Frama-C for Cybersecurity
A Few Case Studies

Julien Signoles

Software Safety & Security Lab

GLSEC 2022

Nov. 24, 2022

Outline

1. Frama-C at a Glance
2. Main Verification Tools
   - Eva
   - Wp
   - E-ACSL
3. Advanced Security Verifications

Part I

Frama-C at a Glance

Framework for analyses of source code written in ISO 99 C

[Baudin & al, 2021]

- developed by CEA LIST since 2005
- comes with a formal specification language: ACSL
- targets both academic and industrial usage
- almost open source (LGPL 2.1)
- first open-source release (1-Hydrogen) in 2008
- last open-source release (26-Iron): yesterday!

http://frama-c.com

- also non-CEA extensions and distributions
- targets both academic and industrial usages

Several tools inside a single platform

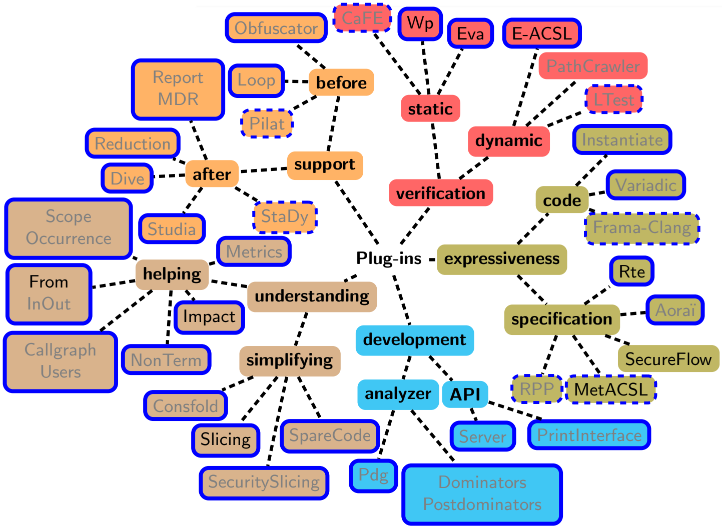
- plug-in architecture *à la* Eclipse [Signoles, 2015]
- tools provided as plug-ins
  - 32 plug-ins in the latest open source distribution
  - outside open source plug-ins
  - close source plug-ins, either at CEA (> 20) or outside

Several tools inside a single platform

- plug-in architecture *à la* Eclipse [Signoles, 2015]
- tools provided as plug-ins
  - 32 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-in Gallery



distributed within Frama-C

external and open source

close source

Frama-C

Development Platform

- developed in OCaml
- library dedicated to analysis of C code

development of plug-ins by third party

- powerful low-cost analyser
- dedicated plug-in for specific task (e.g., coding rule verifier)
- dedicated plug-in for fine-grain parameterization
- extension of existing analysers

Part II

Main Verification Tools

Domain of variations of variables of the program

- abstract interpretation
- automatic analysis
- correct over-approximation
- alarms for potential invalid operations
- alarms for potential invalid ACSL annotations
- ensures the absence of runtime error
- graphical interface: display the domain of each variable at each program point
- [Blazy et al, 2017]

Parameterization

- Eva is automatic
- but requires fine-tuned parameterization to be precise/efficient
- trade-off between time efficiency *vs* memory efficiency *vs* precision
- stubbing: `main` function and missing library function
  - either provide C code or ACSL specification (usually, `assigns`)
  - similar to stubbing required by testing
- 100+ parameters
  - require expertise
  - try `-eva-precision n` first ($0 \leq n \leq 11$)

Demo

PolarSSL

(now known as Mbed-TLS)

https://git.trustedfirmware.org/mirror/mbed-tls.git/about/

- C implementation of TLS (aka SSL)
- not as complex as openSSL

Dependency Analysis

- tied to Eva
- for each memory location `loc` possibly modified, returns its dependencies
- i.e. the set of locations whose value might be used in computing the final value of `loc`
- over-approximation: some dependencies might be spurious
- may help security audits

Example

Keccak (SHA-3)

```
> frama-c -eva -eva-slevel 1000 -deps \
          Keccak-simple.c KeccakNISTInterface.c \
          KeccakSponge.c KeccakF-1600-reference.c test.c

[from] Function rho:
  context.state[0..199]
         FROM context.state[0..199];
         KeccakRhoOffsets[0..24]; A (and SELF)
[from] Function theta:
  context.state[0..199] FROM
      context.state[0..199]; A (and SELF)
[from] Function KeccakPermutationOnWords:
  context.state[0..199] FROM
          context.state[0..199];
          KeccakRoundConstants[0..23];
          KeccakRhoOffsets[0..24]; state (and SELF)
```

Impact

- computes impact of a set S of statements
  [Monate & Signoles, 2008]
- i.e. the statements whose evaluation depend on S
  - data dependency (whether it results from a computation)
    - `x = n; y = x;`
  - address dependency (whether its memory location is impacted)
    - `p = q; *p = 0;`
  - control dependency (whether a branch may be taken)
    - `if (c) x = n; y = x;`
- exploit the 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
- may help security audits

