Frama-C for Code Security

Julien Signoles

Software Safety & Security Lab

Cyber in Saclay

February 11, 2021

Outline

1. Frama-C at a Glance
2. Deductive Verification with ACSL and WP
3. Value Analysis with Eva
4. Runtime Verification with E-ACSL
5. Combining Analyses for Security-Oriented Verifications

Part I

Frama-C at a Glance

Frama-C Distribution

Framework for analyses of source code written in ISO 99 C

[Kirchner et al, 2015]

- developed by CEA LIST since 2005
- almost open source (LGPL 2.1)
- first open-source release aka Hydrogen in 2008
- last open-source release aka 22-Titanium in November 2020

http://frama-c.com

- also proprietary extensions and distributions
- targets both academic and industrial usages

Frama-C, a Collection of Tools

Several tools inside a single platform

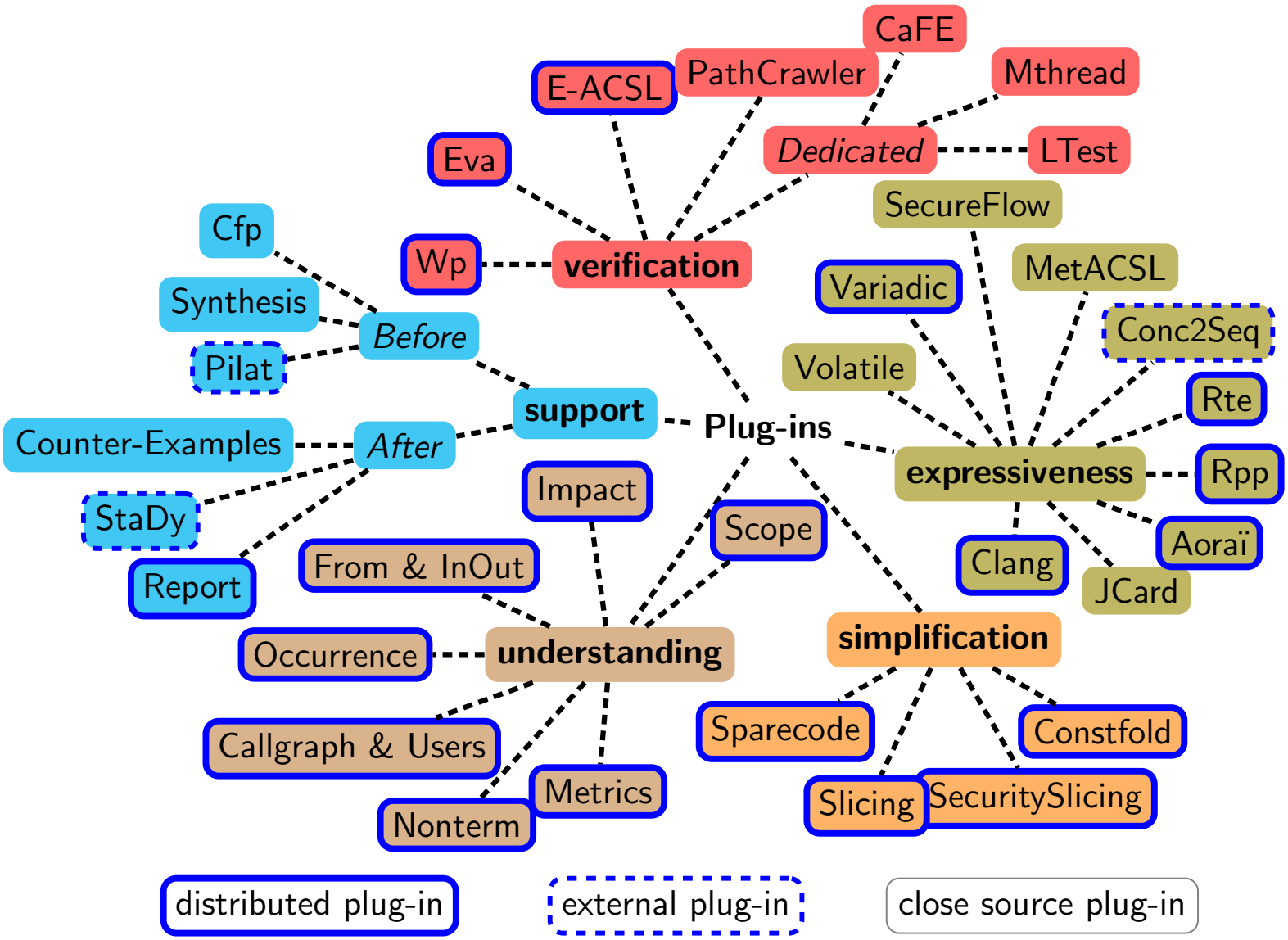
- plug-in architecture *à la* Eclipse [Signoles, 2015]
- tools provided as plug-ins
  - 31 plug-ins in the latest open source distribution
  - outside open source plug-ins
  - close source plug-ins, either at CEA (> 20) or outside

Frama-C, a Collection of Tools

Several tools inside a single platform

- plug-in architecture *à la* Eclipse [Signoles, 2015]
- tools provided as plug-ins
  - 31 plug-ins in the latest open source distribution
  - outside open source plug-ins
  - close source plug-ins, either at CEA (> 20) or outside
- plug-ins connected to a kernel
  - provides an uniform setting (command lines, AST, etc)
  - provides general services (data structures, etc)
  - synthesizes useful information (proved properties, etc)
  - analyzer combinations [Correnson & Signoles, 2012]

Frama-C Plug-ins Gallery



Frama-C, a Development Platform

- developed in OCaml
- was based on Cil [Necula et al, 2002]
- library dedicated to analysis of C code

development of plug-ins by third party

Frama-C, a Development Platform

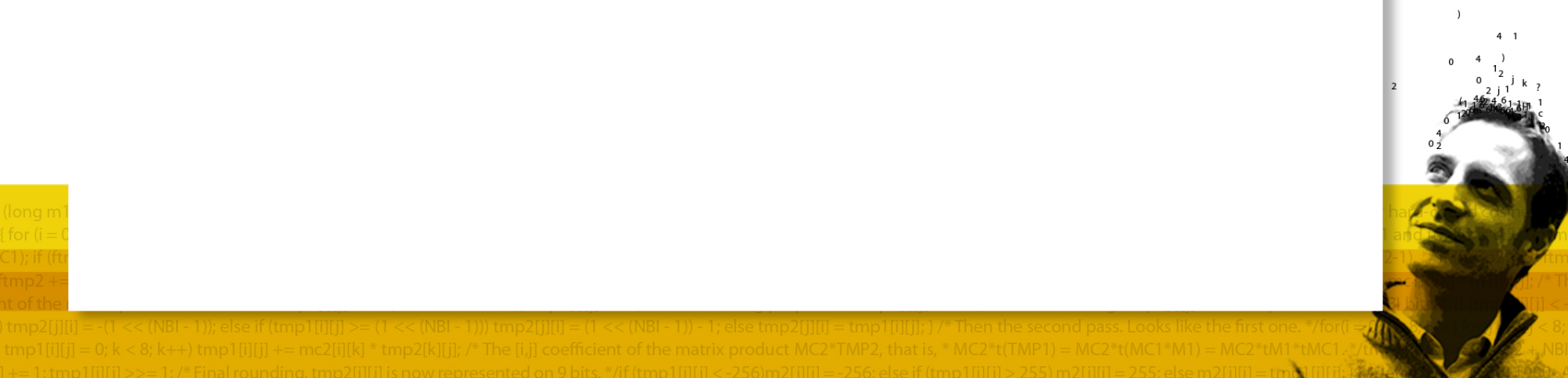
- developed in OCaml
- was based on Cil [Necula et al, 2002]
- library dedicated to analysis of C code

development of plug-ins by third party

- powerful low-cost analyser
- dedicated plug-in for specific task (coding rules verifier)
- dedicated plug-in for fine-grain parameterization
- extension of existing analysers

Part II

Proof of Formal Specifications with ACSL and WP



Deductive Verification in a Nutshell

Objectives

Rigorous, mathematical proof of program semantic properties

- functional properties
  - does a function correctly implement its specification?

Deductive Verification in a Nutshell

Objectives

Rigorous, mathematical proof of program semantic properties

- functional properties
  - does a function correctly implement its specification?
- safety properties: ensure that the program does not fail
  - no undefined behaviors (w.r.t. ISO C99)
    - no division by zero,
    - no arithmetic overflow,
    - all memory accesses are valid, ...
  - if invalid, most of them lead to security vulnerabilities

Deductive Verification in a Nutshell

Objectives

Rigorous, mathematical proof of program semantic properties

- functional properties
  - does a function correctly implement its specification?
- safety properties: ensure that the program does not fail
  - no undefined behaviors (w.r.t. ISO C99)
    - no division by zero,
    - no arithmetic overflow,
    - all memory accesses are valid, ...
  - if invalid, most of them lead to security vulnerabilities
- termination
  - does my program terminate (when expected)?

Deductive Verification in a Nutshell

Input/Output

- based on weakest precondition calculus [Dijkstra, 1975]
- Modular reasoning
  - handle each function independently
- input: function + precondition + postcondition
  - precondition = what the function assumes on its inputs
  - postcondition = what is guarantee as function's outputs if the precondition is satisfied
- output: does the precondition implies this property?

