Sound modular verification of code running in an untrusted binary code context

Frank Piessens, KU Leuven

WORKSHOP ON SECURITY FOR LOW-LEVEL CODE
SEPTEMBER 8, 2015, SAINT-MALO, FRANCE

DistriNet
iMinds KU LEUVEN

Introduction

- Great progress in sound **modular** verification of source code
  - … but (except for some rare cases) whole-system verification is not yet reachable
  - As a consequence, modularly verified code needs to run side-by-side with unverified (= possibly buggy/malicious) code at run time.
- Objective:
  - Maintaining soundness of modular verification after compilation

Our focus for this talk:
- C-like language
- Security properties expressible in separation logic
- Attacker model = attacker can compromise the **machine code** of the non-verified modules of the system

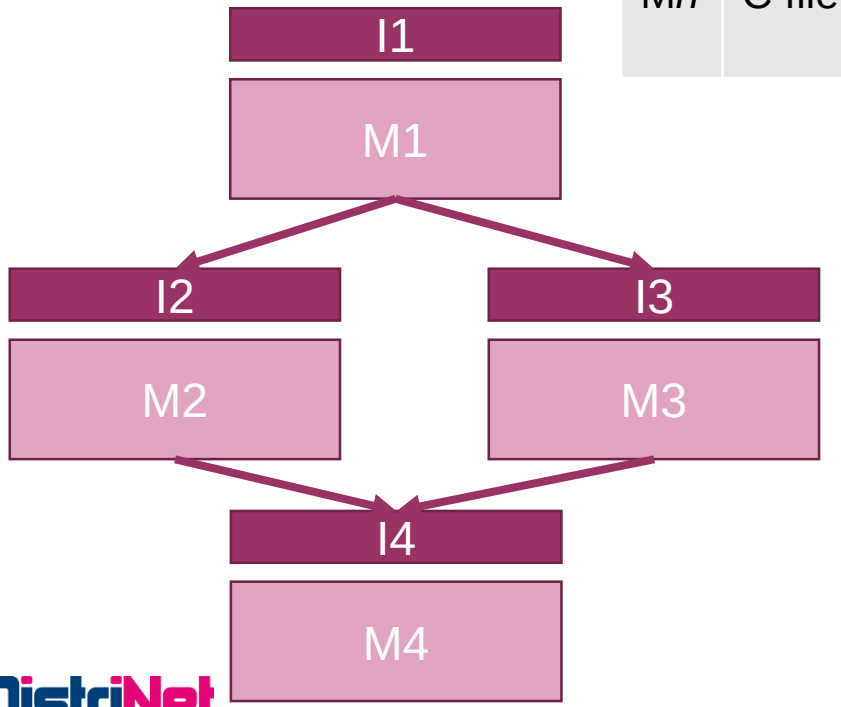
Structure of the talk

- Overview
- Low-level platform protection mechanisms
- Secure compilation of mini-C
- Handling C-style dynamic memory allocation
- Implementation
- Conclusions

Overview

Consider a program consisting of a number of modules, and their dependencies.

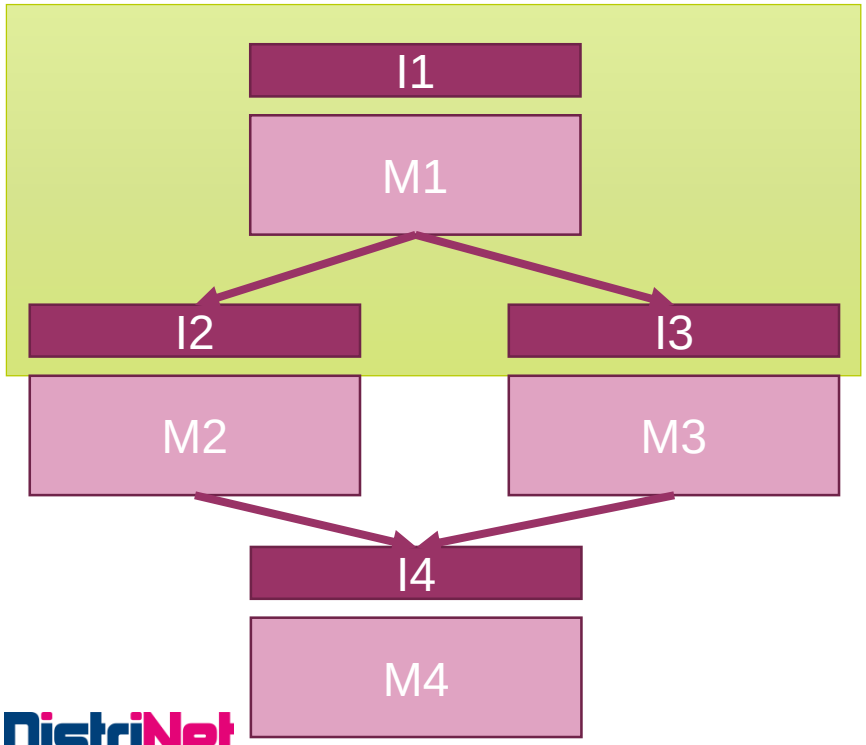
	C	Java	ML
I*n*	Header file	(Roughly) Interfaces	Signature
M*n*	C file	(Roughly) Classes	Structure / Functor



