Runtime Verification of Security Properties with E-ACSL

Julien Signoles

Software Reliability and Security Lab

Sound Static Analysis for Security Workshop

2018/6/28

Context: Frama-C

Frama-C

- opensource framework for analyzing code written in ISO C99
- used in many critical domains (avionic, energy, telecom, health)
- provides lots of analyzers as plug-ins
- C code may be annotated with ACSL formal specifications

This talk: E-ACSL

- Runtime verification with Frama-C by means of E-ACSL
- as automatic as possible
- for finding security weaknesses

E-ACSL Basics

E-ACSL, as a specification language

- Executable-ACSL: executable subset of ACSL
- `\forall integer i; 0 <= i < len-1 ==> t[i] <= t[i+1]`

E-ACSL, as a tool

- convert formal annotations into C code to be validated at runtime
-
```
int res = 1;
for(int i = 0; i < len-1; i++)
  if (t[i] > t[i+1]) { res = 0; break; }
e_acsl_assert(res);
```
- runtime assertion checker

Automatic Usages of E-ACSL for Security

some Frama-C plug-ins generate E-ACSL annotations

- RTE + E-ACSL for preventing undefined behaviors to be executed
- SecureFlow + E-ACSL for detecting information flow leakage
- Aorai + E-ACSL for API conformance checking

Usage 1: Detecting Undefined Behaviors

- Plug-in RTE automatically generates E-ACSL annotations for preventing potential undefined behaviors
- Combining RTE + E-ACSL allows to detect undefined behaviors at runtime

Usage 1: Detecting Undefined Behaviors

- Plug-in RTE automatically generates E-ACSL annotations for preventing potential undefined behaviors
- Combining RTE + E-ACSL allows to detect undefined behaviors at runtime

```
int incr(int x) {
  return x++;
}
```

Usage 1: Detecting Undefined Behaviors

- Plug-in RTE automatically generates E-ACSL annotations for preventing potential undefined behaviors
- Combining RTE + E-ACSL allows to detect undefined behaviors at runtime

```
int incr(int x) {
  return x++;
}
```

RTE →

```
int incr(int x) {
  /*@ assert x+1 <= 2147483647; */
  return x++;
}
```

Usage 1: Detecting Undefined Behaviors

- Plug-in RTE automatically generates E-ACSL annotations for preventing potential undefined behaviors
- Combining RTE + E-ACSL allows to detect undefined behaviors at runtime

```
int incr(int x) {
  return x++;
}
```

RTE

```
int incr(int x) {
  /*@ assert x+1 <= 2147483647; */
  return x++;
}
```

E-ACSL

```
int incr(int x) {
  /*@ assert x+1 <= 2147483647; */
  e_acsl_assert(x+1L <= 2147483647L);
  return x++;
}
```

Usage 1: Detecting Undefined Behaviors

- Plug-in RTE automatically generates E-ACSL annotations for preventing potential undefined behaviors
- Combining RTE + E-ACSL allows to detect undefined behaviors at runtime

```
int incr(int x) {
  return x++;
}
```

RTE →

```
int incr(int x) {
  /*@ assert x+1 <= 2147483647; */
  return x++;
}
```

↓ E-ACSL

```
int incr(int x) {
  /*@ assert x+1 <= 2147483647; */
  e_acsl_assert(x+1L <= 2147483647L);
  return x++;
}
```

- above, why does it convert 1 to 1L?

Usage 1: Detecting Undefined Behaviors

- Plug-in RTE automatically generates E-ACSL annotations for preventing potential undefined behaviors
- Combining RTE + E-ACSL allows to detect undefined behaviors at runtime

```
int incr(int x) {
  return x++;
}
```

RTE →

```
int incr(int x) {
  /*@ assert x+1 <= 2147483647; */
  return x++;
}
```

↙ E-ACSL

```
int incr(int x) {
  /*@ assert x+1 <= 2147483647; */
  e_acsl_assert(x+1L <= 2147483647L);
  return x++;
}
```

