Linear capabilities for fully abstract compilation of separation-logic-verified code

Thomas Van Strydonck Dominique Devriese Frank Piessens

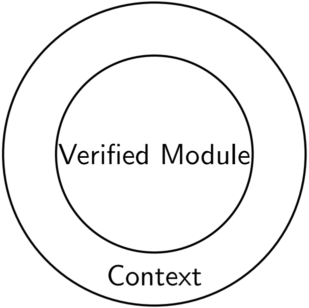
KU Leuven

thomas.vanstrydonck@cs.kuleuven.be

June 24, 2018

Problem: 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).

Overview

1. Problem context
2. Compilation by example
3. Conclusion and future work
4. ADS progress report

Outline

Linear capabilities for
fully abstract compilation of
separation-logic-verified code

Separation Logic

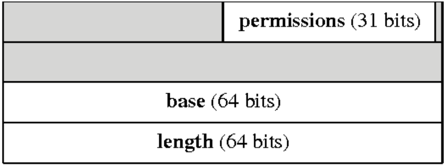
- Linear-like logic, contract based (pre-post), sound (proof implies correct), modular (per-module proofs), chunks, notation (grab from later)

Linear capabilities for fully abstract compilation of separation-logic-verified code

(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

Linear capabilities for
fully abstract compilation of
separation-logic-verified code

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

Recap

Preserving sound, modular verification + The compiler: now comes the example

Outline

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

$=$ proof of function contract

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 ↦ [_]ι}
n: int*
```

⬇

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

Translation: Split/Join

Target language built-in functions *split* and *join*.
Source language split/join
⇒ Target language split/join on corresponding capabilities

```
//{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);
```

Outline

Conclusion and future work

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

  Correctness and security: $\sim$ proven

    Technical report is lacking some details

  Currently writing paper for POPL

Outline

1. Problem context
2. Compilation by example
3. Conclusion and future work
4. ADS progress report

ADS progress

- Publications:
  - Thomas Van Strydonck, Dominique Devriese, Frank Piessens. Linear capabilities for modular fully-abstract compilation of verified code. Peer reviewed extended abstract. Talk at *PRISC 2018* (Principles of Secure Compilation; a workshop at the ACM-supported *POPL* conference), January 2018.
  - POPL 2019 submission currently being written (due 12 July 2018): Thomas Van Strydonck, Dominique Devriese, Frank Piessens. Linear capabilities for fully abstract compilation of separation-logic-verified code.
- The formal course units to be followed during the doctoral training programme:
  - OPLSS summer school (4sp)
  - Intensive academic writing course (SET) (2sp)

ADS progress

- Current contribution to bachelor and master education (work decided on yearly basis)
  - Submitted thesis proposal for 2018-2019 (*Developing attestation for capability machines*)
  - Taught *Modelling of Complex Systems* exercise sessions
  - Supervised *Probleemoplossen en ontwerpen, deel 3* exercise sessions and evaluated students
  - Supererervised a *Software-ontwerp* project group and aided in oral examination
  - Corrected exams for *Methodiek van de informatica*

ADS progress

- A detailed research plan of the doctoral project:
  - WP1: Fully-abstract compilation of verified C (Today's presentation)
  - WP2: Extending WP1 towards more realistic settings (Compiling non-annotated code, cooperating with non-compliant code)
  - WP3: Fully-abstract compilation of Rust