Deductive Verification in a Nutshell

Verification Process

deductive verification is **NOT** an automatic process

- backward program analysis to express the postcondition in terms of function's inputs

Deductive Verification in a Nutshell

Verification Process

deductive verification is **NOT** an automatic process

- backward program analysis to express the postcondition in terms of function's inputs
- for each loop, need to write a loop invariant that hold just before entering the loop and at the end of each iteration
- for termination, also need to write a loop variant for each loop (e.g., a nonnegative value that decreases at each loop iteration)

Deductive Verification in a Nutshell

Verification Process

deductive verification is **NOT** an automatic process

- backward program analysis to express the postcondition in terms of function's inputs
- for each loop, need to write a loop invariant that hold just before entering the loop and at the end of each iteration
- for termination, also need to write a loop variant for each loop (e.g., a nonnegative value that decreases at each loop iteration)
- generate verification conditions (VC) to be proven
- relies on external provers to prove them
  - automatic provers (Alt-Ergo, Z3, CVC4, ...)
  - proof assistants (Coq, PVS, Isabelle/HOL...)

ACSL, a Formal Specification Language for C

Specifications need to be formally expressed

- formal specification language for ISO C 99
- designed by CEA LIST and Inria Toccata
- Frama-C provides a reference implementation
- like JML for Java, spec# for C# and Spark2014 for Ada
- contract-based language like Eiffel
- designed for program proving
- based on first order typed polymorphic logic
- logical terms include pure C expressions, and also mathematical integers and reals

Function Contract

Function contracts:

- precondition: `requires`
  - what is assumed on the inputs (ensured by the callers)
- postcondition: `ensures`
  - what is guaranteed by the function body (assuming the precondition)
- written left-values: `assigns`
  - which memory locations may be modified by the function

Specific keywords:

- `\result`: the returned value
- `\old(x)`: value of `x` in the pre-state
- `\at(x,l)`: value of `x` at label `l`
- `\nothing` (in `assigns`): no left-value is written

Example 1

A First Program Proof

- specify and prove the following program

```
/* returns the absolute value of x */
int abs(int x) {
  if (x >= 0) return x;
  return -x;
}
```

- prove it with Frama-C/WP and, e.g., Alt-Ergo
  - `frama-c -wp file.c`
  - `frama-c-gui -wp file.c`
- also prove the absence of undefined behaviors
  - `frama-c -wp -wp-rte file.c`

Example 1

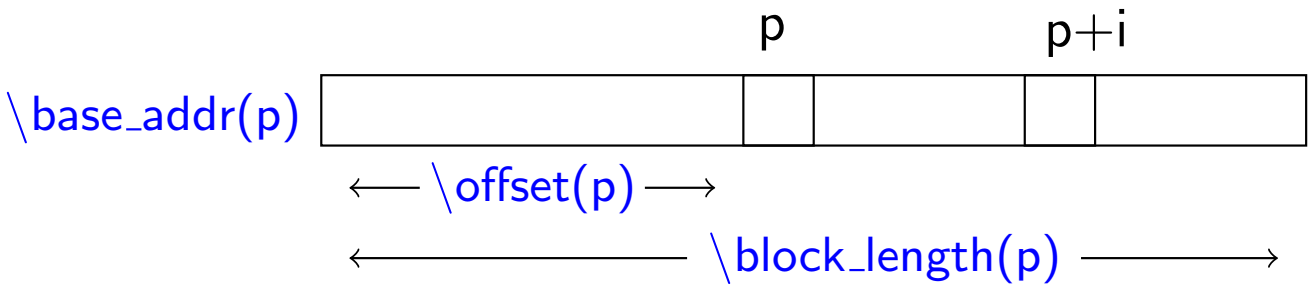
A First Program Proof

- the following specification is proved :-)

```
#include<limits.h>

/*@ requires x > INT_MIN;
  @ assigns \nothing;
  @
  @ behavior pos:
  @   assumes x >= 0;
  @   ensures \result == x;
  @
  @ behavior neg:
  @   assumes x < 0;
  @   ensures \result == -x;
  @
  @ complete behaviors;
  @ disjoint behaviors; */
int my_abs(int x) {
  if (x >= 0) return x;
  return -x;
}
```

ACSL and Memory Properties

- memory properties are important for code security
- ACSL provides built-ins memory-related predicates and functions
  - `\valid(p)`: whether $*p$ have been properly allocated
  - `\valid_read(p)`: same as `\valid(p)` but p is read only (e.g., literal string)
  - `\initialize(p)`: whether $*p$ is initialized
  - `\separated(p,q)`: p and q point to disjoint memory blocks
  - 

  - ...

Deductive Verification and Memory Models

- Proving memory properties requires a memory model, i.e. *"the way that the analysis tool models the storage of the underlying machine on which the code runs"* [Xu et al, 2010]

Deductive Verification and Memory Models

- Proving memory properties requires a memory model, i.e. *"the way that the analysis tool models the storage of the underlying machine on which the code runs"* [Xu et al, 2010]
- a memory model is alway a trade-off between expressivity and automation
  - if very low-level: very expressive but almost no automation
  - if very high-level: automatic proofs but poorly expressive
  - makes assumption about the code

Deductive Verification and Memory Models

- Proving memory properties requires a memory model, i.e. *"the way that the analysis tool models the storage of the underlying machine on which the code runs"* [Xu et al, 2010]
- a memory model is alway a trade-off between expressivity and automation
  - if very low-level: very expressive but almost no automation
  - if very high-level: automatic proofs but poorly expressive
  - makes assumption about the code
- Frama-C/WP may handle several memory models, including:
  - Hoare: automatic proofs but allows no pointers
  - Typed: good trade-off between automation and expressiveness, yet no heterogeneous casts
  - Typed+ref: more automatic than Typed, but assumes no aliasing

Second Example

- specify and prove the following program

```
// returns the index of the minimal element
// of the given array [a] of size [length]
int find_min(int* a, int length) {
  int min = a[0], min_idx = 0;
  for (int i = 1; i < length; i++) {
    if (a[i] < min) {
      min_idx = i;
      min = a[i];
    }
  }
  return min_idx;
}
```

Second Example

solution

```
/*@ requires length > 0 && \valid(a+(0..length-1));
  @ assigns \nothing;
  @ ensures \forall integer j;
  @   0 <= j < length ==> a[\result] <= a[j];
  @ ensures 0 <= \result < length; */
int find_min(int* a, int length) {
  int min = a[0], min_idx = 0;

  /*@ loop invariant 0<=i<=length && 0<=min_idx<length;
    @ loop invariant \forall integer j;
    @   0 <= j < i ==> min <= a[j];
    @ loop invariant a[min_idx] == min;
    @ loop assigns min, min_idx, i;
    @ loop variant length - i; */
  for (int i = 1; i < length; i++) {
    if (a[i] < min) {
      min_idx = i;
      min = a[i];
    }
  }
  return min_idx;
}
```

What do you (not) prove?

- what do you prove?
  - the source code matches the spec
  - under the assumptions of your toolset
    - check the manuals, emitted warnings, ...

What do you (not) prove?

- what do you prove?
  - the source code matches the spec
  - under the assumptions of your toolset
    - check the manuals, emitted warnings, ...
- what do you **not** prove?
  - the executed code does what you have in mind
    - same mistake in the code and in the spec
    - missed assumptions
    - analyzer bug: use Verasco [Jourdan, 2016]
    - compiler bug: use CompCert [Leroy, 2009]
    - hardware bug
    - ...

WP Applications

Avionic System

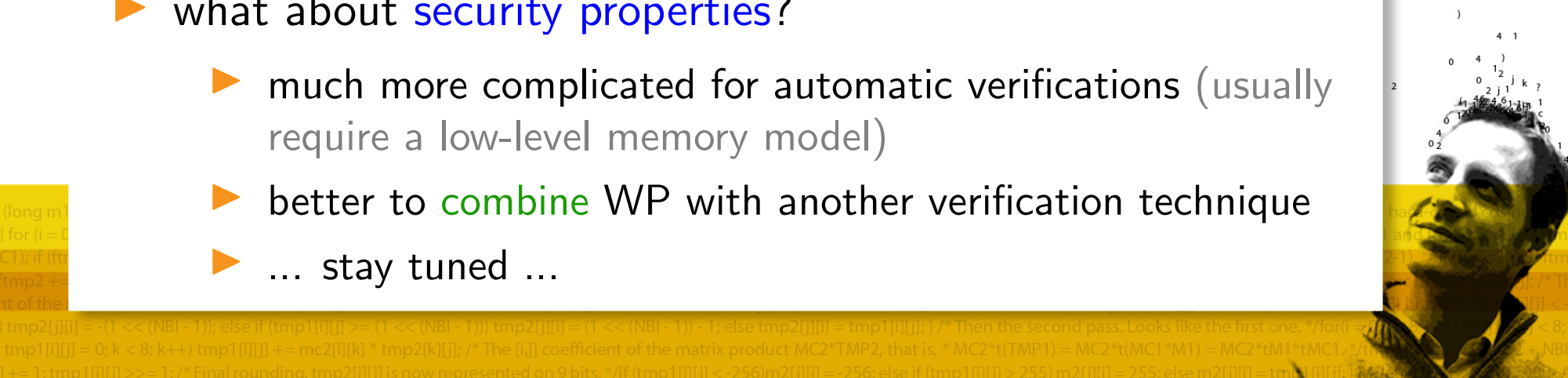
- used by Airbus for verifying safety properties of critical control-command code of avionic systems [Atki et al, 2018]

WP Applications

Avionic System

- used by Airbus for verifying safety properties of critical control-command code of avionic systems [Atki et al, 2018]

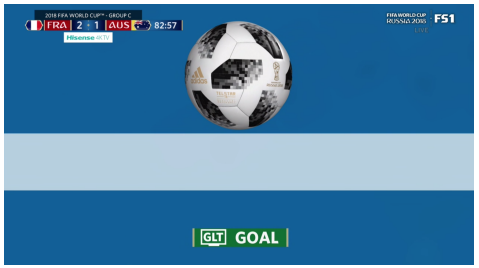
- what about security properties?
  - much more complicated for automatic verifications (usually require a low-level memory model)
  - better to combine WP with another verification technique
  - ... stay tuned ...