Overview

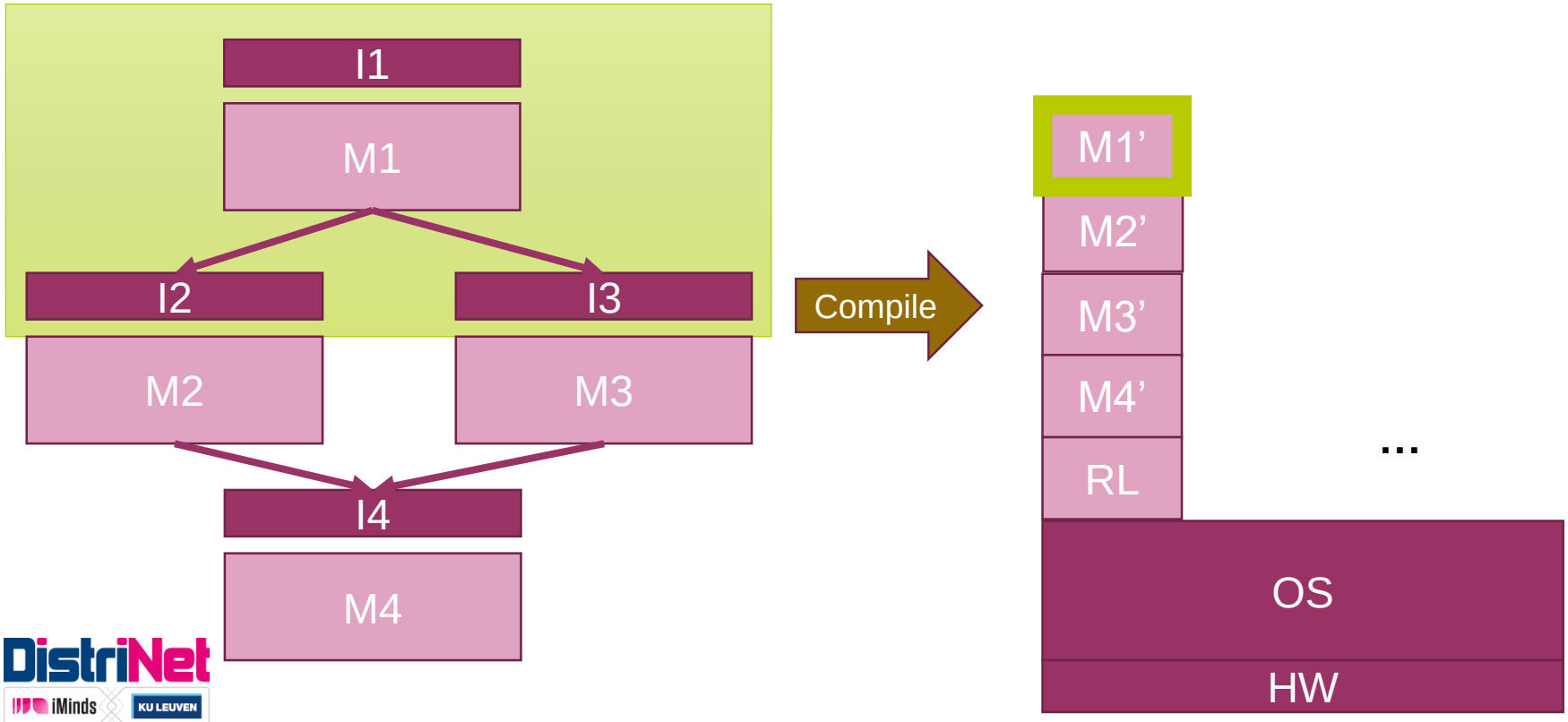
Suppose you have proven a (security) property of module M1 by modular reasoning. E.g.:

- Some invariant holds on the module's state
- The integrity of some data in the module is protected from other modules
- (Some data in the module remains confidential towards other modules)



Overview

M1 is compiled to a machine code module M1' running in a process on top of an OS/HW platform.
Other compiled modules and a runtime library (RL) run in the same process and share memory with M1'.
How can we secure interactions between M1' and its context such that verified properties can not be invalidated?



Structure of the talk

- Overview
- Low-level platform protection mechanisms
- Secure compilation of mini-C
- Handling C-style dynamic memory allocation
- Implementation
- Conclusions

Low-level protection

- Typed assembly language
  - Morrisett et al. *From System F to typed assembly language*, ACM TOPLAS (1999)
- Hardware supported low-level security monitors
  - Intel SGX
  - Sancus machine
    - Noorman et al. , *Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base*, Usenix Security 2013
  - PUMP machine
    - Dhawan et al. *Architectural Support for Software-Defined Metadata Processing*, ASPLOS 2015

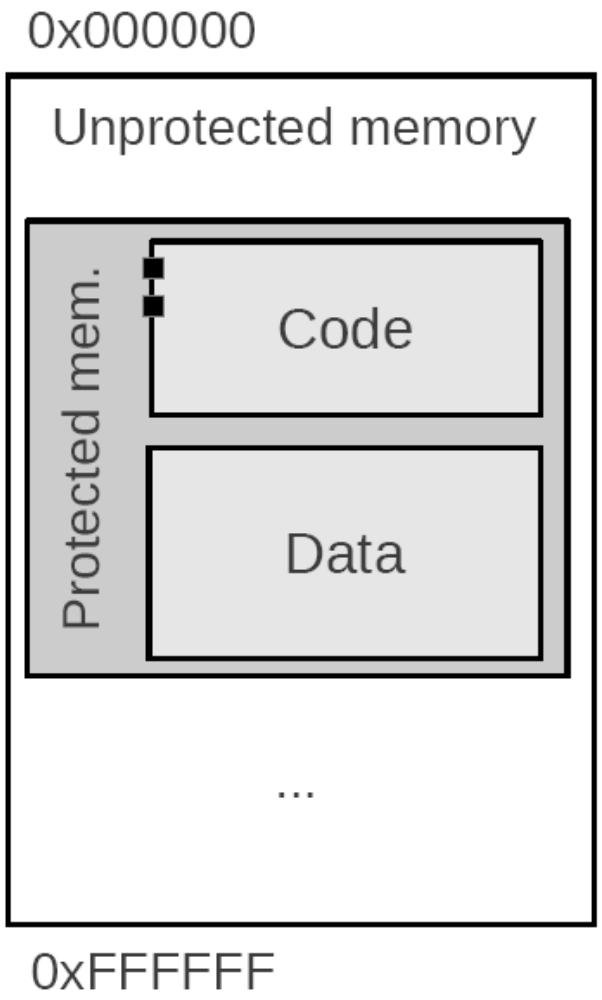
A simplified SGX model

- Standard Intel x86 style platform
  - Processor with
    - Program Counter
    - Registers and a Stack Pointer
    - Status (flags) registers
  - 32-bit memory space mapping 32-bit addresses to 32-bit words
- Extended with a program-counter based memory access control model
  - “SGX enclaves” or “protected modules”
- (Note that SGX has many more features that we do not model)

Low-level protection mechanism

- Need some low-level protection mechanism
- Program counter-based memory access control

from \ to	*Protected*			*Unprotected*
	Entry point	*Code*	*Data*	
Protected	r x	r x	r w	r w x
Unprotected	x			r w x



Structure of the talk

- Overview
- Low-level platform protection mechanisms
- Secure compilation of mini-C
- Handling C-style dynamic memory allocation
- Implementation
- Conclusions

Preserving validity of assertions

- Consider a sequential subset of C without dynamic memory allocation

M1.c

```
static int value = 0;

int get() {
  return value;
}

void inc() {
  int oldval = value;
  value += 1;
  int newval = value;
  observer();
  assert(newval == oldval + 1);
  return;
}
```

M2.c

```
void observer() {

// code of observer omitted

}
```

Preserving validity of assertions

- Consider a sequential subset of C without dynamic memory allocation

M1.c

```
static int value = 0;

int get() {
  return value;
}

void inc() {
  int oldval = value;
  value += 1;
  int newval = value;
  observer();
  assert(newval == oldval + 1);
  return;
}
```

