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

July 20, 2018

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

Overview

1. Full abstraction

2. Full abstraction for verified code

3. Compilation by example

4. Conclusion and future work

Outline

1. Full abstraction
2. Full abstraction for verified code
3. Compilation by example
4. Conclusion and future work

Full abstraction (FA)

Intuition

Attacking the compiled code is as hard as attacking the source code

Full abstraction (FA)

Intuition

Attacking the compiled code is as hard as attacking the source code

Definition of FA

$=$ reflection and preservation of contextual equivalence $\simeq_{ctx}$

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

$\supseteq$ preservation of integrity and confidentiality properties

Full abstraction (FA)

Intuition

Attacking the compiled code is as hard as attacking the source code

Definition of FA

$=$ reflection and preservation of contextual equivalence $\simeq_{ctx}$

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

$\supseteq$ preservation of integrity and confidentiality properties

Methodology: Change the source code by $[\![\cdot]\!]$ + prove FA

$\Rightarrow$ Change does not alter security aspects

eg. CFI, notion of private fields, ...

Outline

1. Full abstraction

2. Full abstraction for verified code

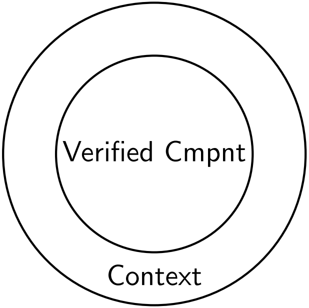
3. Compilation by example

4. Conclusion and future work

Problem: Preserving verification during compilation

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

  FA is exactly what we need!



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

Separation Logic

- Substructural logic (linear aspects)
- Program verification
  - Sound
  - Modular
- Contract-based, eg. :

```
void p(int x,int *data)
//@pre data ↦ _ * x > 0;
//@post data ↦ x;
{*data = x}
```

Notation:

- $*$, $\mapsto$ (resource: permission)
- @pre/post: contract
  Consume/produce
- Array resource notation:
  $\mapsto[a_1, \ldots, a_n]$

- Hoare-logic-style program proofs: $\Rightarrow \{P\}\ c\ \{Q\}$
  Functions: $\{@pre\}\ BODY\ \{@post\}$

Motivating example

	Verified Component	Context Declaration

Source	`void f(int* a)`
	`//@pre n: a ↦ [0]`
	`//@post n : a ↦ [1]`
	`{`
	`   id(a);`
	`   a[0]=1;`
	`}`
		`void id(int* a)`
		`//@pre n: a ↦ [0]`
		`//@post n : a ↦ [0]`

- SOA: compilers erase contracts
- Untrusted function *id* can:
  - Overread/-write using *a*, copy *a*
  - Not satisfy postcondition (eg. $a[0] = 2$)
- **NOT fully abstract!**

The compiler

Source language

- Regular verified C code
- *Separation logic* annotated

Target language

- Regular unverified C code
- **Language enhancement to allow full abstraction???**

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

What (language) enhancements do we need?

	Verified Component	Context Declaration

Source	`void f(int* a)`
	`//@pre n: a ↦ [0]`
	`//@post n : a ↦ [1]`
	`{`
	`  id(a);`
	`  a[0]=1;`
	`}`
		`void id(int* a)`
		`//@pre n: a ↦ [0]`
		`//@post n : a ↦ [0]`

Recall, *id* can:

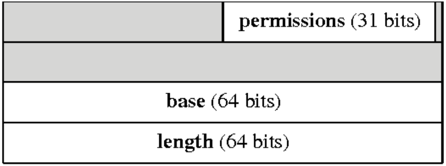
- Overread/-write using a, copy a
  - $\Rightarrow$ **Capabilities** implement POLA
  - $\Rightarrow$ **Linear Capabilities** prevent copying
- Not satisfy postcondition (eg. $a[0] = 2$)
  - $\Rightarrow$ **Checking functions aka. stubs**

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 $\Rightarrow$ callees cannot keep copies
- Intuitive: separation logic is linear

Goal: proving that contracts are compiled away safely by proving full abstraction

Source language

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

FA!

Target language

- Regular unverified C code
- Support for *capabilities*
  - CHERI-inspired
  - Linear capabilities

Related work (Agten et al.)

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

Outline

Compilation: Intuition

- Resources reified into linear capabilities
  - Behave linearly
  - Contain all permissions
  - This is why we **name** heap resources!
- Original pointers become addresses
  - Regular ints[1]
  - Lose all permission
  - Kept for address operations

```
//{c1: n ↦ [1,2,3]}
n: int*
```

⬇

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

$\Rightarrow$ *Separation-logic-proof-directed*:

- **proof of** input program used as input
- compiled operations on resources, eg. $a[0] \Rightarrow_{[\![\cdot]\!]} n[0]$

[1] This is a slight simplification

Motivating example: overread/overwrite/copy

	Verified Component	Context Declaration

Source	`void f(int* a)`
	`//@pre n: a ↦ [0]`
	`//@post n : a ↦ [1]`
	`{`
	`  id(a);`
	`  a[0]=1;`
	`}`
		`void id(int* a)`
		`//@pre n: a ↦ [0]`
		`//@post n : a ↦ [0]`

Target	`int* f(int a,int* n)`
	`{`
	`  n = id(a,n);`
	`  n[0]=1;`
	`  return n;`
	`}`
		`int* id(int a,int* n)`

Prevented by: Linear capabilities + Proof-directed-compilation

Motivating example: postcondition

	Stub	Context Declaration

Source		`void id(int* a)`
		`//@pre n: a ↦ [0]`
		`//@post n : a ↦ [0]`

Target	`int* id_stub(int a,int* n)`
	`{`
	`  n = id(a,n);`
	`  assert(a == addr(n));`
	`  assert(len(n) == 1);`
	`  assert(n[0] == 0);`
	`  return n;`
	`}`
		`int* id(int a,int* n)`

Prevented by: stub

Outline

Conclusion

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

Fully abstractly compiling Rust (IDEA STAGE)

- Ownership and borrowing; linear aspects
  $\Rightarrow$ Compile borrows to linear capabilities
- Start from λ_{Rust} in RustBelt