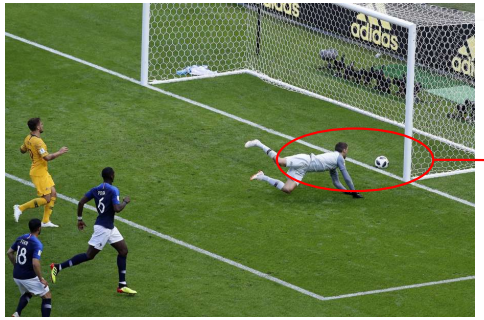
Part III

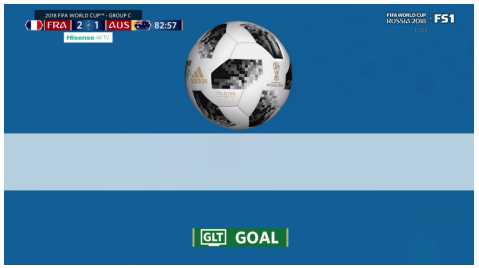
Value Analysis with Eva

Abstract Interpretation in two pictures



Abstract Interpretation in two pictures

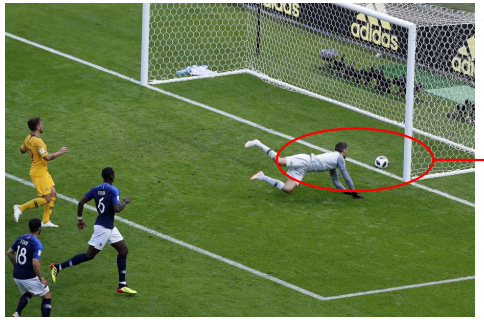


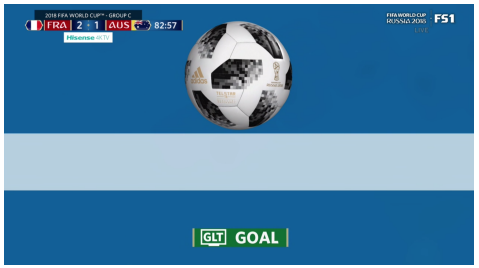


Abstract interpretation is about

- abstracting away information

Abstract Interpretation in two pictures

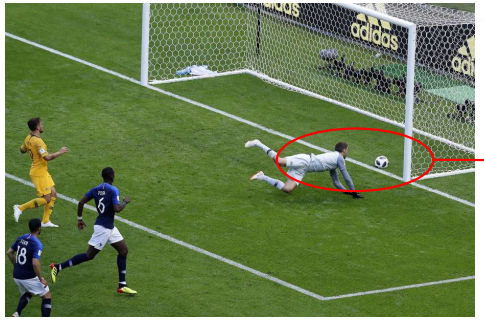


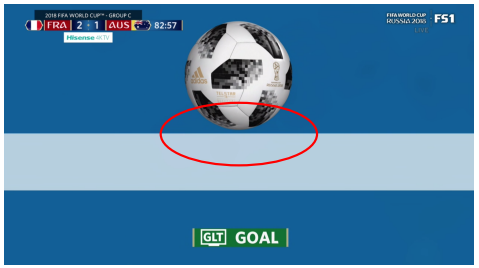


Abstract interpretation is about

- abstracting away information
- ensuring answer in a reasonable time

Abstract Interpretation in two pictures





Abstract interpretation is about

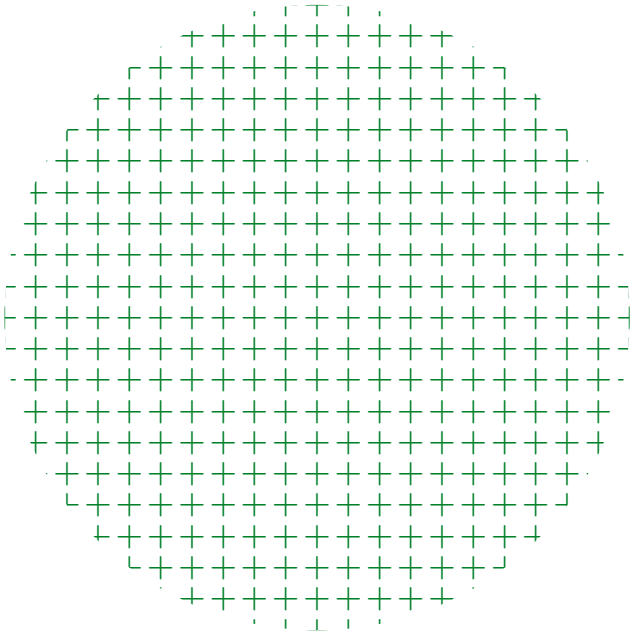
- abstracting away information
- ensuring answer in a reasonable time
- while retaining adequate precision

Abstract Interpretation in two pictures

Abstract interpretation is about

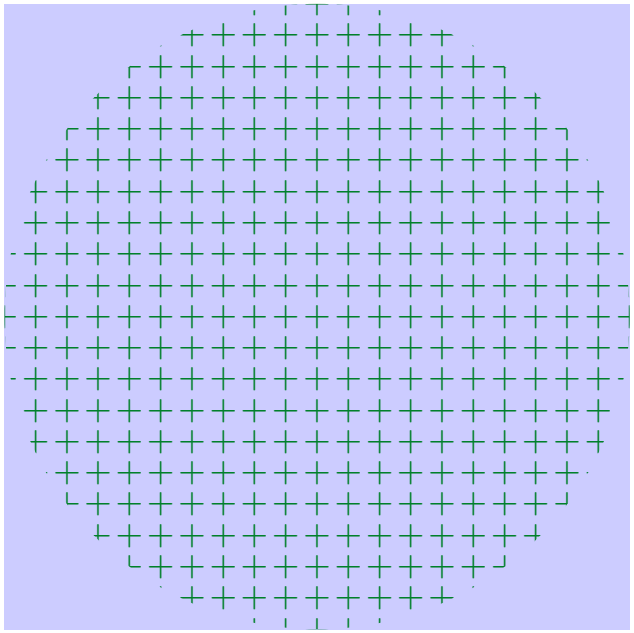
- abstracting away information
- ensuring answer in a reasonable time
- while retaining adequate precision
- and guaranteeing correct answers

Abstract Interpretation in a Nutshell



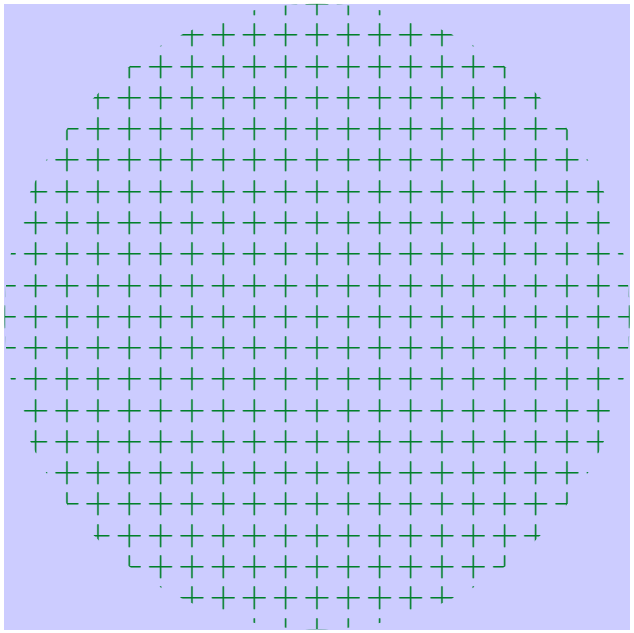
- replace all possible concrete executions ...

Abstract Interpretation in a Nutshell



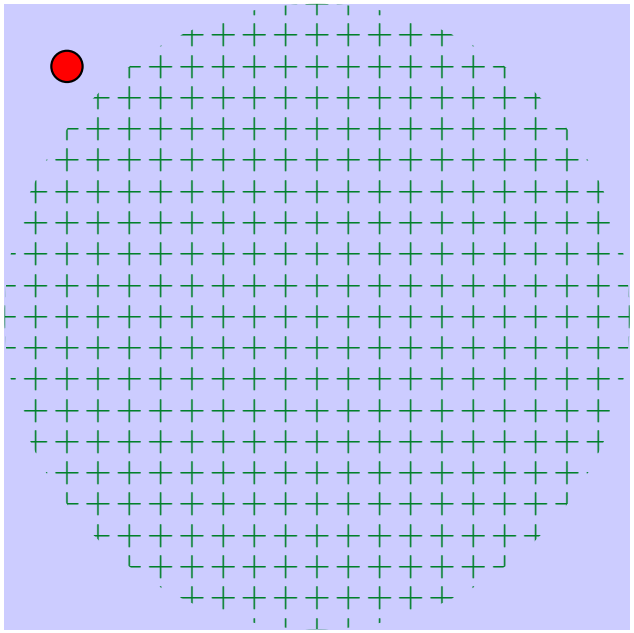
- replace all possible concrete executions ...
- ... by one abstract execution