M2.c

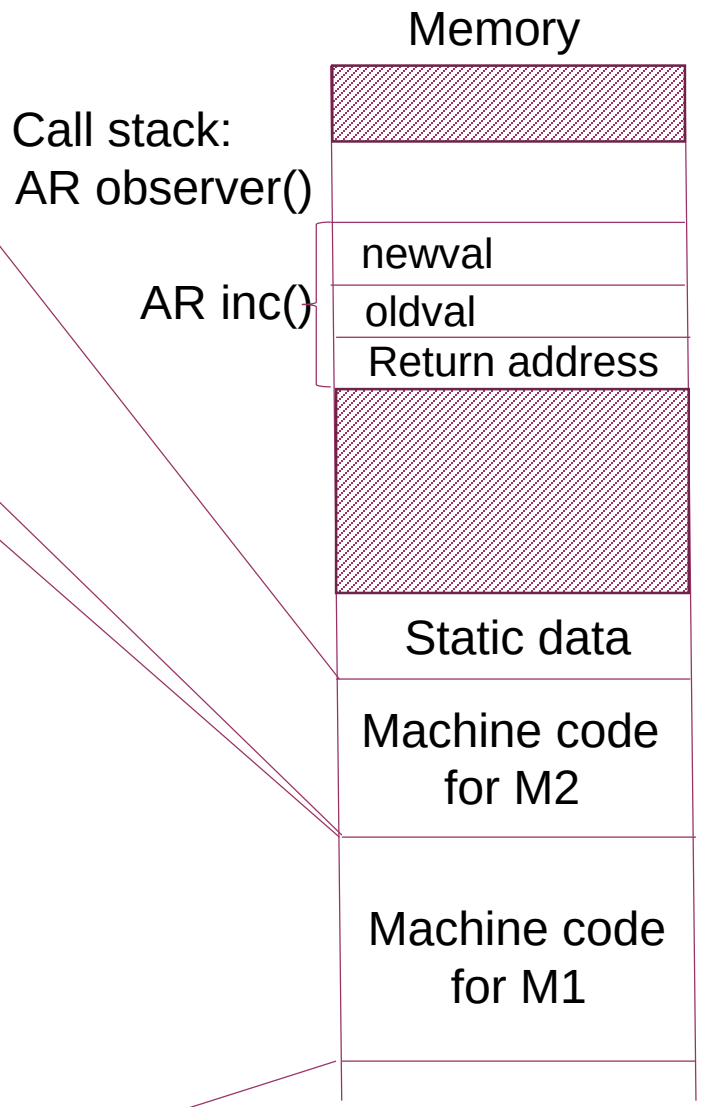
```
void observer() {

// code of observer omitted

}
```

This assertion is valid according to the source code semantics …
But fails if an attacker can mess with the machine code of M2.c after compilation.

Standard compilation

```
void observer() {

// code of observer omitted

}
```

```
static int value = 0;

int get() {
  return value;
}

void inc() {
  int oldval = value;
  value += 1;
  int newval = value;
  observer();
  assert(newval == oldval + 1);
  return;
}
```



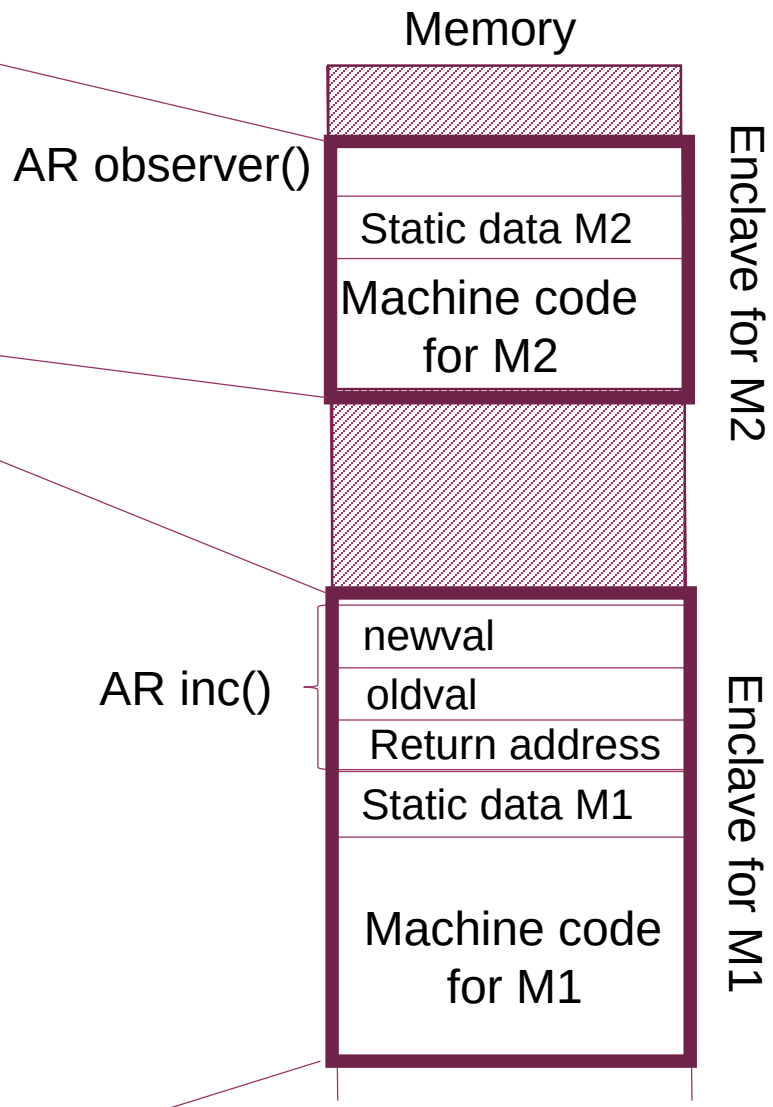
Compilation to simplified SGX

- C modules are compiled to SGX enclaves
  - Space for static (private) vars in the data section
  - Machine code for all functions in the text section
  - Entry points for each publicly accessible function
- Calling conventions and call stack:
  - Pass parameters through processor registers
  - Call stack: activation record of a function call stored in the data section of the enclave containing the function
  - A specific return entry point supports returning from a callback

Secure compilation

```
void observer() {

// code of observer omitted

}
```

```
static int value = 0;

int get() {
  return value;
}

void inc() {
  int oldval = value;
  value += 1;
  int newval = value;
  observer();
  assert(newval == oldval + 1);
  return;
}
```



