation: In-/Outcall

- In/outcall stub:
  transitions to/from verified module
- Functions:
  - Check linearity of capabilities
  - Check any conditions in the contract

```
int p(int x,int *data)
//@pre data ↦ _;
//@post data ↦ _;
{...}
```

```
{int,int*} p_stub(int x, int data, int* data_cap){
  intptr_t cap_address = (intptr_t) data_cap;

  {int result, int* data_cap} = p(x,data,data_cap);

  assert(cap_address == (intptr_t) data_cap);
  assert(is_linear(data_cap));

  return {result, data_cap};
}
```

Proof: Outline

Full abstraction

$$\vdash s \rightsquigarrow t \land \vdash s' \rightsquigarrow t' \Rightarrow (s \simeq_{ctx} s' \Leftrightarrow t \simeq_{ctx} t')$$

- $\vdash s$: specific proof of s
- $\rightsquigarrow$: compiles to
- $x \simeq_{ctx} x' \equiv \forall C : C[x] \Downarrow \Leftrightarrow C[x'] \Downarrow$
  Requires operational semantics!

Proof: Outline

Full abstraction

$$\vdash s \rightsquigarrow t \wedge \vdash s' \rightsquigarrow t' \Rightarrow (s \simeq_{ctx} s' \Leftrightarrow t \simeq_{ctx} t')$$

Correctness $\Leftarrow$

Reflection of $\simeq_{ctx}$

Security $\Rightarrow$

Preservation of $\simeq_{ctx}$

Proof: Correctness $\Leftarrow$

Approach

$$\frac{t \simeq_{ctx} t' \qquad \vdash s \rightsquigarrow t \qquad \vdash s' \rightsquigarrow t'}{s \simeq_{ctx} s'}$$

Techniques: compilation $[[\cdot]]$ + simulation relation R

Proof: Correctness

$$\frac{t \simeq_{ctx} t' \qquad \vdash s \rightsquigarrow t \qquad \vdash s' \rightsquigarrow t'}{s \simeq_{ctx} s'}$$

$$\begin{array}{lcccc}
 & & s \simeq^{?}_{ctx} s' & & \\
 & & \Updownarrow & \text{definition } \simeq_{ctx} & \\
\forall C. & C[s] \Downarrow & \Rightarrow^{?} & & C[s'] \Downarrow \\
 & \Downarrow \textit{coherence} & & & \Uparrow \textit{coherence} \\
 & \vdash C[s] \Downarrow & & & \vdash C[s'] \Downarrow \\
 & \Downarrow * & & & \Uparrow * \\
 & [[C]][t] \Downarrow & \Rightarrow & & [[C]][t'] \Downarrow \\
 & & \Updownarrow & \text{definition } \simeq_{ctx} & \\
 & & t \simeq_{ctx} t' & &
\end{array}$$

$*$ **Lemma's:** $\vdash C[s] \; R \; [[C]][[[s]]], \; \vdash x \; R \; y \Rightarrow \vdash x \Downarrow \Leftrightarrow y \Downarrow$

Proof: Security $\Rightarrow$

Approach

$$\frac{s \simeq_{ctx} s' \qquad \vdash s \rightsquigarrow t \qquad \vdash s' \rightsquigarrow t'}{t \simeq_{ctx} t'}$$

Techniques: back-translation $\langle\langle \cdot \rangle\rangle$ + simulation relation R

Proof: Security

$$\frac{s \simeq_{ctx} s' \qquad \vdash s \rightsquigarrow t \qquad \vdash s' \rightsquigarrow t'}{t \simeq_{ctx} t'}$$

$$\begin{array}{ccc}
 & s \simeq_{ctx} s' & \\
 & \Updownarrow \quad \text{definition } \simeq_{ctx} & \\
\langle\langle C\rangle\rangle[s] \Downarrow & \Rightarrow & \langle\langle C\rangle\rangle[s'] \Downarrow \\
\Uparrow \textit{coherence} & & \Downarrow \textit{coherence} \\
\vdash \langle\langle C\rangle\rangle[s] \Downarrow & & \vdash \langle\langle C\rangle\rangle[s'] \Downarrow \\
\Uparrow * & & \Downarrow * \\
\forall C. \quad C[t] \Downarrow & \Rightarrow^{?} & \vdash C[t'] \Downarrow \\
 & \Updownarrow \quad \text{definition } \simeq_{ctx} & \\
 & t \simeq^{?}_{ctx} t' &
\end{array}$$

$*$ **Lemma's:** $(\vdash \langle\langle C\rangle\rangle[s])\ R\ C[[[s]]]$, $\vdash x\ R\ y \Rightarrow \vdash x \Downarrow \Leftrightarrow y \Downarrow$

Proof: Security - back-translation example

Intuition:

- Construct minimal contract for context functions
- Insert assertions where necessary
- **Goal**: prove that $\vdash \langle\langle C \rangle\rangle[s] \Downarrow \Leftrightarrow C[t] \Downarrow$

Example:

Target

```
int f(int* a, int b){
  free(a);
  b = 5;
  return b;
}
```

$\langle\langle \cdot \rangle\rangle$ ⇒

Source

```
int f(int* a, int b)
//@pre (a_chunk: a ↦ [_]_L) ∨ a = null;
//@post true;
{
  (...)
}
```

Proof: Security - back-translation example

Intuition:

- Construct minimal contract for context functions
- Insert assertions where necessary

Example:

Target

```
int f(int* a, int b){
  free(a);
  (...)
}
```

$\langle\langle\cdot\rangle\rangle$ ➡

Source

```
int f(int* a, int b)
//@pre (a_chunk: a ↦ [_]_L) ∨ a = null;
//@post true;
{
  //{(a_chunk: a ↦ [_]_L) ∨ a = null}
  assert(a != null);
  //{a_chunk: a ↦ [_]_L}
  free(a);
  //{}
  (...)
}
```

Proof: Security - back-translation example

Intuition:

- Construct minimal contract for context functions
- Insert assertions where necessary

Example:

Target

```
int f(int* a, int b){
  (...)
  b = 5;
  return b;
}
```

$\langle\langle\cdot\rangle\rangle$

Source

```
int f(int* a, int b)
//@pre (a_chunk: a ↦ [_]_L) ∨ a = null;
//@post true;
{
  (...)
  //{}
  b = 5;
  //{b = 5}
  return b;
  //{result = 5 * b = 5}
  //{}
}
```

Conclusion and future work

- Compiler from verified C to unverified C with (linear) capabilities
- **Claim**: Full Abstraction

  $\Rightarrow$ Gave some proof intuition
- **State**:

  Correctness: $\sim$ proven

  Security: currently constructing back-translation