Abstract Interpretation in a Nutshell



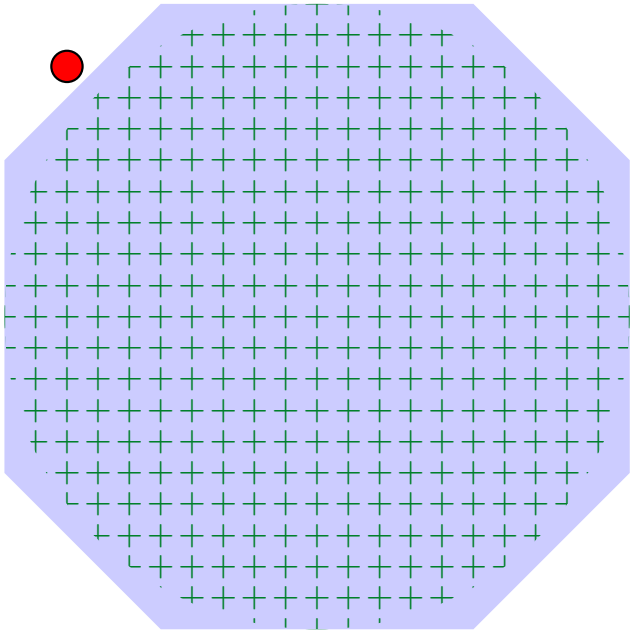
- replace all possible concrete executions ...
- ... by one abstract execution
- analysis is guaranteed to terminate

Abstract Interpretation in a Nutshell



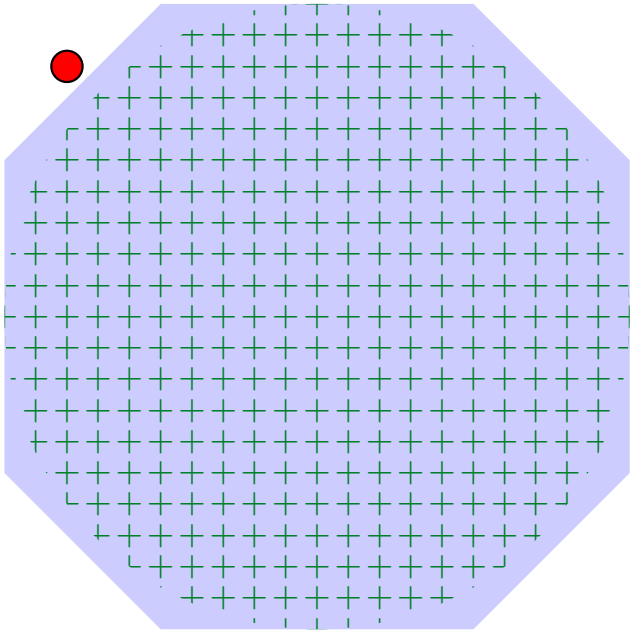
- replace all possible concrete executions ...
- ... by one abstract execution
- analysis is guaranteed to terminate
- over-approximation and false alarms

Abstract Interpretation in a Nutshell



- replace all possible concrete executions ...
- ... by one abstract execution
- analysis is guaranteed to terminate
- over-approximation and false alarms
- trade-off between precision and computation time

Abstract Interpretation in a Nutshell



- replace all possible concrete executions ...
- ... by one abstract execution
- analysis is guaranteed to terminate
- over-approximation and false alarms
- trade-off between precision and computation time

Theoretical Foundation introduced by Cousot

[Cousot & Cousot, 77]

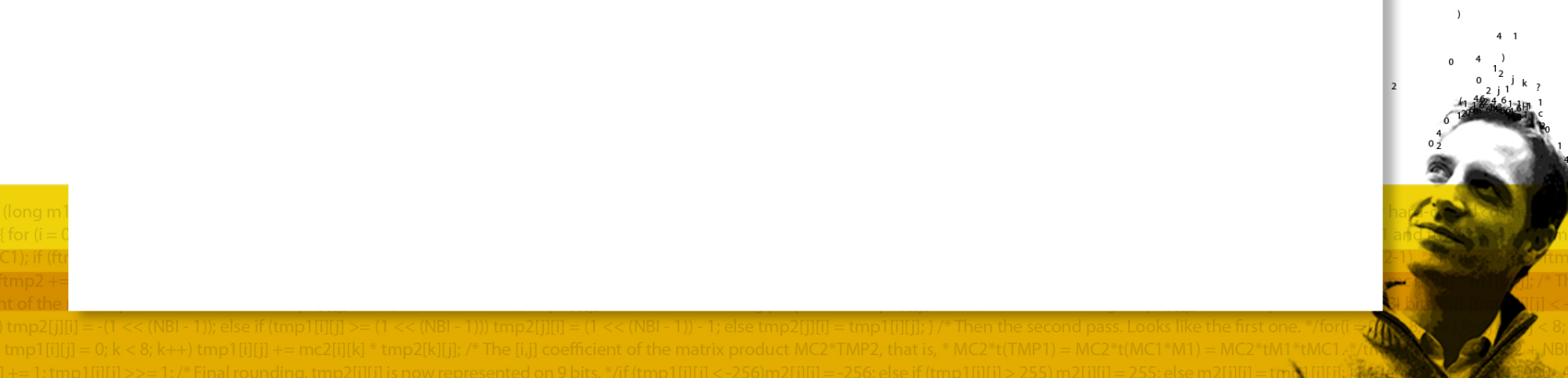
- One hard-wired non-relational domain
  - small sets of integers, e.g. $\{5, 18, 42\}$
  - reduced product of intervals: quick to compute, e.g. $[1..41]$
  - modulo: pretty good for arrays of structures, e.g. $[1..41], 1\%2$
  - precise representation of pointers, e.g. *32-bit aligned offset from* $\&t[0]$
  - initialization information

- One hard-wired non-relational domain
  - small sets of integers, e.g. $\{5, 18, 42\}$
  - reduced product of intervals: quick to compute, e.g. $[1..41]$
  - modulo: pretty good for arrays of structures, e.g. $[1..41], 1\%2$
  - precise representation of pointers, e.g. *32-bit aligned offset from* $\&t[0]$
  - initialization information
- alarms on potential undefined behaviors and invalid annotations
- highly optimized
  - excellent results on embedded critical code
  - possible usage in low-level C code

- generic analysis on the abstract domain
- allow combination of abstract domains and some inter-reductions of their states [Blazy et al, 2017]
- easy to add new domain
  - Apron's domains [Jeannet & Miné, 2009]
  - gauge [Venet, 2012]
  - conditional predicate [Blazy et al, 2016]
  - numerors [Jacquemin et al, 2018]
  - ...
  - contribute yourself :-)

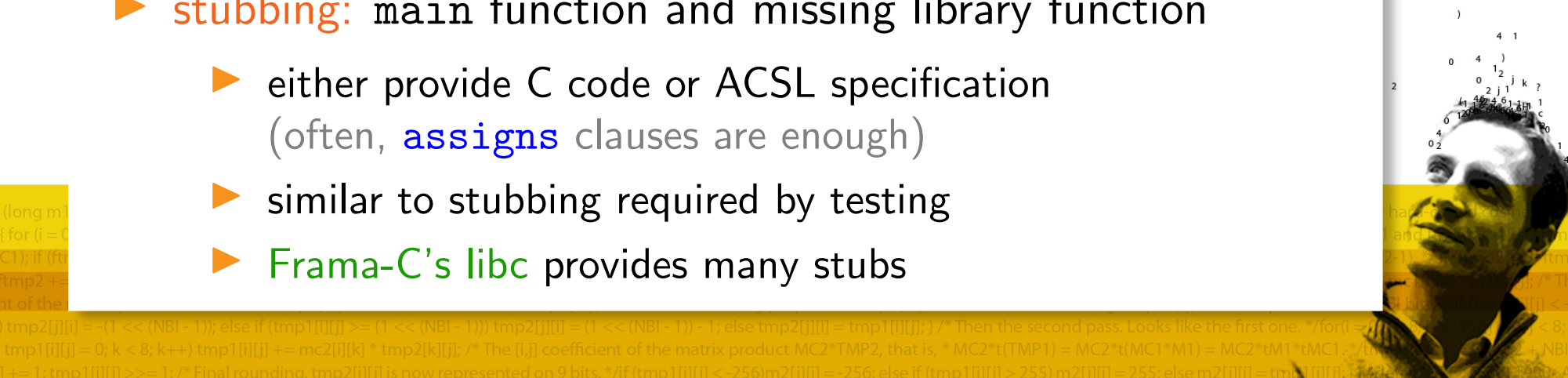
Eva Parameterization

- Eva is automatic
- Eva requires fine-tuned parameterization to be both precise and efficient
  - lots of parameters, but not so many almost always useful
  - try `-eva-precision n` ($0 \leq n \leq 11$)
- trade-off between time efficiency *vs* memory efficiency *vs* precision



Eva Parameterization

- Eva is automatic
- Eva requires fine-tuned parameterization to be both precise and efficient
  - lots of parameters, but not so many almost always useful
  - try `-eva-precision n` ($0 \leq n \leq 11$)
- trade-off between time efficiency *vs* memory efficiency *vs* precision
- stubbing: `main` function and missing library function
  - either provide C code or ACSL specification (often, `assigns` clauses are enough)
  - similar to stubbing required by testing
  - Frama-C's libc provides many stubs