Slicing

- removes all statements that do not change some slicing criterion
- slicing criterion
  - value of a variable at a given point
  - truth value of an ACSL assertion
  - final state of the program
- same dependencies as impact, but used in the opposite direction (dual analysis)
- may make other analyses more tractable
- may help security audits

Wp

- based on Dijkstra's weakest preconditon calculus
- generates theorems (proof obligations) to ensure that a code satisfies its ACSL specification
- uses automatic/interactive theorem provers to verify these theorems
  - rely on Why3 as back-end
  - use Alt-Ergo by default
- is able to verify complex specifications
- modular verification
  - prove each function independently from each other
  - require no stubbing
- requires to manually add extra annotations (e.g. loop invariants)

Wp

Student Example

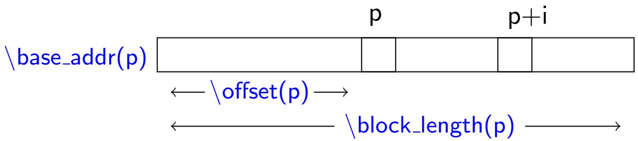
```
/*@ predicate sorted{L}(int* a, int length) =
    \forall integer i,j; 0<=i<=j<length ==> a[i]<=a[j]; */

/*@ requires \valid(a+(0..length-1));
    requires sorted(a,length);
    requires length >=0;

    assigns \nothing;
    behavior exists:
      assumes \exists integer i; 0 <= i < length && a[i] == key;
      ensures 0<=\result<length && a[\result] == key;
    behavior not_exists:
      assumes \forall integer i; 0<=i<length ==> a[i] != key;
      ensures \result == -1;
    complete behaviors;
    disjoint behaviors; */
int binary_search(int* a, int length, int key);
```

non-security oriented!

ACSL and Memory Properties

- memory properties are important for code security
- ACSL provides built-ins memory-related predicates and functions
  - `\valid(p)`: whether $*p$ has 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
  - 

  - ...

WP's Use Cases

- X509 parser developped by ANSSI
  - https://github.com/ANSSI-FR/x509-parser
- Wookey, secure storage device developped by ANSSI
  - https://github.com/wookey-project
  - [Benadjila et al, 2019]
- proved RTE-free by ANSSI
  - functional correctness also proved
  - combined Eva and Wp
  - X509: [Ebalard et al, 2019]
  - Wookey: [Benadjila et al, 2021]

verification of ACSL properties at runtime

- generates inline monitors for ACSL properties
  - takes as input an 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)
  - [Signoles et al, 2017]

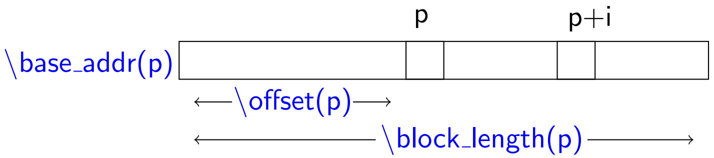
```
int div(int x, int y) {
  /*@ assert y-1 != 0; */
  return x / (y-1);
}
```

$\xrightarrow{\text{E-ACSL}}$

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

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

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';
```

E-ACSL Expressiveness

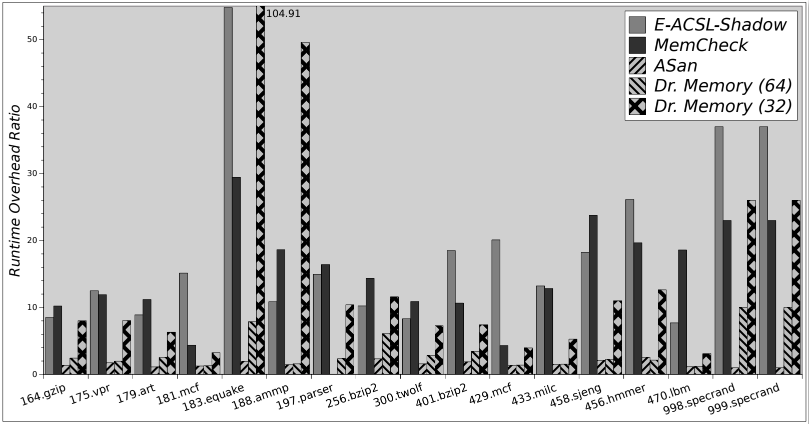
[Vorobyov et al, 2018]

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 mainstream tools

E-ACSL Performance

[Vorobyov et al, 2017]



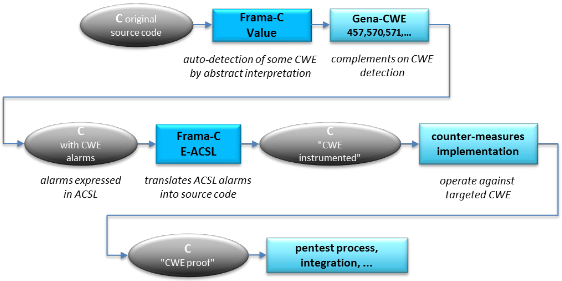
×17 time-overhead; ×2.4 memory overhead on SPEC-CPU

comparable to Valgrind; still slower than AddressSanitizer

less memory-overhead than these tools

E-ACSL

Dassault Aviation's Use Case


C original source code
Frama-C Value
Gena-CWE 457,570,571,...
auto-detection of some CWE by abstract interpretation
complements on CWE detection
C with CWE alarms
Frama-C E-ACSL
C "CWE instrumented"
counter-measures implementation
alarms expressed in ACSL
translates ACSL alarms into source code
operate against targeted CWE
C "CWE proof"
pentest process, integration, ...

first, use automatic static analysis to detect vulnerabilities;
then, switch to fast runtime monitoring

Experimented on modules from Apache / OpenSSL
[Pariente & Signoles, 2017]

Part III

Advanced Security Verifications

Test Inversion and Countermeasures