DistriNet
iMinds
KU LEUVEN

Secure compilation

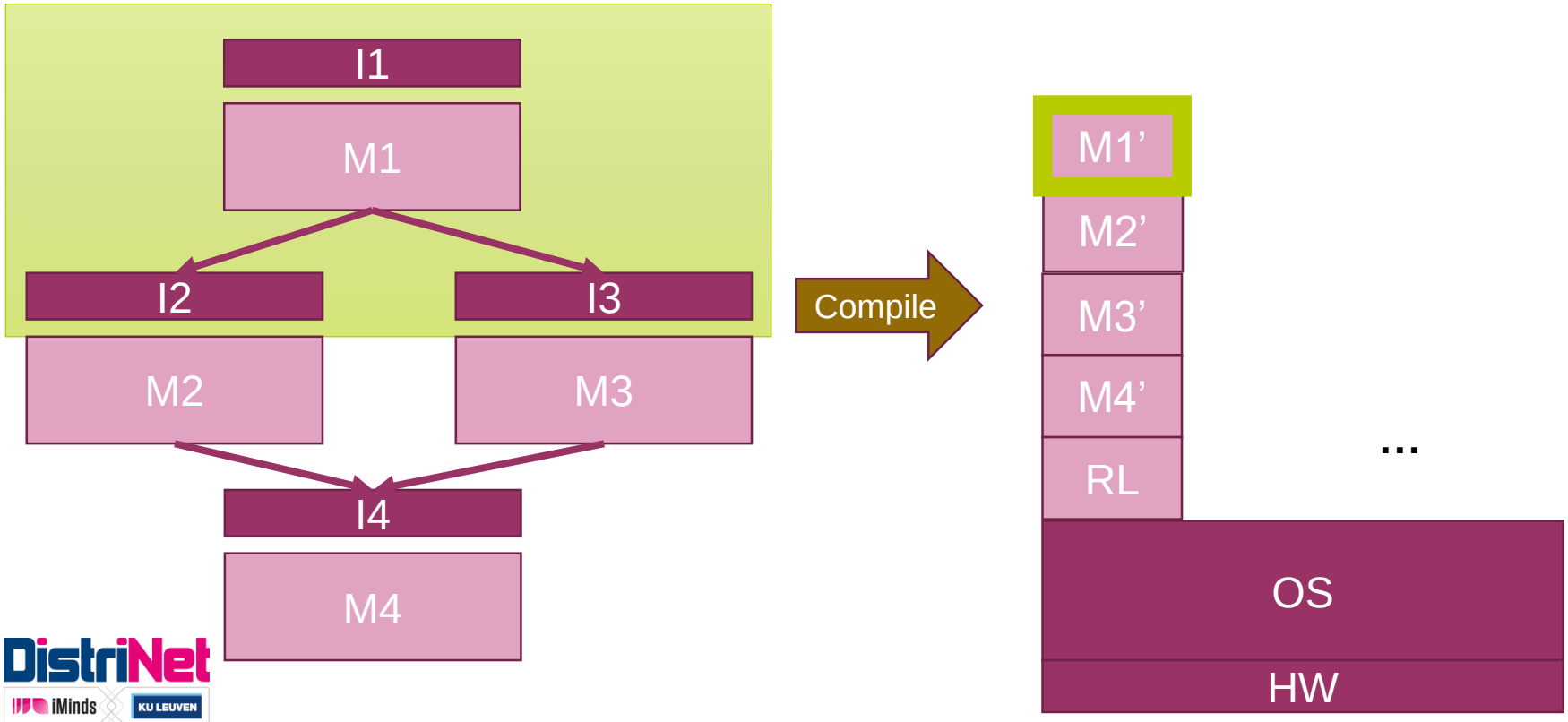
- Many details to get right
  - Depending on the source language features to support
    - Function pointers, objects, classes, exceptions, …
- See the following papers:
  - Agten et al. *Secure compilation to modern processors*, CSF 2012
  - Patrignani et al. *Secure compilation to protected module architectures*, TOPLAS 2015

    for formal proofs that this style of compilation can be made **fully abstract**
    - i.e. machine code can only interact with a module as source code can

What about interface specs?

We now know how to compile M1 to a hardened M1' such that machine code contexts can only do what source code contexts can do.
But verification of properties of M1 might rely on specs of I2 and I3!

HENCE: we need to insert run time checks for these contracts.



Checking interface contracts

- We will implement interface contract checking as a source-to-source program transformation
- The resulting “hardened” module can be verified in any source code context
  - i.e. with empty interface contracts
- And hence will maintain any verified property after compilation to machine code with the secure (fully abstract) compiler just discussed

Checking interface contracts

```
// Original module
int fac(int x)
 req x >= 0;
 ens res == fact(x);
{
  if (x == 0) return 1;
  int p = prod(x,
    fac(x-1));
  return p;
}

int prod(int x, int y);
 req true;
 ens res == x * y;
```

Program transformation

```
// Hardened module
// (Functional part)
static int _fac(int x) {
  if (x == 0) return 1;
  int p = _prod(x,
    _fac(x-1));
  return p;
}

// (Boundary part)
static
int _prod(int x, int y){
  int r = prod(x, y);
  if (! (r == x * y))
    trap();
  return r;
}

int fac(int x) {
  if (! (x >= 0)) trap();
  return _fac(x);
}
```

Checking interface contracts

```
// Original module
int fac(int x)
 req x >= 0;
 ens res == fact(x);
{
  if (x == 0) return 1;
  int p = prod(x,
    fac(x-1));
  return p;
}

int prod(int x, int y);
 req true;
 ens res == x * y;
```

Program transformation

Alpha-rename the body of the verified functions

```
// Hardened module
// (Functional part)
static int _fac(int x) {
  if (x == 0) return 1;
  int p = _prod(x,
    _fac(x-1));
  return p;
}

// (Boundary part)
static
int _prod(int x, int y){
  int r = prod(x, y);
  if (! (r == x * y))
    trap();
  return r;
}

int fac(int x) {
  if (! (x >= 0)) trap();
  return _fac(x);
}
```

Checking interface contracts

```
// Original module
int fac(int x)
 req x >= 0;
 ens res == fact(x);
{
  if (x == 0) return 1;
  int p = prod(x,
    fac(x-1));
  return p;
}

int prod(int x, int y);
 req true;
 ens res == x * y;
```

Program transformation

On entry, check validity of the precondition

```
// Hardened module
// (Functional part)
static int _fac(int x) {
  if (x == 0) return 1;
  int p = _prod(x,
    _fac(x-1));
  return p;
}

// (Boundary part)
static
int _prod(int x, int y){
  int r = prod(x, y);
  if (! (r == x * y))
    trap();
  return r;
}
```

```
int fac(int x) {
  if (! (x >= 0)) trap();
  return _fac(x);
}
```

Checking interface contracts

```
// Original module
int fac(int x)
 req x >= 0;
 ens res == fact(x);
{
  if (x == 0) return 1;
  int p = prod(x,
    fac(x-1));
  return p;
}
```

```
int prod(int x, int y);
 req true;
 ens res == x * y;
```

Program transformation

```
// Hardened module
// (Functional part)
static int _fac(int x) {
  if (x == 0) return 1;
  int p = _prod(x,
    _fac(x-1));
  return p;
}

// (Boundary part)
static
int _prod(int x, int y){
  int r = prod(x, y);
  if (! (r == x * y))
    trap();
  return r;
}

int fac(int x) {
  if (! (x >= 0)) trap();
  return _fac(x);
}
```

