FROM SAFETY TO SECURITY

SOFTWARE ASSESSMENTS AND GUARANTEES

FLORENT KIRCHNER (LIST)
Figure 1-2. Software Systems Size Growth 1960 to 1995
ONBOARD SOFTWARE SIZE

- F-22 (2005)
- 787 (2010)
- F-35 (2010)
- F-35 (2012)
- Volt (2011)
TRUST

WHY DO WE (OUR) SOFTWARE?
DO-178C FOR AEROSPACE SAFETY
EN 50128 FOR RAIL SAFETY
IEC 60880 FOR NUCLEAR INSTALLATIONS
IEC 62304 FOR MEDICAL EQUIPMENT
ISO 26262 FOR ROAD VEHICLES
TRUST WHY DO WE (OUR) SOFTWARE?
“[20 KLOC NASA software] all had from 6.5 to nearly 9 defects per KLOC found in system test, and that does not guarantee that all the defects were found.”

- Watts S. Humphrey, 1999

“If we are to achieve the expected gains from purchasing software versus building it ourselves, then for the entire life cycle of the products, we cannot allow any modification.”

- Business Process Reengineering with Commercial Off-the Shelf Software
APPROACHES TO SOFTWARE RELIABILITY
Software is quite unlike other human artifacts

... roads, power lines, jet engines – even microcontrollers and circuit boards

Follows a **clean** set of rules in a **defined** environment

\[0101 \oplus 0011 \Rightarrow 0110\]
Initial state

Empty slot
Occupied slot

Aim
Reach the following final state.

One move
TRY AND PLAY

→ There are many possible **pegs** and **moves** to consider

→ It is hard to get a **global vision** for how to get to a **given final state**
3-WORD VOCABULARY | WORD 1: REMOVE A LINE

1. Empty slot
2. Master peg
3. One can jump directly from 1 to 4
4. The three pegs are gone

The three pegs are gone
3-WORD VOCABULARY | WORDS 2 AND 3
BACK TO A SIMPLE PUZZLE

This formalism has resulted in a more manageable problem, and guided us toward the solution.

- Semantics
  The three words

- Algorithm
  The sequence of words used to solve the puzzle
A GENERIC FORMALISM | USED TO SOLVE ANY PUZZLE
A FORMALISM FOR SOFTWARE VALIDATION

Specifies logical behaviors

```c
/*@ ensures \result >= x && \result >= y;
ensures \result == x || \result == y; */

int max (int x, int y) {
    return (x > y) ? x : y;
}
```

Mathematical universe

Frame the computational universe using logics

Computational universe

Modifies transistor states on micro-chips during execution
A FORMALISM FOR SOFTWARE VALIDATION

specifies logical behaviors

**Mathematical universe**

\[
\text{prop\_surete} = \text{IMPLIQUE}(\text{ouvert}, \text{not}\left(\text{en\_route}\right));
\]

\[
\text{en\_route} = \begin{cases} 
\text{if} (\text{ouvert or abandon}) & \text{then false} \\
\text{else if demarrage} & \text{then true} \\
\text{else} & (\text{false} \rightarrow \text{pre}\left(\text{en\_route}\right));
\end{cases}
\]

Frame the computational universe using logics

Models transistor states on micro-chips during execution
FORMAL SPECIFICATION...

• Properties are formalized using unequivocal specifications

\( \forall a, i ; \text{valid}(a+(0..N-1)) \Rightarrow o \leq i < N \Rightarrow a[i] \leq C \)

• Software systems are treated as sets of rules
  • Transforming the system state
  • Satisfying certain properties

... AND PROOFS

• Formal methods are used to prove that some software properties hold...

• ... or to provide insight on why other properties do not.
TOOLS FOR HIGH CONFIDENCE SOFTWARE & SYSTEMS
% Conflict during interval \([B,T]\)
\begin{align*}
\text{conflict}_2\text{D}?&(s,v): \text{bool} = \\
&\text{EXISTS } (t: \text{Lookahead}): \\
&\text{sqv}(s+t*v) < \text{sq}(D)
\end{align*}

% 2-D Conflict Detection (cd2d)
\begin{align*}
\text{cd2d}?&(s,v): \text{bool} = \\
&\text{horizontal_los?}(s+B*v) \text{ OR } \\
&\omega_v(s)(v) < 0
\end{align*}

% THEOREM: cd2d is correct and complete
\begin{align*}
\text{cd2d} : &\text{THEOREM} \\
\text{conflict}_2\text{D}?&(s,v) \\
&\text{IFF} \\
&\text{cd2d}?&(s,v)
\end{align*}
```c
int abs(int x)
{
    int r;
    if (x >= 0)
        r = x;
    else
        r = -x;
    return r;
}

/*@ requires -1000 <= x <= 1000;
   ensures \result >= 0; */
int abs(int x)
{
    int r;
    if (x >= 0)
        r = x;
    else
        r = -x;
    return r;
}
```
A SOURCE CODE ANALYSIS PLATFORM