```
if (password != secret)   return 1;
if ! (password == secret) return 1;
```

- countermeasures: redundancy checks for critical sections
  - repeat critical checks at least $k+1$ time each, assuming the attacker can invert up to k tests
- prove correctness of redundant-check countermeasures
  - rely on mutation testing
- implemented in Frama-C/LTest
  - LTest is a suite of Frama-C plug-ins providing test coverage metrics
- successfully applied on Wookey
  - 1 incorrect countermeasure found
  - proved after fixing
  - [Martin et al, 2022]

MetACSL: System-Level Properties

[Robles et al, 2019]

- ACSL is a quite low-level specification language
  - difficult to express system-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 in Practice

- example from an OS microkernel's specification:

```
/*@ 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))
      ); */
```

- MetaCSL used for specificying and verifiying with WP the Wookey's bootloader module [Robles et al, 2021]

Common Criteria Certification

[Djoudi et al, 2021]

- Formal verification of a JavaCard Virtual Machine
- Common Criteria's EAL7 certificate
- Example of properties
  - header integrity
    - allocated object's header cannot be modified during a run
  - data integrity
    - allocated object's data can be modified only by the owner
  - data confidentiality
    - allocated object's data can be read only by the owner
- generate $\approx 400,000$ ACSL annotations from ≈ 500 MetACSL properties
  - all proved with Wp

THALES

Conclusion and Perspectives

- Frama-C provides scalable analyzers for C code verification
  - Eva: proving absence of undefined behaviors
  - Wp: proving functional properties
  - E-ACSL: checking properties at runtime
- possible to check advanced security properties
  - correctness of redundancy checks
  - system-level properties
  - but also (not shown here):
    - information flow properties [Barany and Signoles, 2017]
    - relational properties [Blatter et al, 2022]
    - privacy properties (Clouet's talk this afternoon)
    - taint analysis (ongoing work)
    - type-state analysis (ongoing work)
    - access-control policies (ongoing work)
    - ...
- usable for real-world applications
  - EAL7 certification

Software Analyzers

Bibliography

By order of appearance

1. P. Baudin, F. Bobot, D. Bühler, L. Correnson, F. Kirchner, N. Kosmatov, A. Maroneze, V. Perrelle, V. Prevosto, J. Signoles, and N. Williams
   The Dogged Pursuit of Bug-Free C Programs: The Frama-C Software Analysis Platform
   In Communications of the ACM, 2021

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. 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

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

6. K. J. Ottenstein and L. M. Ottenstein
   The program dependence graph in a software development environment
   In Software Engineering Symp. on Practical Software Development Environments (SDE), 1984

7. R. Benadjila, A. Michelizza, M. Renard, P. Thierry, and P. Trebuchet
   WooKey: Designing a Trusted and Efficient USB Device
   In Annual Computer Security Applications Conf. (ACSAC), 2019

Bibliography
Cont'd

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

9. R. Benadjila, C. Debergé, P. Mouy, and P. Thierry
From CVEs to proof: Make your USB device stack great again
In Symp. sur la Sécurité des Systèmes de l'Information et des Communications (SSTIC), 2021

10. 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

11. 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

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

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

14. T. Martin, N. Kosmatov, and V. Prevosto
Verifying Redundant-Check Based Countermeasures: A Case Study
In Int. Symp. on Applied Computed (SAC), 2022

Bibliography
Cont'd

15. 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

16. V. Robles, N. Kosmatov, V. Prevosto, L. Rilling, and P. Le Gall
Methodology for Specification and Verification of High-Level Requirements with MetAcsl
In Int. Conf. on Formal Methods in Software Engineering (FormaliSE), 2021

17. A. Djoudi, M. Hana, and N. Kosmatov
Formal Verification of a JavaCard Virtual Machine with Frama-C
In Int. Conf. on Formal Methods (FM), 2021

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

19. L. Blatter, N. Kosmatov, V. Prevosto, and P. Le Gall
Certified Verification of Relational Properties
In Int. Conf. on Integrated Formal Methods (IFM), 2022