Much more during the

practical session

Eva Applications

Scada System

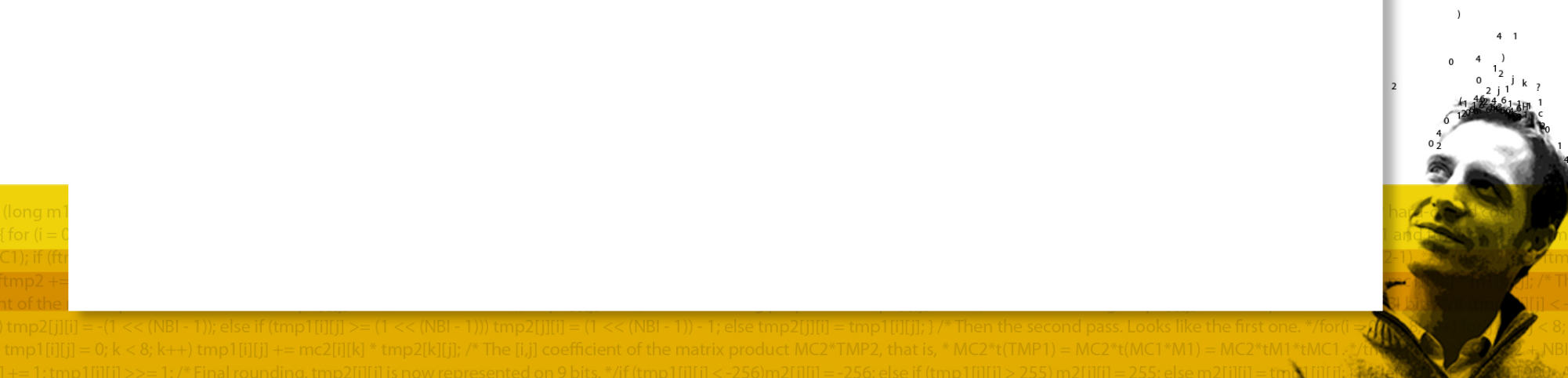
can we guarantee absence of defaults in large system-level code?

- scada systems of 100+ kloc of C code
- highest certification requirements (IEC60880 class 1)
- pinpoint the undefined behaviors and help investigate their cause
- structural properties on memory separation and cyclic behaviors
- 80% code coverage, $<$ 100 alarms
- [Ourghanlian, 2015]

frama C
Software Analyzers

Part IV

Runtime Verification with E-ACSL



Runtime Verification in a Nutshell

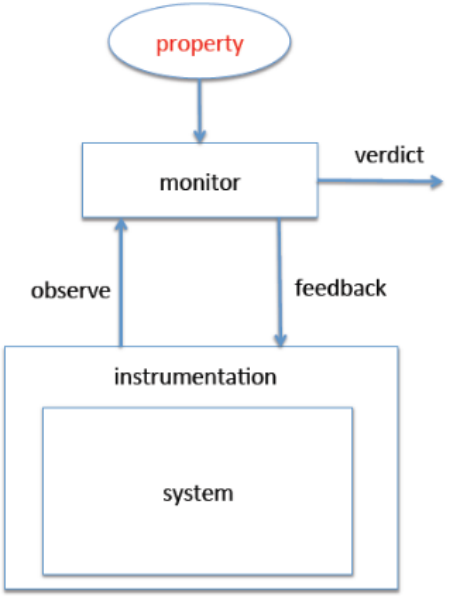
[Falcone et al, 2013]

- check whether a system under scrutiny satisfies a given property
- offline monitoring: after execution
- online monitoring: during execution
  - inline: the monitor is part of the system
  - outline: the monitor is outside the system
- most approaches focus on verifying temporal properties

Runtime Verification in a Nutshell

[Falcone et al, 2013]

- check whether a system under scrutiny satisfies a given property
- offline monitoring: after execution
- online monitoring: during execution
  - inline: the monitor is part of the system
  - outline: the monitor is outside the system
- most approaches focus on verifying temporal properties
- ... yet not E-ACSL



What is E-ACSL?

- E-ACSL is a a subset of ACSL [Delahaye et al, 2013]
- E-ACSL is a Frama-C plug-in that generates inline monitors for (E-)ACSL properties on C programs [Signoles et al, 2017]
  - takes as input an (E-)ACSL-annotated C program
  - generates a new C program
  - that behaves as the original C program if all the annotations are valid; or
  - fails on the first invalid annotation (by default)

```
int div(int x, int y) {
  /*@ assert y-1 != 0; */
  return x / (y-1);
}
```

E-ACSL →

```
int div(int x, int y) {
  /*@ assert y-1 != 0; */
  e_acsl_assert(y-1 != 0L);
  return x / (y-1);
}
```

E-ACSL Specification Language

[Delahaye et al, 2013]

- ACSL was designed for program proving tools only
  - based on logic and mathematics
  - cannot execute any term/predicate (e.g. unbounded quantification)
  - cannot be used by dynamic analysis tools
- E-ACSL was introduced to circumvent this issue
  - a few restrictions (e.g. syntactically-bounded quantifications)
  - one semantic change: 3-valued logic with undefinedness

ACSL vs E-ACSL Semantics

The Undefinedness Problem

What is the semantics of `1/0 == 1/0`? [Cheon, 2003]

ACSL vs E-ACSL Semantics

The Undefinedness Problem

What is the semantics of `1/0 == 1/0`? [Cheon, 2003]

- ACSL semantics is a 2-valued logic with only total functions
  - no term is undefined, even `1/0` and `*p` when `p` is null
  - predicates such as `1/0 == 1/0` are valid
  - easier to prove properties with automatic theorem provers

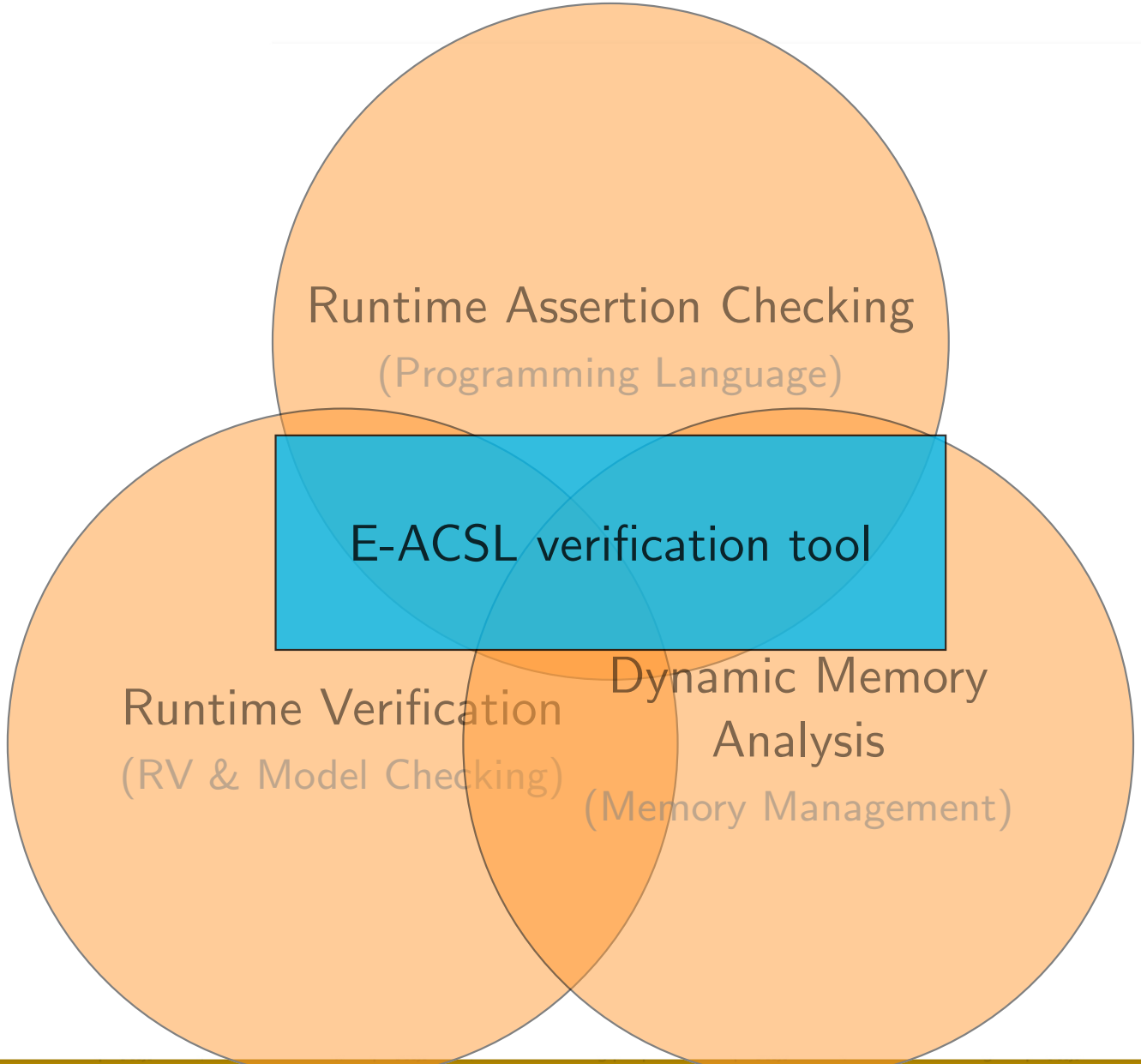
ACSL vs E-ACSL Semantics

The Undefinedness Problem

What is the semantics of `1/0 == 1/0`? [Cheon, 2003]

- ACSL semantics is a 2-valued logic with only total functions
  - no term is undefined, even `1/0` and `*p` when `p` is null
  - predicates such as `1/0 == 1/0` are valid
  - easier to prove properties with automatic theorem provers
- E-ACSL semantics is a 3-valued logic with undefined values
  - Chalin's Runtime Assertion Checking semantics [Chalin, 2009]
  - Follows the strong validity principle [Konikowska et al, 1991]
  - the semantics of 1/0 is "undefined"
  - a predicate is valid if it is both defined and true
  - at runtime, usual programming-language semantics
    - executing undefined terms is undefined behaviors