- Formal **mathematical specification** language with ACSL
- Software **computations** in ANSI / ISO C 99
- Modular **verifiers** following varied approaches
FRAMA-C | AN EXTENSIBLE PLATFORM

```
/*@ ensures \result >= x && \result >= y;
   ensures \result == x || \result == y;
*/
int max (int x, int y) { return (x > y) ? x : y; }
```
/*@ ensures \result >= x && \result >= y; ensures \result == x // \result == y; */
int max (int x, int y) { return (x > y) ? x : y; }
Function: main
Statement: 8
Variable S has type "int ".
It is a global variable.
It is referenced and its address is not taken.
Before statement:
S ∈ {0; 1; 3; 6; }
At next statement:
S ∈ {0; 1; 3; 6; 10; }
```c
void main(int n)
{
    p = T;
}
    int i;
    i = 0;
    while (i < n) {
        /*@ assert \valid(p+i); 
           // synthesized */
        *(p + i) ++;
    }
    i ++;
}
```

Function: main
Statement: 8 (line 9 in work/\(\cdot\))
Variable p has type "int *".
It is a global variable.
It is referenced and its address is not taken.
Variable i has type "int".
It is a local variable.
It is referenced and its address is not taken.
Before statement (evaluation may have failed in some cases):
*(p + i) ∈ [\(\cdot\\cdot\\cdot\\cdot\)'
After statement:
*(p + i) ∈ [\(\cdot\\cdot\\cdot\\cdot\)']
Can we guarantee the absence of defaults in large system-level code?

- Pinpoint all runtime errors and help investigate their cause
- Structural properties on memory separation and cyclic behaviors.
- Using collaborative, automated analyses.
- Justification and certification to IEC60880 Class 1

- 100+ kloc
- C source code
- Highest certification requirements
- 80% code coverage
- 200 alarms
#ifndef OPENSSL_NO_HEARTBEATS
int
tls1_process_heartbeat(SSL *s)
{
    unsigned char *p = &s->s3->rrec.data[0], *pl;

    /* Read type and payload length first */
    hbtype = *p++;
    n2s(p, payload);
    pl = p;

    if (hbtype == TLS1_HB_REQUEST)
    {
        /* Enter response type, length and copy payload */
        *bp++ = TLS1_HB_RESPONSE;
        s2n(payload, bp);
        memcpy(bp, pl, payload);
        bp += payload;
        /* Random padding */
        RAND_pseudo_bytes(bp, padding);

        r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);

        if (r >= 0 && s->msg_callback)
            s->msg_callback(1, s->version, TLS1_RT_HEARTBEAT,

#endif

2552 #ifndef OPENSSL_NO_HEARTBEATS
2553 int
2554 tls1_process_heartbeat(SSL *s)
2555 {
2556     unsigned char *p = &s->s3->rrec.data[0], *pl;
2557 [...]
2561     /* Read type and payload length first */
2562     hbtype = *p++;
2563     n2s(p, payload);
2564     pl = p;
2565 [...]
2567     if (hbtype == TLS1_HB_REQUEST)
2568     {
2569 [...]
2574     /* Enter response type, length and copy payload */
2575     *bp++ = TLS1_HB_RESPONSE;
2576     s2n(payload, bp);
2577     memcpy(bp, pl, payload);
2578     bp += payload;
2579     /* Random padding */
2580     RAND_pseudo_bytes(bp, padding);
2581 [...]
2587     r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
2588 [...]
2593     if (r >= 0 && s->msg_callback)
2594         s->msg_callback(1, s->version, TLS1_RT_HEARTBEAT,
DETECTING SECURITY FLAWS | COTS LIBRARY

How do we reach the sophisticated “last vulnerabilities” in core IT components?

• Detect all occurrences of a given category of vulnerabilities.