- above, why does it convert 1 to 1L?
  - for being efficient, while remaining sound

Usage 1: E-ACSL Memory Monitoring
cont'd

```
void f(void) {
  int x, y, z, *p;



  p = &x;
  x = 0;
  y = 1;
  z = 2;
  /*@ assert \valid(p); */



  *p = 3;



}
```

Usage 1: E-ACSL Memory Monitoring

cont'd

```
void f(void) {
  int x, y, z, *p;
  // local variable allocation
  store_block((void *)(&p), 4U); store_block((void *)(&z), 4U);
  store_block((void *)(&y), 4U); store_block((void *)(&x), 4U);
  full_init((void *)(&p)); p = &x; // initialization of p
  full_init((void *)(&x)); x = 0;  // initialisation de x
  full_init((void *)(&y)); y = 1;  // initialisation de y
  full_init((void *)(&z)); z = 2;  // initialisation de z
  /*@ assert \valid(p); */
  // validity check
  { int e_acsl_valid;
    e_acsl_valid = valid((void *)p, sizeof(int));
    e_acsl_assert(e_acsl_valid); }
  *p = 3;
  // memory deallocation
  delete_block((void *)(&p)); delete_block((void *)(&z));
  delete_block((void *)(&y)); delete_block((void *)(&x));
}
```

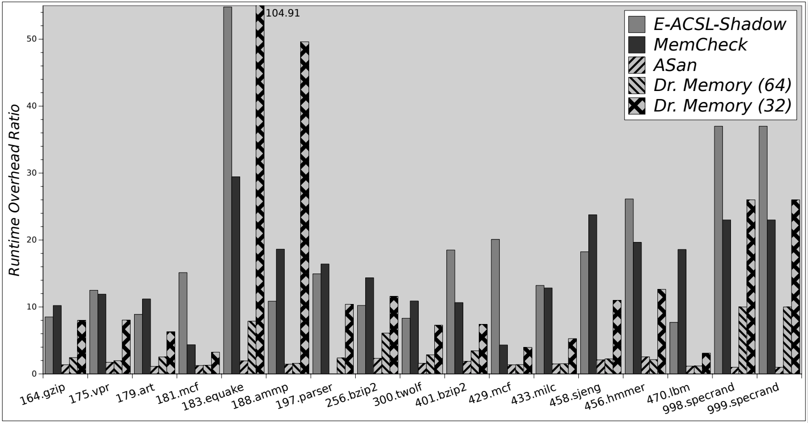
Usage 1: E-ACSL Memory Monitoring

cont'd

```
void f(void) {
  int x, y, z, *p;
  // local variable allocation
  store_block((void *)(&p), 4U); store_block((void *)(&z), 4U);
  store_block((void *)(&y), 4U); store_block((void *)(&x), 4U);
  full_init((void *)(&p)); p = &x; // initialization of p
  full_init((void *)(&x)); x = 0;  // initialization of x
  full_init((void *)(&y)); y = 1;  // initialization of y
  full_init((void *)(&z)); z = 2;  // initialization of z
  /*@ assert \valid(p); */
  // validity check
  { int e_acsl_valid;
    e_acsl_valid = valid((void *)p, sizeof(int));
    e_acsl_assert(e_acsl_valid); }
  *p = 3;
  // memory deallocation
  delete_block((void *)(&p)); delete_block((void *)(&z));
  delete_block((void *)(&y)); delete_block((void *)(&x));
}
```

Efficient Generated Code

E-ACSL Performance



$\times 17$ time-overhead; $\times 2.4$ memory overhead on SPEC-CPU

comparable to Valgrind; still slower than AddressSanitizer
less memory-overhead than these tools