E-ACSL's Scientific Fields

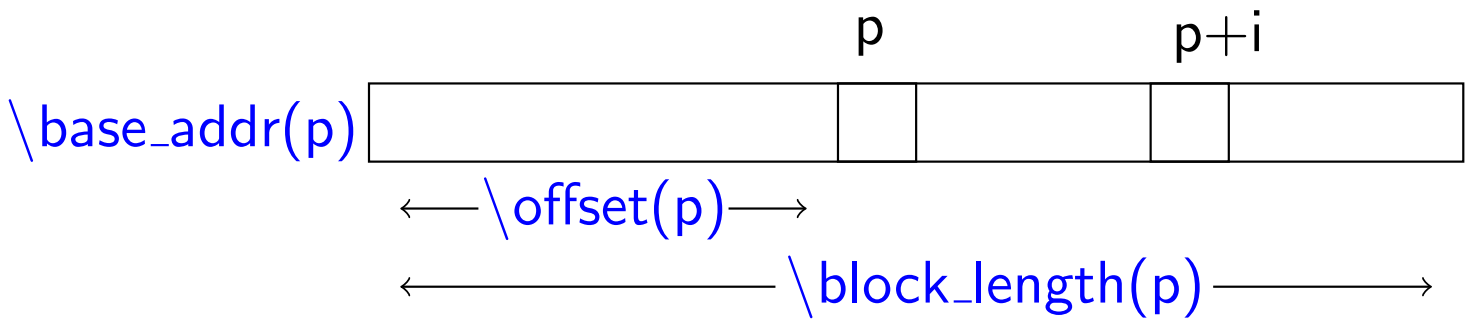


Memory Monitoring

how to monitor memory-related properties, e.g. `\valid(p+i)`?

Memory Monitoring

how to monitor memory-related properties, e.g. `\valid(p+i)`?



- block-level memory properties

```
char buf1[1], buf2[1];
/*@ assert \valid(buf1 + 1); */ // must fail
buf1[1] = 'a';
```

Dynamic Memory Analyzers

- fat pointers [Austin et al, 1994]
  - extend the pointer representation with bounds information.
  - expressive, but hard to preserve the original program behavior

Dynamic Memory Analyzers

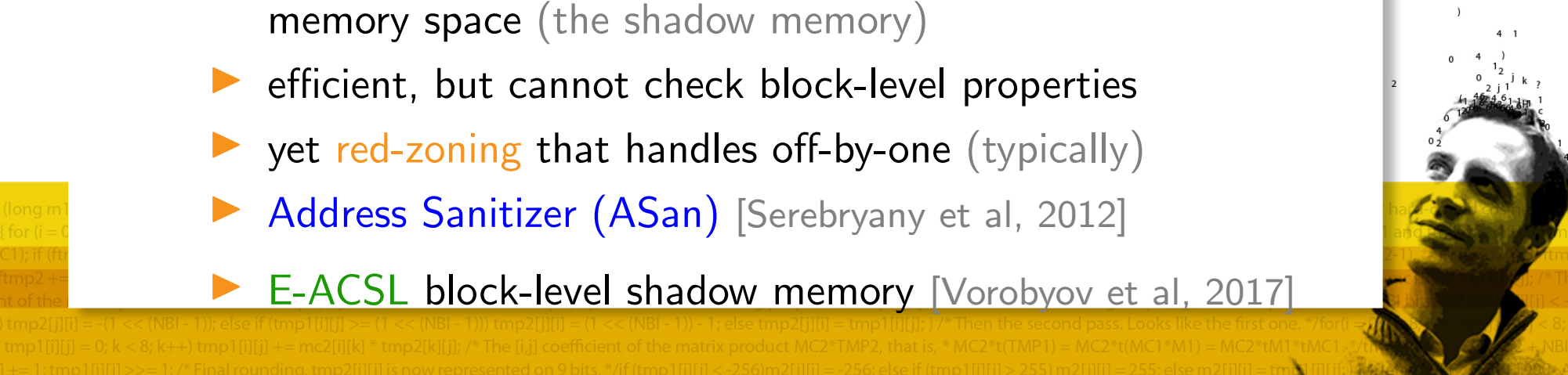
- fat pointers [Austin et al, 1994]
  - extend the pointer representation with bounds information.
  - expressive, but hard to preserve the original program behavior
- dictionary-based models [Jones & Kelly, 1997]
  - associate any data to memory locations in a dictionary
  - expressive, but slow: scalability issue

Dynamic Memory Analyzers

- fat pointers [Austin et al, 1994]
  - extend the pointer representation with bounds information.
  - expressive, but hard to preserve the original program behavior
- dictionary-based models [Jones & Kelly, 1997]
  - associate any data to memory locations in a dictionary
  - expressive, but slow: scalability issue
- shadow memory
  - associate a few data to memory locations in a disjoint memory space (the shadow memory)
  - efficient, but cannot check block-level properties
  - yet red-zoning that handles off-by-one (typically)
  - Address Sanitizer (ASan) [Serebryany et al, 2012]

Dynamic Memory Analyzers

- fat pointers [Austin et al, 1994]
  - extend the pointer representation with bounds information.
  - expressive, but hard to preserve the original program behavior
- dictionary-based models [Jones & Kelly, 1997]
  - associate any data to memory locations in a dictionary
  - expressive, but slow: scalability issue
- shadow memory
  - associate a few data to memory locations in a disjoint memory space (the shadow memory)
  - efficient, but cannot check block-level properties
  - yet red-zoning that handles off-by-one (typically)
  - Address Sanitizer (ASan) [Serebryany et al, 2012]
  - E-ACSL block-level shadow memory [Vorobyov et al, 2017]



E-ACSL Expressiveness

[Vorobyov et al, 2018]

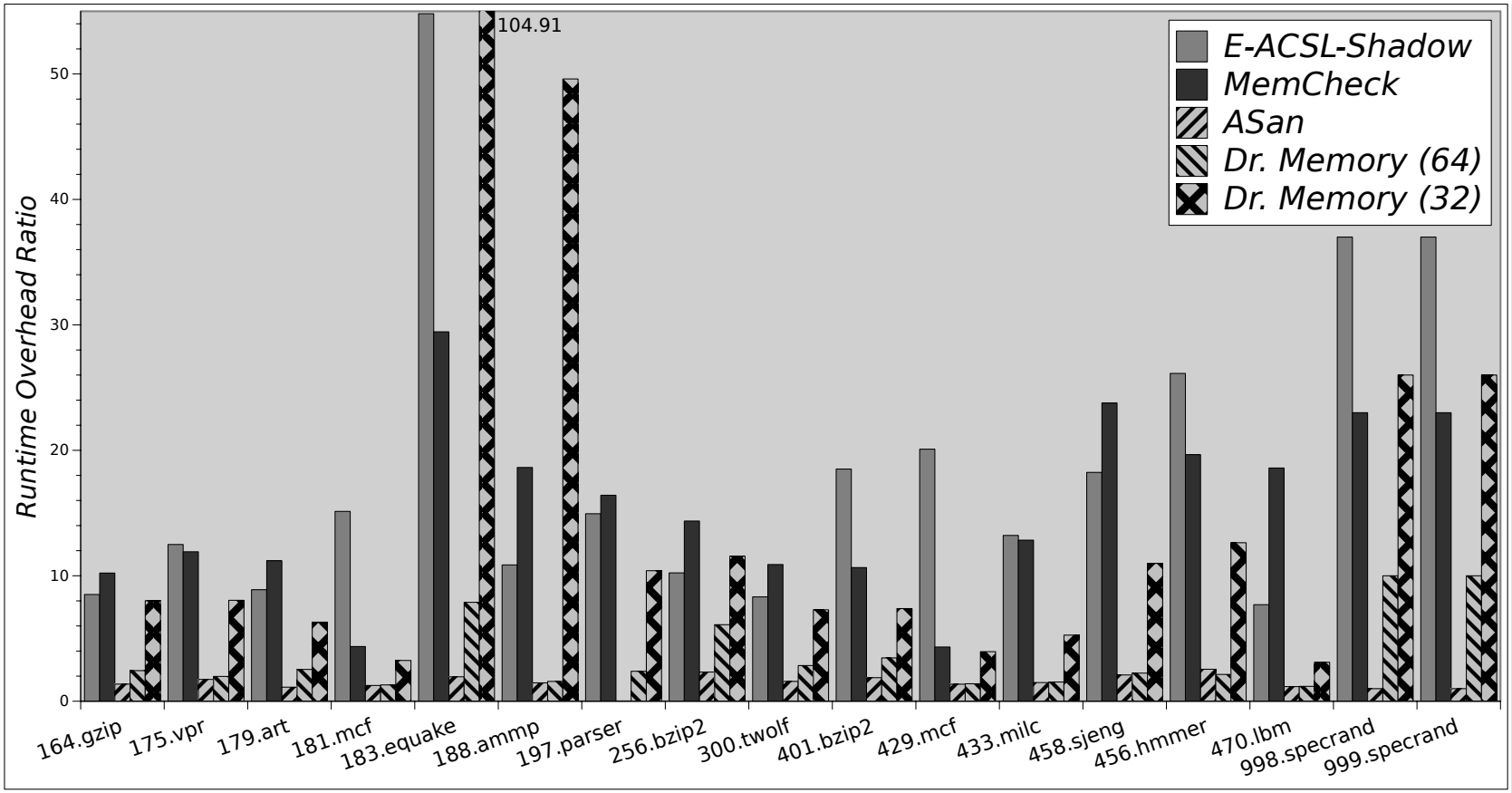
Defect Type	E-ACSL	Google's Sanitizers in Clang
Dynamic Memory	94% (81/86)	78% (67/86)
Static Memory	✓ (67/67)	96% (64/67)
Pointer-related	56% (47/84)	32% (27/84)
Stack-related	35% (7/20)	70% (14/20)
Resource	99% (95/96)	60% (58/96)
Numeric	93% (100/108)	59% (64/108)
Miscellaneous	94% (33/35)	49% (17/35)
Inappropriate Code	– (0/64)	– (0/64)
Concurrency	– (0/44)	73% (32/44)
Overall	71% (430/604)	57% (343/604)