• Using automated code analyses

• Handling general-purpose code constructs

• Working with ANSSI, ETSI, and NIST on cybersecurity

Fixed a condition where QLZ_MEMORY_SAFE could fail detecting corrupted data. Thanks to Pascal Cuoq and Kerstin Hartig who used Frama-C's value analysis!
FRAMA-C | CORE COMPONENTS

```c
/*@ ensures \result >= x && \result >= y;
   ensures \result == x || \result == y;
+/
int max (int x, int y) { return (x > y) ? x : y; }
```

Deductive Verification
```c
/*@ requires 0 <= k ∧ k < 10;
   ensures K: k = old(k)+1;

   ensures P: p = {old(p) \with \[old(k)] = \[old(x)]};
   assigns k, p[k];
*/

void job(int x)
```

---

<table>
<thead>
<tr>
<th>Module</th>
<th>Property</th>
<th>Kind</th>
<th>Model</th>
<th>WP</th>
<th>Alt-Ergo</th>
<th>Coq</th>
<th>Z3</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function 'job'</td>
<td>Assigns ...</td>
<td>Property</td>
<td>store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function 'job'</td>
<td>Post-condition 'K'</td>
<td>Property</td>
<td>store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function 'job'</td>
<td>Post-condition 'P'</td>
<td>Property</td>
<td>store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function 'job2'</td>
<td>Assigns ...</td>
<td>Property</td>
<td>store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function 'job2'</td>
<td>Post-condition 'K'</td>
<td>Property</td>
<td>store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

1. Goal `store_job_post_K`:

   ```
  forall k_0:int.
   is_sint32(k_0) ->
   (0 <= k_0) ->
   (k_0 < 10) ->
   (let k_1 = (k_0+1) in
   tag_K:(k_1 = (k_0+1)))
   ```
How do we prove that embedded software properties are satisfied?

- Automate unit proofs on critical code
- Using C code and ACSL stubs
- A380, A350 & A400M programs
- Qualification and certification to DO178B/C level A
SECURITY POLICY VALIDATION | CLOUD COMPUTING

Can we instantiate and verify security-policies on a custom execution platform?

- Formal interactive verification of the page allocation algorithm
- Software property specifications for confidentiality
- Derived from system-level specifications
**ALGORITHMIC CONFORMANCE PROOF | ATC**

**Theorem** (CD2D Correctness & Completeness)

\[ \forall s, v_o, v_i, conflict(s, v_o, v_i, D) \iff cd2d(s, v_o, v_i, D) \]
FRAMA-C | CORE COMPONENTS

```c
/*@ ensures \result >= x && \result >= y;
   ensures \result == x || \result == y;
*/
int max (int x, int y) { return (x > y) ? x : y; }
```

Structural exploration
Binary search of a given element in a given ordered array returning 1 if the element is present and 0 if not. In this example, the array dimension is fixed. The example is interesting because of - the loop with a variable number of iterations - the precondition that the input array is ordered and - an example of an oracle which is a more inefficient implementation of the same algorithm.

```c
int Bsearch ( int A[10], int elem){
    int low, high, mid, found; low = 0; high = 9; found = 0;
    while ( ( high > low ) ) {
        mid = (low + high) / 2 ;
        if( elem == A[mid] )
            found = 1;
        else
            if( elem > A[mid] )
                low = mid + 1 ;
            else
                high = mid - 1;    
    }    mid = (low + high) / 2 ;
    if( ( found != 1)  && ( elem == A[mid]) )
        found = 1;
    return found ;}
```

---

### Test-cases generated

**General test session information**

- **Function under test**: Bsearch
- **Coverage criterion**: all feasible paths

<table>
<thead>
<tr>
<th>Test case ID</th>
<th>Verdict</th>
<th>Time, sec.</th>
<th>Prefix ID</th>
<th>Path prefix/suffix</th>
</tr>
</thead>
</table>
| TC_1         | failure | 0          | P_1       | bsearch.c: +18;-21;+23;+18;-12;+23;+18;-21;+23;+18;-12;+23;+18;+18;+21;+18;+21;+23;+18;+18;+21;+18;
| TC_2         | failure | 0          | P_2       | bsearch.c: +18;-21;+23;+18;-21;+23;+18;-21;+23;+18;-21;+23;+18;+18;+21;+18;+21;+23;+18;+18;+21;+18;
| TC_3         | failure | 0          | P_5       | bsearch.c: +18;-21;+23;+18;-21;+23;+18;-21;+23;+18;-21;+23;+18;+18;+21;+18;+21;+23;+18;+18;+21;+18;
| TC_4         | success | 0          | P_9       | bsearch.c: +18;-21;+23;+18;-21;+23;+18;-21;+23;+18;-21;+23;+18;+18;+21;+18;+21;+23;+18;+18;+21;+18;
| TC_5         | failure | 0          | P_14      | bsearch.c: +18;-21;+23;+18;-21;+23;+18;-21;+23;+18;-21;+23;+18;+18;+21;+18;+21;+23;+18;+18;+21;+18;
| TC_6         | failure | 0          | P_15      | bsearch.c: +18;-21;+23;+18;-21;+23;+18;-21;+23;+18;-21;+23;+18;+18;+21;+18;+21;+23;+18;+18;+21;+18;
| TC_7         | failure | 0          | P_18      | bsearch.c: +18;-21;+23;+18;-21;+23;+18;-21;+23;+18;-21;+23;+18;+18;+21;+18;+21;+23;+18;+18;+21;+18;
| TC_8         | success | 0          | P_22      | bsearch.c: +18;-21;+23;+18;-21;+23;+18;-21;+23;+18;-21;+23;+18;+18;+21;+18;+21;+23;+18;+18;+21;+18;
| TC_9         | success | 0          | P_27      | bsearch.c: +18;-21;+23;+18;-21;+23;+18;-21;+23;+18;-21;+23;+18;+18;+21;+18;+21;+23;+18;+18;+21;+18;
LINEAR TEMPORAL LOGICS