On outcall, check validity of the postcondition

Structure of the talk

- Overview
- Low-level platform protection mechanisms
- Secure compilation of mini-C
- Handling C-style dynamic memory allocation
- Implementation
- Conclusions

Example context + module

Context

```
// Prototypes
int med(struct lst *l);

// Unverified functions
int main()
{
  struct lst *l =
    input_list();
  output_int(med(l));
}

void srt(struct lst *l)
{
  < unverified sort
    implementation >
}
```

Module

```
// Prototypes
void srt(struct lst *l);

// Verified functions
int med(struct lst *l)

{
 int s = len(l);
 struct lst *l0 =
   copy(l);
 srt(l0);
 return nth(l0, s/2);
}
```

Example context + module

Context

```
// Prototypes
int med(struct lst *l);

// Unverified functions
int main()
{
  struct lst *l =
    input_list();
  output_int(med(l));
}

void srt(struct lst *l)
{
  < unverified sort
    implementation >
}
```

Module

```
// Prototypes
void srt(struct lst *l);

// Verified functions
int med(struct lst *l)

{
 int s = len(l);
 struct lst *l0 =
   copy(l);
 srt(l0);
 return nth(l0, s/2);
}
```

Example context + module

Context

```
// Prototypes
int med(struct lst *l);

// Unverified functions
int main()
{
  struct lst *l =
    input_list();
  output_int(med(l));
}

void srt(struct lst *l)
{
  < unverified sort
    implementation >
}
```

Module

```
// Prototypes
void srt(struct lst *l);

// Verified functions
int med(struct lst *l)

{
 int s = len(l);
 struct lst *l0 =
   copy(l);
 srt(l0);

 return nth(l0, s/2);
}
```

DistriNet
iMinds KU LEUVEN

Example context + module

Context

```
// Prototypes
int med(struct lst *l);

// Unverified functions
int main()
{
  struct lst *l =
    input_list();
  output_int(med(l));
}

void srt(struct lst *l)
{
  < unverified sort
    implementation >
}
```

Module

```
// Prototypes
void srt(struct lst *l);

// Verified functions
int med(struct lst *l)

{
 int s = len(l);
 struct lst *l0 =
   copy(l);
 srt(l0);

 return nth(l0, s/2);
}
```

DistriNet
iMinds KU LEUVEN

Example context + module

Context

```
// Prototypes
int med(struct lst *l);

// Unverified functions
int main()
{
  struct lst *l =
    input_list();
  output_int(med(l));
}

void srt(struct lst *l)
{
  < unverified sort
    implementation >
}
```

Module

```
// Prototypes
void srt(struct lst *l);

// Verified functions
int med(struct lst *l)

{
 int s = len(l);
 struct lst *l0 =
   copy(l);
 srt(l0);

 return nth(l0, s/2);
}
```

DistriNet
iMinds KU LEUVEN

Example context + module

Context

```
// Prototypes
int med(struct lst *l);

// Unverified functions
int main()
{
  struct lst *l =
    input_list();
  output_int(med(l));
}

void srt(struct lst *l)
{
  < unverified sort
    implementation >
}
```

Module

```
// Prototypes
void srt(struct lst *l);

// Verified functions
int med(struct lst *l)

{
 int s = len(l);
 struct lst *l0 =
   copy(l);
 srt(l0);
 return nth(l0, s/2);
}
```

DistriNet
iMinds KU LEUVEN

Example context + module

Context

```
// Prototypes
int med(struct lst *l);

// Unverified functions
int main()
{
  struct lst *l =
    input_list();
  output_int(med(l));
}

void srt(struct lst *l)
{
  < unverified sort
    implementation >
}
```

Module

```
// Prototypes
void srt(struct lst *l);

// Verified functions
int med(struct lst *l)
{
 int s = len(l);
 struct lst *l0 =
   copy(l);
 srt(l0);
 return nth(l0, s/2);
}
```

DistriNet
iMinds KU LEUVEN

Example context + module

Context

```
// Prototypes
int med(struct lst *l);

// Unverified functions
int main()
{
  struct lst *l =
    input_list();
  output_int(med(l));
}

void srt(struct lst *l)
{
  < unverified sort
    implementation >
}
```

Module

```
// Prototypes
void srt(struct lst *l);

// Verified functions
int med(struct lst *l)

{
 int s = len(l);
 struct lst *l0 =
   copy(l);
 srt(l0);
 return nth(l0, s/2);
}
```

DistriNet
iMinds KU LEUVEN

Example context + module

Context

```
// Prototypes
int med(struct lst *l);

// Unverified functions
int main()
{
  struct lst *l =
    input_list();
  output_int(med(l));
}

void srt(struct lst *l)
{
  < unverified sort
    implementation >
}
```

Module

```
// Prototypes
void srt(struct lst *l);

// Verified functions
int med(struct lst *l)

{
 int s = len(l);
 struct lst *l0 =
   copy(l);
 srt(l0);

 return nth(l0, s/2);
}
```

DistriNet
iMinds KU LEUVEN

Example context + module

Context

```
// Prototypes
int med(struct lst *l);

// Unverified functions
int main()
{
  struct lst *l =
    input_list();
  output_int(med(l));
}

void srt(struct lst *l)
{
  < unverified sort
    implementation >
}
```

Module

```
// Prototypes
void srt(struct lst *l);
 req list(l, ?v0);
 ens list(l, ?v1)
  &*& val_eq(v0, v1)
  &*& sorted(v1);

// Verified functions
int med(struct lst *l)


{
 int s = len(l);
 struct lst *l0 =
   copy(l);
 srt(l0);

 return nth(l0, s/2);
}
```

DistriNet
iMinds KU LEUVEN

Example context + module

Context

```
// Prototypes
int med(struct lst *l);

// Unverified functions
int main()
{
  struct lst *l =
    input_list();
  output_int(med(l));
}

void srt(struct lst *l)
{
  < unverified sort
    implementation >
}
```

Module

```
// Prototypes
void srt(struct lst *l);
 req list(l, ?v0);
 ens list(l, ?v1)
  &*& val_eq(v0, v1)
  &*& sorted(v1);

// Verified functions
int med(struct lst *l)
 req list(l, ?v0)
  &*& 0 < length(v0);
 ens list(l, ?v0)
  &*& res == median(v0);
{
 int s = len(l);
 struct lst *l0 =
   copy(l);
 srt(l0);
 return nth(l0, s/2);
}
```

Example context + module

Context

```
// Prototypes
int med(struct lst *l);

// Unverified functions
int main()
{
  struct lst *l =
    input_list();
  output_int(med(l));
}

void srt(struct lst *l)
{
  < unverified sort
    implementation >
}
```