Detection Capabilities over Toyota ITC Benchmark:

more expressive than the mainstream tools

E-ACSL Performance

[Vorobyov et al, 2017]



×17 time-overhead; ×2.4 memory overhead on SPEC-CPU

comparable to Valgrind; still slower than AddressSanitizer

less memory-overhead than these tools

Part V

Combining Analyses for Security-Oriented Verifications

Combining E-ACSL, Eva, and WP

Security-Oriented Applications

- no verification technique is perfect
  - WP is not always fully automatic
  - Eva requires fine-tuning and cannot discharge all alarms
  - E-ACSL does not check all possible inputs

Combining E-ACSL, Eva, and WP

Security-Oriented Applications

- no verification technique is perfect
  - WP is not always fully automatic
  - Eva requires fine-tuning and cannot discharge all alarms
  - E-ACSL does not check all possible inputs
- combining them can be helpful for large verification tasksr
  - Dassault Aviation combines Eva and E-ACSL for deploying security counter-measures on OpenSSL/Apache modules [Pariente & Signoles, 2017]
  - ANSSI combines Eva and WP for checking that a parser is bug-free [Ebalard et al, 2019]
  - Contiki modules were verified by ruling them all [Blanchard et al, 2018]
    Contiki is an OS for IoT systems [Dunkels et al, 2004]

Non-Interference Properties

[Goguen and Meseguer, 1982]

- confidentiality, e.g. no confidential data shall leak to a public channelé
- integrity, e.g. no untrusted input shall be propagated into an internal storage

Non-Interference Properties

[Goguen and Meseguer, 1982]

- confidentiality, e.g. no confidential data shall leak to a public channelé
- integrity, e.g. no untrusted input shall be propagated into an internal storage
- direct or explicit flows through direct assignments

```
lo = hi; // the secret data 'hi' directly flows to
         // the public data 'lo'
```

- indirect or implicit flows through control flows

```
if (hi) lo = 0; // the secret indirectly flows
```

- hidden channels
  - (non-)termination
  - execution time
  - ...

Information Flow Analysis

[Hedin and Sabelfeld, 2012]

how to check information flow properties?

- static analyses [Denning and Denning, 1977]
  - many type-based approaches [Volpano et al, 1996]
  - can be automatic and sound
  - practical issue: may reject programs with no flaws
  - possible workaround: declassification but often heavy in practice

Information Flow Analysis

[Hedin and Sabelfeld, 2012]

how to check information flow properties?

- static analyses [Denning and Denning, 1977]
  - many type-based approaches [Volpano et al, 1996]
  - can be automatic and sound
  - practical issue: may reject programs with no flaws
  - possible workaround: declassification but often heavy in practice
- dynamic analyses [Vachharajani et al, 2004]
  - more precise than static approaches
  - necessarily unsound for flow-sensitive non-interference: cannot detect all incorrect flows [Russo and Sabelfeld, 2010]

Information Flow Analysis

[Hedin and Sabelfeld, 2012]

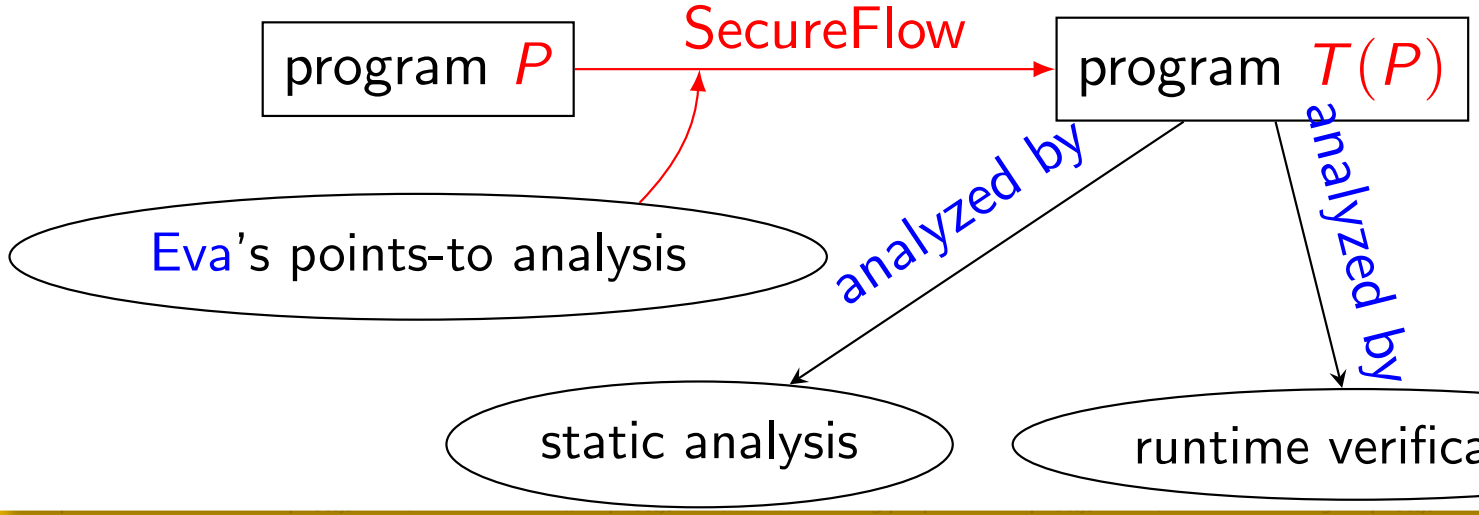
how to check information flow properties?

- static analyses [Denning and Denning, 1977]
  - many type-based approaches [Volpano et al, 1996]
  - can be automatic and sound
  - practical issue: may reject programs with no flaws
  - possible workaround: declassification but often heavy in practice
- dynamic analyses [Vachharajani et al, 2004]
  - more precise than static approaches
  - necessarily unsound for flow-sensitive non-interference: cannot detect all incorrect flows [Russo and Sabelfeld, 2010]
- hybrid analyses [Le Guernic et al, 2006]
  - mix static and dynamic approaches
  - try to be precise while removing unsoundness

SecureFlow

(close source) Frama-C Plug-in for Information Flow Tracking

- (Termination-Insensitive) Non-Interference (TINI) properties
- hyprid approach: generates inline monitors based on static analysis for soundness in presence of pointers and aliasing [Assaf et al, 2013]
- verify the generated monitors with standard techniques
  - Eva
  - E-ACSL



SecureFlow for Time Channel Detection

[Barany & Signoles, 2017]

- may also detect some time-channel attacks
  - by checking flows of branch conditions
- experimented on LibTomCrypt (60,000-line C crypto library)
  - prove that all the 14 symmetric cryptosystems as secure
  - detect one known vulnerability in 1 asymmetric cryptosystem
    - a loop condition leaks one bit of a secret key each time it is checked
    - eventually, all bits are leaked
  - yet, detect that the patch is not fully secure

MetACSL: Expressing High-Level Properties

[Robles et al, 2019]

- ACSL is a quite low-level specification language
  - difficult to express high-level properties
  - e.g. security policies
- MetACSL introduces a higher-level specification language
- MetACSL automatically converts specifications written in this language to sequences of ACSL annotations
- verify the generated annotations with standard techniques
  - WP
  - E-ACSL

MetACSL: Example

Confidentiality-oriented page management system

- modeling a security policy of a confidentiality-oriented page management system
  - MetACSL allows to model one rule at a time

MetACSL: Example

Confidentiality-oriented page management system

- modeling a security policy of a confidentiality-oriented page management system
  - MetACSL allows to model one rule at a time

```
/*@ meta \macro, \name(\forall_page), \arg_nb(2), ... */

// Never write to a lower confidentiality page
// outside of free
/*@ meta \prop,
  \name(confidential_write),
    \targets(\diff(\ALL, page_free, init)),
    \context(\writing),
      \forall_page(p,
        p->status == PAGE_ALLOCATED
        && user_level > page_level(p)
        ==> \separated(\written, page_data(p))
      ); */
```

Impact: Frama-C Plug-in for Impact Analysis

[Monate & Signoles, 2008]

- the Impact plug-in computes the "impact" of a set S of statements: help security audits
- i.e. the statements whose evaluation depend on S
  - data dependency (result of computation is impacted)

```
x = a + b; // dependencies to the definitions of a and b
```

  - address dependency (memory location is impacted)

```
*p = x; // *p depends on x (data dependency);
        // but the memory location of *p depends on p
```

  - control dependency (branch may be taken or not)

```
if (c) x = a; // x at L depends on c
L:
```

Impact: Frama-C Plug-in for Impact Analysis

[Monate & Signoles, 2008]

- the Impact plug-in computes the "impact" of a set S of statements: help security audits
- i.e. the statements whose evaluation depend on S
  - data dependency (result of computation is impacted)

```
x = a + b; // dependencies to the definitions of a and b
```

  - address dependency (memory location is impacted)

```
*p = x; // *p depends on x (data dependency);
        // but the memory location of *p depends on p
```

  - control dependency (branch may be taken or not)

```
if (c) x = a; // x at L depends on c
L:
```

- exploit the Frama-C's Program Dependence Graph (PDG)
  - make explicit all the program dependencies [Ottenstein and Ottenstein, 1984]
  - Frama-C's PDG relies on Eva for infering aliasing information

Conclusion

Security Verification on C Code

- Easy, isn't it?
  - Code security is hard
  - "C is quirkly, flawed and an enormous success" [Ritchie, 1993]
  - Combining both is sometimes wtf, but always challenging :-)

Conclusion

Security Verification on C Code

- Easy, isn't it?
  - Code security is hard
  - "C is quirkly, flawed and an enormous success" [Ritchie, 1993]
  - Combining both is sometimes wtf, but always challenging :-)
- Absence of undefined behaviors is security-critical for C code
- Yet it is not the only desirable property
  - non-interference properties
  - correct implementation of security policies
  - constant-time implementation (Pichardie's lecture)

Conclusion

Security Verification on C Code

- Easy, isn't it?
  - Code security is hard
  - "C is quirkly, flawed and an enormous success" [Ritchie, 1993]
  - Combining both is sometimes wtf, but always challenging :-)
- Absence of undefined behaviors is security-critical for C code
- Yet it is not the only desirable property
  - non-interference properties
  - correct implementation of security policies
  - constant-time implementation (Pichardie's lecture)
  - access control enforcement? Boukir's just-started postdoc
  - privacy properties? Clouet's ongoing PhD
  - ...