Can we specify timing constraints on sequential behaviors?

- Function call automatas including counters and guarded parameters
- Frama-C/Aoraï translate states and transitions to code-level annotations
- Use in conjunction with Frama-C/WP or Frama-C/Value to perform ACSL-level verification

```latex
/*@
 requires aorai_state_S_0 == 1 || aorai_state_S_1 == 1 || ...;
 requires aorai_state_S_0 == 1 ==>
 has_possible_transition_S0; ...;
 ensures aorai_state_S_2 == 1 || aorai_state_S_3 == 1 || ...;
 ensures old(aorai_state_S_0 == aorai_state_S_2 == 1) ==>
 program_state_when_in_S_2; ... */
```
RUNTIME MONITORING AND VERIFICATION

Is it possible to toughen runtime checks in dynamic and compositional systems?

- Automate monitor generation from formal specifications
- Benefit from static analyses to infer specifications and minimize overhead
- Use in conjunction with hardware enablers

```c
int div (int x, int y) {
    // make sure y != 0;
    return x / y;
}
```

```c
int div (int x, int y) {
    if (y==0) e_acsl_fail();
    return x / y;
}
```
HYBRID ANALYSES

Can we refine static analysis results using dynamic techniques?

- False alarms are raised statically, but never occur dynamically

- Use automated test case generation techniques to reach alarm points

- Simplify program exploration by factoring out non-relevant constructs
GAMIFYING CYBERSECURITY

Is it possible to make CWE verification more cost-effective?

• Leverage crowd-sourcing and gaming communities

• Create easy questions on anonymized software items.

• Merge partial results into a global rigorous verification

• Check viability on large, open-source codebases

GAMIFYING CYBERSECURITY

Is it possible to make CWE verification more cost-effective?

• Leverage crowd-sourcing and gaming communities

• Create easy questions on anonymized software items.

• Merge partial results into a global rigorous verification

• Check viability on large, open-source codebases


Frama-C
A Software Analysis Perspective*

Pascal Cuoq, Florian Kirscher, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakovlevski

with Patrick Basile, Richard Boucher, Bernard Bokem, Loïc Cornelsen, Zynose Dazaghe, Philippe Herrmann, Benjamin Monate, Yannick Mey, Anne Pacalet, Arnaud Prevetti, Muriel Roger, and Neky Williams

CEA, LIST, Software Safety Laboratory, 374, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr

Abstract. Frama-C is a source code analysis platform that aims at conducting verification of industrial-size C programs. It provides its users with a collection of plugins that perform static analysis, deductive verification, and testing, for safety- and security-critical software. Collaborative verification across cooperating plugins is enabled by their integration on top of a shared kernel and datastructures, and their compliance to a common specification language. This foundational article presents a synthetic view of the platform, its tools and composite analyses, and some of its industrial achievements.

Introduction

In past forty years, many of the groundwork of formal software analysis have been laid. Several angles and theoretical avenues have been explored, from deductive reasoning to abstract interpretation to program transformation to concolic testing. While much remains to be done from an academic standpoint, one of the major advances in those fields are already being successfully implemented and with growing industrial interest. The ensuing tasks for mainstream diffusion of software analysis techniques has raised several challenges. Chief among them are: a) their scalability, b) their interoperability, c) the soundness of their results.

Parallelism is predictably important from the point of view of adaptability. Scaling to large problems is a prerequisite for the industrial diffusion of software analysis and verification techniques. It also represents a mean to better understand how to engage teams (e.g. pointers, unions, or dynamic memory allocation) influence the underlying architecture of large software developments. Overall, achieving scalability in the design of software analyzers for a wide range of software patterns remains a difficult question.

*This work was partly supported by ANR FSCART and No CLOCK, and FUD Hi-Lite projects
WHAT WE DO | LIST

- Research and technology: software safety & security tools
- For intrinsically secure, high-confidence software and systems
- Contributing to normative and defacto standards
- Supporting component certification and tool deployment