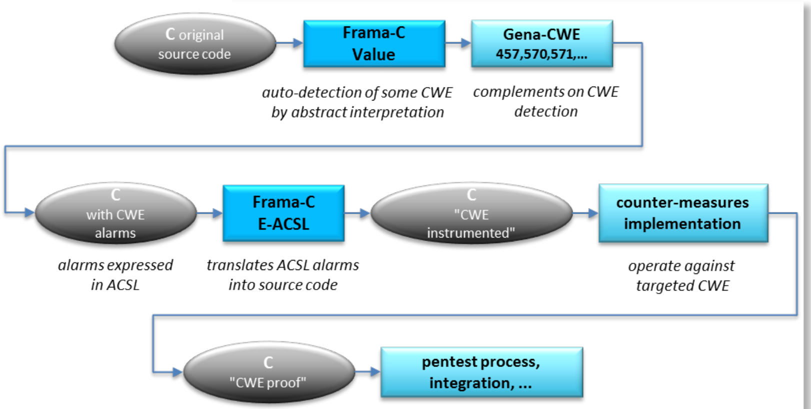
E-ACSL Expressiveness

Defect Type	E-ACSL	Google's Sanitizers
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

Application to Generating Counter-Measures

by Dassault Aviation



First, use automatic static analysis to detect vulnerabilities

Then, switch to fast runtime monitoring

Experimented on modules from Apache / OpenSSL

Usage 2: Information Flow Tracking

- SecureFlow encodes program's information flow to C code
- allow usual code analyzers to verify information flow properties with standard techniques
  - Frama-C/Eva
  - Frama-C/E-ACSL
- does any private key leak on a public channel?
- may also detect some time-channel attacks
- experimented on LibTomCrypt (60,000-line C crypto library)
  - prove all the 14 symmetric cryptosystems as secure
  - detect one known vulnerability in 1 asymmetric cryptosystem
  - detect that the patch is unsecure

Usage 3: API Conformance Checking

- Aorai encodes temporal properties / automata to C code
- once again, allow usual code analyzers to be used
  - Frama-C/WP
  - Frama-C/E-ACSL
- is my crypto function always called after being initialized and eventually cleaned?
- experimented on a Linux driver to check that a sequence of hardware events does never happen from API calls

Conclusion

- E-ACSL: Frama-C's runtime verification tool
- may detect undefined behaviors
  - reasonably efficient
  - better detection power than Google's sanitizer and Valgrind
- may detect incorrect information flows
- may detect incorrect API usages
- future works include
  - supporting additional safety and security properties
  - generating much more efficient code
    - time-overhead may still be reduced by $2\times$, at the least

A Few References

- J. Signoles, N. Kosmatov, and K. Vorobyov.
  E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs (tool paper).
  RV-CuBES 2017.

- K. Vorobyov, J. Signoles, and N. Kosmatov.
  Shadow state encoding for efficient monitoring of block-level properties.
  ISMM 2017.

- D. Pariente and J. Signoles.
  Static Analysis and Runtime Assertion Checking: Contribution to Security Counter-Measures.
  SSTIC 2017.

- G. Barany and J. Signoles.
  Hybrid Information Flow Analysis for Real-World C Code.
  TAP 2017

Bonus: E-ACSL Integer Support

- use GMP library for mathematical integers

```
/*@ assert y-1 == 0; */
mpz_t e_acsl_1, e_acsl_2, e_acsl_3, e_acsl_4;
int e_acsl_5;
mpz_init_set_si(e_acsl_1, y);                    // e_acsl_1 = y
mpz_init_set_si(e_acsl_2, 1);                    // e_acsl_2 = 1
mpz_init(e_acsl_3);
mpz_sub(e_acsl_3, e_acsl_1, e_acsl_2);   // e_acsl_3 = y-1
mpz_init_set_si(e_acsl_4, 0);                    // e_acsl_4 = 0
e_acsl_5 = mpz_cmp(e_acsl_3, e_acsl_4); // (y-1) == 0
e_acsl_assert(e_acsl_5 == 0);                   // runtime check
mpz_clear(e_acsl_1); mpz_clear(e_acsl_2); // deallocate
mpz_clear(e_acsl_3); mpz_clear(e_acsl_4);
```

- how to restrict GMPs as most as possible? on-the-fly typing

almost no GMP in practice

[Jakobsson, Kosmatov & Signoles @JFLA'15]