Module

```
// Prototypes
void srt(struct lst *l);
 req list(l, ?v0);
 ens list(l, ?v1)
  &*& val_eq(v0, v1)
  &*& sorted(v1);

// Verified functions
int med(struct lst *l)
 req list(l, ?v0)
  &*& 0 < length(v0);
 ens list(l, ?v0)
  &*& res == median(v0);
{
 int s = len(l);
 struct lst *l0 =
   copy(l);
 srt(l0);
 <proof statements>
 return nth(l0, s/2);
}
```

Example context + module

Context

```
// Prototypes
int med(struct lst *l);

// Unverified functions
int main()
{
  struct lst *l =
    input_list();
  output_int(med(l));
}

void srt(struct lst *l)
{
  < unverified sort
    implementation >
}
```

Module

```
// Prototypes
void srt(struct lst *l);
 req list(l, ?v0);
 ens list(l, ?v1)
  &*& val_eq(v0, v1)
  &*& sorted(v1);

// Verified functions
int med(struct lst *l)
 req list(l, ?v0)
  &*& 0 < length(v0);
 ens list(l, ?v0)
  &*& res == median(v0);
{
 int s = len(l);
 struct lst *l0 =
   copy(l);
 srt(l0);
 <proof statements>
 return nth(l0, s/2);
}
```

Example context + module

Context

```
// Prototypes
int med(struct lst *l);

// Unverified functions
int main()
{
  struct lst *l =
    input_list();
  output_int(med(l));
}

void srt(struct lst *l)
{
  < unverified sort
    implementation >
}
```

Module

```
// Prototypes
void srt(struct lst *l);
 req list(l, ?v0);
 ens list(l, ?v1)
  &*& val_eq(v0, v1)
  &*& sorted(v1);

// Verified functions
int med(struct lst *l)
 req list(l, ?v0)
  &*& 0 < length(v0);
 ens list(l, ?v0)
  &*& res == median(v0);
{
 int s = len(l);
 struct lst *l0 =
   copy(l);
 srt(l0);
 <proof statements>
 return nth(l0, s/2);
}
```

DistriNet
iMinds KU LEUVEN

Example context + module

Context

```
// Prototypes
int med(struct lst *l);

// Unverified functions
int main()
{
  struct lst *l =
    input_list();
  output_int(med(l));
}

void srt(struct lst *l)
{
  < unverified sort
    implementation >
}
```

Module

```
// Prototypes
void srt(struct lst *l);
 req list(l, ?v0);
 ens list(l, ?v1)
  &*& val_eq(v0, v1)
  &*& sorted(v1);

// Verified functions
int med(struct lst *l)
 req list(l, ?v0)
  &*& 0 < length(v0);
 ens list(l, ?v0)
  &*& res == median(v0);
{
 int s = len(l);
 struct lst *l0 =
   copy(l);
 srt(l0);
 <proof statements>
 return nth(l0, s/2);
}
```

DistriNet
iMinds KU LEUVEN

Example context + module

Context

```
// Prototypes
int med(struct lst *l);

// Unverified functions
int main()
{
  struct lst *l =
    input_list();
  output_int(med(l));
}

void srt(struct lst *l)
{
  < unverified sort
    implementation >
}
```

Module

```
// Prototypes
void srt(struct lst *l);
 req list(l, ?v0);
 ens list(l, ?v1)
  &*& val_eq(v0, v1)
  &*& sorted(v1);

// Verified functions
int med(struct lst *l)
 req list(l, ?v0)
  &*& 0 < length(v0);
 ens list(l, ?v0)
  &*& res == median(v0);
{
 int s = len(l);
 struct lst *l0 =
   copy(l);
 srt(l0);
 <proof statements>
 return nth(l0, s/2);
}
```

DistriNet
iMinds KU LEUVEN

Example context + module

Context

```
// Prototypes
int med(struct lst *l);

// Unverified functions
int main()
{
  struct lst *l =
    input_list();
  output_int(med(l));
}

void srt(struct lst *l)
{
  < unverified sort
    implementation >
}
```

Module

```
// Prototypes
void srt(struct lst *l);
 req list(l, ?v0);
 ens list(l, ?v1)
  &*& val_eq(v0, v1)
  &*& sorted(v1);

// Verified functions
int med(struct lst *l)
 req list(l, ?v0)
  &*& 0 < length(v0);
 ens list(l, ?v0)
  &*& res == median(v0);
{
 int s = len(l);
 struct lst *l0 =
   copy(l);
 srt(l0);
 <proof statements>
 return nth(l0, s/2);
}
```

DistriNet
iMinds KU LEUVEN

Example context + module

Context

```
// Prototypes
int med(struct lst *l);

// Unverified functions
int main()
{
  struct lst *l =
    input_list();
  output_int(med(l));
}

void srt(struct lst *l)
{
  < unverified sort
    implementation >
}
```

Module

```
// Prototypes
void srt(struct lst *l);
 req list(l, ?v0);
 ens list(l, ?v1)
  &*& val_eq(v0, v1)
  &*& sorted(v1);

// Verified functions
int med(struct lst *l)
 req list(l, ?v0)
  &*& 0 < length(v0);
 ens list(l, ?v0)
  &*& res == median(v0);
{
 int s = len(l);
 struct lst *l0 =
   copy(l);
 srt(l0);
 <proof statements>
 return nth(l0, s/2);
}
```

DistriNet
iMinds KU LEUVEN

Example context + module

Context

```
// Prototypes
int med(struct lst *l);

// Unverified functions
int main()
{
  struct lst *l =
    input_list();
  output_int(med(l));
}

void srt(struct lst *l)
{
  < unverified sort
    implementation >
}
```

Module

```
// Prototypes
void srt(struct lst *l);
 req list(l, ?v0);
 ens list(l, ?v1)
  &*& val_eq(v0, v1)
  &*& sorted(v1);

// Verified functions
int med(struct lst *l)
 req list(l, ?v0)
  &*& 0 < length(v0);
 ens list(l, ?v0)
  &*& res == median(v0);
{
 int s = len(l);
 struct lst *l0 =
   copy(l);
 srt(l0);
 <proof statements>
 return nth(l0, s/2);
}
```

DistriNet
iMinds KU LEUVEN

Example context + module

Context

```
// Prototypes
int med(struct lst *l);

// Unverified functions
int main()
{
  struct lst *l =
    input_list();
  output_int(med(l));
}

void srt(struct lst *l)
{
  < unverified sort
    implementation >
}
```

Module

```
// Prototypes
void srt(struct lst *l);
 req list(l, ?v0);
 ens list(l, ?v1)
  &*& val_eq(v0, v1)
  &*& sorted(v1);

// Verified functions
int med(struct lst *l)
 req list(l, ?v0)
  &*& 0 < length(v0);
 ens list(l, ?v0)
  &*& res == median(v0);
{
 int s = len(l);
 struct lst *l0 =
   copy(l);
 srt(l0);
 <proof statements>
 return nth(l0, s/2);
}
```