Conclusion

Frama-C for Code Security

- Frama-C provides industrial-strengh verification tools
  - WP for deductive verification
  - Eva for abstract interpretation
  - E-ACSL for runtime verification
  - possible to combine them in many ways

Conclusion

Frama-C for Code Security

- Frama-C provides industrial-strengh verification tools
  - WP for deductive verification
  - Eva for abstract interpretation
  - E-ACSL for runtime verification
  - possible to combine them in many ways
- Frama-C provides many other plug-ins, including
  - SecureFlow for information flow analysis
  - MetACSL for expressing high-level properties
  - Impact for helping security audits

Conclusion

Frama-C for Code Security

- Frama-C provides industrial-strengh verification tools
  - WP for deductive verification
  - Eva for abstract interpretation
  - E-ACSL for runtime verification
  - possible to combine them in many ways
- Frama-C provides many other plug-ins, including
  - SecureFlow for information flow analysis
  - MetACSL for expressing high-level properties
  - Impact for helping security audits
- Frama-C is open-source and extensible
  - feel free to develop your own plug-ins!

Bibliography

By order of appearance

1. F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski
   Frama-C: a Software Analysis Perspective
   In Journal of Formal Aspects of Computing, 2015

2. J. Signoles
   Software Architecture of Code Analysis Frameworks Matters: The Frama-C Example
   In Int. Workshop on Formal Integrated Development Environment (F-IDE), 2015

3. L. Correnson, and J. Signoles
   Combining Analyses for C Program Verification
   In Int. Workshop on Formal Methods for Industrial Case Studies (FMICS), 2012

4. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer
   CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs
   In Int. Conf. on Compiler Construction (CC), 2002

5. E. W. Dijkstra
   Guarded commands, nondeterminacy, and formal derivation of programs
   In Communications of the ACM, 1975

6. Z. Xu, T. Kremenek, and J. Zhang
   A Memory Model for Static Analysis of C Programs
   In Int. Symp. on Leveraging Applications of Formal Methods, Verification, and Validation (ISoLA), 2010

7. J.-H. Jourdan
   A formally-verified C static analyzer
   In PhD Thesis, Université Paris Diderot, 2016

Bibliography

Cont'd

8. X. Leroy
Formal verification of a realistic compiler
In Communications of the ACM, 2009

9. A. Atki, A. Brahmi, D. Delmas, H. Essoussi, T. Marie, and F. Randimbivololona
Formalise to automate: deployment of a safe and cost-efficient process for avionics software
In Eur. Conf. on Embedded Real Time Software and Systems (ERTSS), 2018

10. P. Cousot, and R. Cousot
Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints
In Int. Conf. on Principles of Programming Languages (POPL), 1977

11. S. Blazy, D. Bühler, and B. Yakobowski
Structuring Abstract Interpreters through State and Value Abstractions
In Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI), 2017

12. B. Jeannet, and A. Miné
Apron: A Library of Numerical Abstract Domains for Static Analysis
In Int. Conf. on Computer-Aided Verification (CAV), 2009

13. A. J. Venet
The gauge domain: scalable analysis of linear inequality invariants
In Int. Conf. on Computer-Aided Verification (CAV), 2012

14. S. Blazy, D. Bühler, and B. Yakobowski
Improving static analyses of C programs with conditional predicates
In Journal of Science of Computer Programming, 2016

Bibliography

Cont'd

15. M. Jacquemin, S. Putot, and F. Védrine
A Reduced Product of Absolute and Relative Error Bounds for Floating-Point Analysis
In Static Analysis Symposium (SAS), 2018

16. A. Ourghanlian
Evaluation of static analysis tools used to assess software important to nuclear power plant safety
In Journal of Nuclear Engineering and Technology, 2015

17. Y. Falcone, K. Havelund, and G. Reger
A Tutorial on Runtime Verification
In Engineering Dependable Software Systems, 2015

18. M. Delahaye, N. Kosmatov, and J. Signoles
Common Specification Language for Static and Dynamic Analysis of C Programs
In Symposium on Applied Computing (SAC), 2013

19. J. Signoles, N. Kosmatov, and K. Vorobyov
E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs
In Int. Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES), 2017

20. Y. Cheon
A runtime assertion checker for the Java Modeling Language
In PhD Thesis, Iowa State University, 2003

21. P. Chalin
Engineering a Sound Assertion Semantics for the Verifying Compiler
In Transactions on Software Engineering, 2009

Bibliography

Cont'd

22. B. Konikowska, A. Tarlecki, and A. Blikle
A three-valued logic for software specification and validation
In Fundamenta Informaticae, 1991

23. T. M. Austin, S. E. Breach, and G. S. Sohi
Efficient Detection of All Pointer and Array Access Errors
In Int. Conf. on Programming Language Design and Implementation (PLDI), 1994

24. R. W. M. Jones, and P. H. J. Kelly
Backwards-compatible bounds checking for arrays and pointers in C programs
In Int. Workshop on Automatic Debugging (AADEBUG), 1997

25. K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov
AddressSanitizer: A Fast Address Sanity Checker
In USENIX Annual Technical Conference (USENIX), 2012

26. K. Vorobyov, J. Signoles, and N. Kosmatov
Shadow State Encoding for Efficient Monitoring of Block-level Properties
In Int. Symposium on Memory Management (ISMM), 2017

27. K. Vorobyov, N. Kosmatov, and J. Signoles
Detection of Security Vulnerabilities in C Code using Runtime Verification
In Int. Conf. on Tests and Proofs (TAP), 2018

28. D. Pariente, and J. Signoles
Static Analysis and Runtime Assertion Checking: Contribution to Security Counter-Measures
In Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC), 2017

Bibliography

Cont'd

29. A. Ebalard, P. Mouy, and R. Benadjila
Journey to a RTE-free X.509 parser
In Symposium sur la Sécurité des Systèmes de l'Information et des Communications (SSTIC), 2019

30. A. Blanchard, N. Kosmatov, and F. Loulergue
A Lesson on Verification of IoT Software with Frama-C
In Int. Conf. on High Performance Computing & Simulation (HPCS), 2018

31. A. Dunkels, B. Gronvall, and T. Voigt
Contiki – a lightweight and flexible operating system for tiny networked sensors
In Int. Conf. on Local Computer Networks (LCN), 2004

32. J. A. Goguen, and J. Meseguer
Security Policies and Security Models
In Int. Symp. on Security and Privacy (S&P), 1982

33. D. Hedin and A. Sabelfeld
A perspective on information-flow control
In Software safety and security, 2012,

34. D. E. R. Denning and D. J. Denning
Certification of programs for secure information flow
In Communications of the ACM, 1977

35. D. Volpano, C. Irvine, and G. Smith
A sound type system for secure flow analysis
In Journal of Computer Security, 1996

Bibliography

Cont'd

36. N. Vachharajani, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, D. I. August
Rifle: An architectural framework for user-centric information-flow security
In Int. Symp. on Microarchitecture (MICRO), 2004

37. G. Le Guernic, A. Banerjee, T. P. Jensen, and D. A. Schmidt
Automata-based confidentiality monitoring
In Asian Computing Science Conf. (ASIAN), 2006

38. A. Russo, and A. Sabelfeld
Dynamic vs. Static Flow-Sensitive Security Analysis
In Int. Conf. on Computer Security Foundation (CSF), 2010

39. M. Assaf, J. Signoles, É. Totel, and F. Tronel
Program transformation for non-interference verification on programs with pointers
In IFIP Int. Information Security and Privacy Conf. (IFIP-SEC), 2013

40. G. Barany, and J. Signoles
Hybrid Information Flow Analysis for Real-World C Code
In Int. Conf. on Tests and Proofs (TAP), 2017

41. V. Robles, N. Kosmatov, V. Prevosto, L. Rilling, and P. Le Gall
MetAcsl: Specification and Verification of High-Level Properties
In Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2019

42. B. Monate, and J. Signoles
Slicing for Security of Code
In Int. Conf. on Trusted Computing and Trust in Information Technologies (TRUST), 2008

Bibliography

Cont'd

43. K. J. Ottenstein and L. M. Ottenstein
The program dependence graph in a software development environment
In software engineering symposium on practical Software Development Environments (SDE), 1984

44. D. M. Ritchie
The Development of the C Language
In ACM SIGPLAN Notices, 1993