DistriNet
iMinds KU LEUVEN

Extending the interface checks

- Before, interface checks were just assertion checks:
  - On entering a function of the module
  - On returning from an outcall
- Now, these assertions can be “spatial” assertions:
  - Interface checks should also maintain the **footprint** of the module
  - And check integrity of that footprint
    - On entering a function of the module
    - On returning from an outcall

Spatial assertions

```
// Original module
struct pair {int a, b;};


void f(struct pair* p)
 req p->a |-> ?a &*&
     p->b |-> ?b
 ens p->a |-> _ &*&
     p->b |-> _;
{
 <...>
 ct(p);
 <...>
}


void ct(struct pair* p);
 req p->a |-> ?n;
 ens p->a |-> ?m &*&
     m == n + 1;
```

Program transformation

```
// Hardened module
struct pair {int a, b;};

static
void _f(struct pair* p) {
 <...>
 _ct(p);
 <...>
}

static
void _ct(struct pair* p) {
 char h0[32], h1[32];
 int n = intp(&(p->a),C);
 fp_hash(h0);
 ct(p);
 fp_hash(h1);
 if (!eq(h0, h1)) trap();
 int m = intp(&(p->a),P);
 if (m != n+1) trap();
}

void f(struct pair* p) {
 a = intp(&(p->a),P);
 b = intp(&(p->b),P);
 _f(p);
 intp(&(p->a),C);
 intp(&(p->b),C);
}
```

Spatial assertions

```
// Original module
struct pair {int a, b;};


void f(struct pair* p)
 req p->a |-> ?a &*&
     p->b |-> ?b
 ens p->a |-> _ &*&
     p->b |-> _;
{
 <...>
 ct(p);
 <...>
}


void ct(struct pair* p);
 req p->a |-> ?n;
 ens p->a |-> ?m &*&
     m == n + 1;
```

Program transformation

On entry:

Produce the footprint of the precondition

```
// Hardened module
struct pair {int a, b;};

static
void _f(struct pair* p) {
 <...>
 _ct(p);
 <...>
}

static
void _ct(struct pair* p) {
 char h0[32], h1[32];
 int n = intp(&(p->a),C);
 fp_hash(h0);
 ct(p);
 fp_hash(h1);
 if (!eq(h0, h1)) trap();
 int m = intp(&(p->a),P);
 if (m != n+1) trap();
}

void f(struct pair* p) {
 a = intp(&(p->a),P);
 b = intp(&(p->b),P);
 _f(p);
 intp(&(p->a),C);
 intp(&(p->b),C);
}
```

Spatial assertions

```
// Original module
struct pair {int a, b;};


void f(struct pair* p)
 req p->a |-> ?a &*&
     p->b |-> ?b
 ens p->a |-> _ &*&
     p->b |-> _;
{
 <...>
 ct(p);
 <...>
}


void ct(struct pair* p);
 req p->a |-> ?n;
 ens p->a |-> ?m &*&
     m == n + 1;
```

Program transformation

On return:

Consume the footprint of the postcondition

```
// Hardened module
struct pair {int a, b;};

static
void _f(struct pair* p) {
 <...>
 _ct(p);
 <...>
}

static
void _ct(struct pair* p) {
 char h0[32], h1[32];
 int n = intp(&(p->a),C);
 fp_hash(h0);
 ct(p);
 fp_hash(h1);
 if (!eq(h0, h1)) trap();
 int m = intp(&(p->a),P);
 if (m != n+1) trap();
}

void f(struct pair* p) {
 a = intp(&(p->a),P);
 b = intp(&(p->b),P);
 _f(p);
 intp(&(p->a),C);
 intp(&(p->b),C);
}
```

Spatial assertions

```
// Original module
struct pair {int a, b;};


void f(struct pair* p)
 req p->a |-> ?a &*&
     p->b |-> ?b
 ens p->a |-> _ &*&
     p->b |-> _;
{
 <...>
 ct(p);
 <...>
}


void ct(struct pair* p);
 req p->a |-> ?n;
 ens p->a |-> ?m &*&
     m == n + 1;
```

Program transformation

On outcall:

Consume the footprint of the precondition

```
// Hardened module
struct pair {int a, b;};

static
void _f(struct pair* p) {
 <...>
 _ct(p);
 <...>
}

static
void _ct(struct pair* p) {
 char h0[32], h1[32];
 int n = intp(&(p->a),C);
 fp_hash(h0);
 ct(p);
 fp_hash(h1);
 if (!eq(h0, h1)) trap();
 int m = intp(&(p->a),P);
 if (m != n+1) trap();
}

void f(struct pair* p) {
 a = intp(&(p->a),P);
 b = intp(&(p->b),P);
 _f(p);
 intp(&(p->a),C);
 intp(&(p->b),C);
}
```

Spatial assertions

```
// Original module
struct pair {int a, b;};


void f(struct pair* p)
 req p->a |-> ?a &*&
     p->b |-> ?b
 ens p->a |-> _ &*&
     p->b |-> _;
{
 <...>
 ct(p);
 <...>
}


void ct(struct pair* p);
 req p->a |-> ?n;
 ens p->a |-> ?m &*&
     m == n + 1;
```

Program transformation

On return from outcall:

Produce the footprint of the postcondition

```
// Hardened module
struct pair {int a, b;};

static
void _f(struct pair* p) {
 <...>
 _ct(p);
 <...>
}

static
void _ct(struct pair* p) {
 char h0[32], h1[32];
 int n = intp(&(p->a),C);
 fp_hash(h0);
 ct(p);
 fp_hash(h1);
 if (!eq(h0, h1)) trap();
 int m = intp(&(p->a),P);
 if (m != n+1) trap();
}

void f(struct pair* p) {
 a = intp(&(p->a),P);
 b = intp(&(p->b),P);
 _f(p);
 intp(&(p->a),C);
 intp(&(p->b),C);
}
```

Spatial assertions

```
// Original module
struct pair {int a, b;};


void f(struct pair* p)
 req p->a |-> ?a &*&
     p->b |-> ?b
 ens p->a |-> _ &*&
     p->b |-> _;
{
 <...>
 ct(p);
 <...>
}


void ct(struct pair* p);
 req p->a |-> ?n;
 ens p->a |-> ?m &*&
     m == n + 1;
```

Frame rule for ct

Program transformation

And check that the context did not modify the contents of the footprint of the module

```
// Hardened module
struct pair {int a, b;};

static
void _f(struct pair* p) {
 <...>
 _ct(p);
 <...>
}

static
void _ct(struct pair* p) {
 char h0[32], h1[32];
 int n = intp(&(p->a),C);
 fp_hash(h0);
 ct(p);
 fp_hash(h1);
 if (!eq(h0, h1)) trap();
 int m = intp(&(p->a),P);
 if (m != n+1) trap();
}

void f(struct pair* p) {
 a = intp(&(p->a),P);
 b = intp(&(p->b),P);
 _f(p);
 intp(&(p->a),C);
 intp(&(p->b),C);
}
```

Main theorems

Theorem 2 (Safety). *For a command c and well-formed assertions a_{pre} and a_{post} such that $\Gamma \vdash \{a_{pre}\}\ c\ \{a_{post}\}$, we have* $\forall \Delta.\ \mathsf{nofail}(\Delta) \models \{\top\}\ [c]_{\Gamma, a_{pre}}\ \{\top\}$.

Theorem 3 (Precision). *For a command c and well-formed assertions a_{pre} and a_{post} such that $\Gamma \vdash \{a_{pre}\}\ c\ \{a_{post}\}$, we have that* $\forall \Delta.\ \Delta \models \Gamma \Rightarrow \Delta \models \{a_{pre}\}\ [c]_{\Gamma, a_{pre}}\ \{a_{post}\}$.

For details, see: Agten et al. *Sound modular verification of C code executing in an unverified context*, POPL 2015

DistriNet iMinds KU LEUVEN

Structure of the talk

- Overview
- Low-level platform protection mechanisms
- Secure compilation of mini-C
- Handling C-style dynamic memory allocation
- Implementation
- Conclusions

Implementation

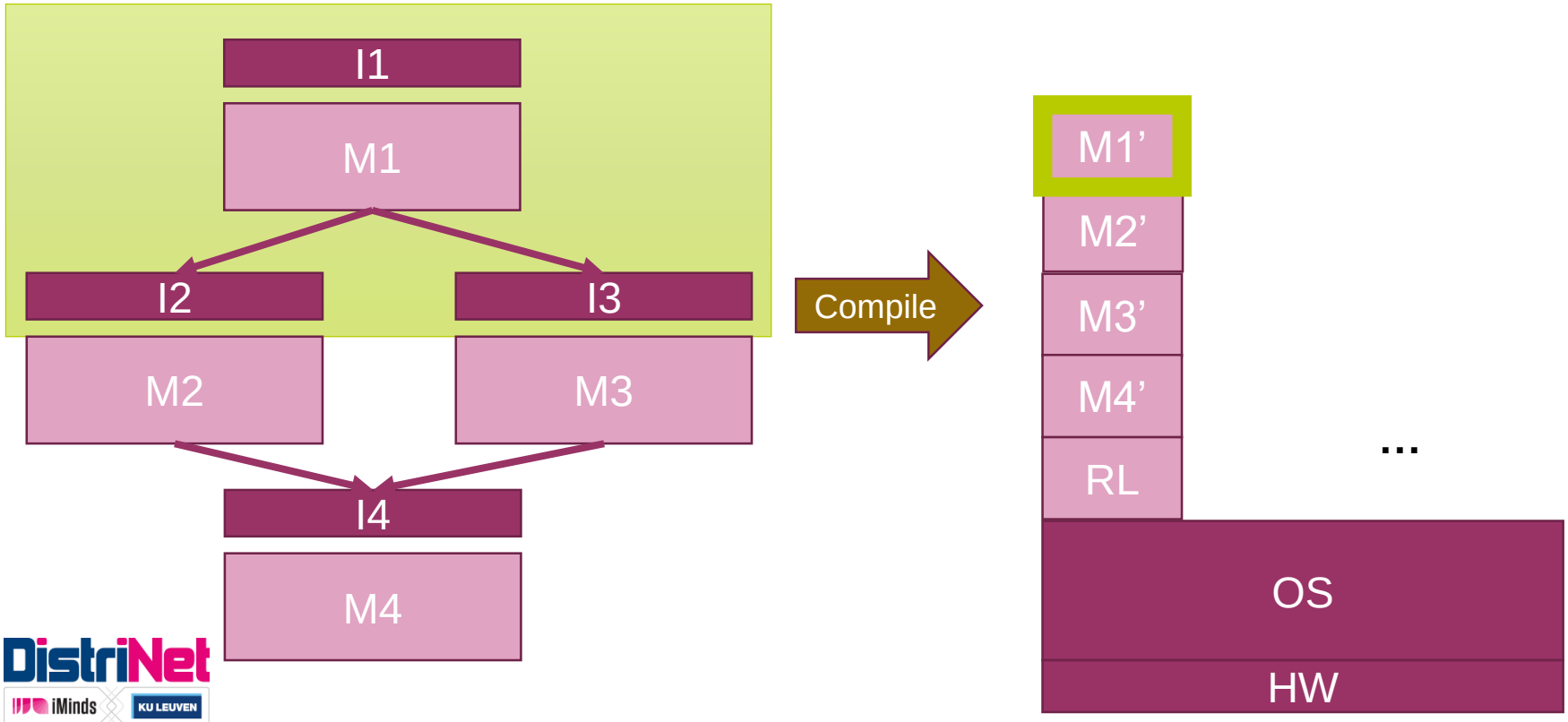
- We have an end-to-end implementation
  - Verifier = VeriFast [very mature]
  - Program transformations for run time contract checking support a subset of C and VeriFasts program logic [prototype]
  - Secure compiler is an LLVM based “pragmatic” implementation of a fully abstract compiler [initial prototype]
  - Protected Module Architecture is either Sancus or Fides [stable prototypes], but should be Intel SGX soon
- Benchmarks show acceptable costs

	Execution time (s)		
	unhardened	hardened	overhead
mod_authn_anon	33.164	33.388	0.224 (0.68)%
mod_authn_file	33.554	34.809	1.255 (3.74)%
ftpd	23.193	23.242	0.049 (0.21)%

Conclusions

A property verified of M1 is true at run time, relying only on:

- Soundness of the verifier
- Correctness and security of the compiler (including the runtime checks discussed in this talk)
- Correctness of the hardware (including the memory access control)



DistriNet
iMinds
KU LEUVEN

Future Work

- Implementation and benchmarking on Intel SGX
- Handling concurrency
  - VeriFast is sound for concurrent code
  - But clearly, the current run time contract checks are not
- Preserving relational program properties
  - For instance non-interference
  - The current run time checks are only sound for safety properties
- Evaluating other low-level protection mechanisms
- …

Questions?

Thank you!

References

- **P. Agten, B. Jacobs, F. Piessens, Sound modular verification of C code executing in an unverified context, POPL 2015**
- J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, F. Piessens, *Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base*, USENIX Security 2013
- P. Agten, R. Strackx, B. Jacobs, F. Piessens, Secure compilation to modern processors, CSF 2012
- M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, F. Piessens, Secure compilation to Protected Module Architectures, TOPLAS 2015
- B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, F. Piessens, VeriFast: A powerful, sound, predictable, fast verifier for C and Java, NASA Formal Methods 2011