

Verificarea Monocypher cu Framac/Eva

ADRIAN MANEA, 410 SLA

Cuprins

1	Uneltele de verificare	5
1.1	FramaC	5
1.2	Eva	8
1.2.1	Motivație	8
1.2.2	Interpretarea abstractă	8
2	Verificarea	11
2.1	Rularea inițială, fără Eva	11
2.2	Verificarea cu Eva	13
2.3	Rafinarea verificării	15
3	Concluzii	19
	Bibliografie	21

Proiectul prezentat este Monocypher ([Loup Vaillant \(2019a\)](#)), care este o bibliotecă criptografică scrisă de Loup Vaillant, începînd cu 2016. După cum menționează autorul pe pagina [Loup Vaillant \(2017\)](#), intenția a fost de a concepe o bibliotecă de criptare suficient de sigură, de rapidă, dar și de simplă pentru a fi folosită atît de el, cît și de alții. Autorul a considerat că bibliotecile actuale ori nu sînt suficient de sigure, ori sînt supraîncărcate pentru necesitățile sale, așa că a pornit în a-și concepe propria soluție. Proiectul este scris în C.

Verificarea proiectului Monocypher se va face folosind pachetul Framac ([Baudin \(2019\)](#)), specific pentru limbajul C. Cum Framac este, de fapt, mai curînd un mediu în care se pot rula mai multe unelte de verificare formală, am ales verificarea Monocypher cu Eva.

Detaliile despre modul de funcționare a proiectului Monocypher, cît și a verificării prin Eva sînt date în secțiunile următoare.

Toate exemplele și capturile proprii au fost realizate pe o mașină Core i7 8th gen și:

- `frama-c` Chlorine 20180502, instalat via `opam`;
- Eva 18 Argon;
- sistemul de operare Manjaro 18 (Illyria), Community Edition i3 ([Manjaro \(2019\)](#));
- editorul text Emacs 26.2 [GNU \(2019\)](#) cu `acsl-mode` pentru adnotări ACSL ([Framac \(2016\)](#));
- emulatorul de terminal `st` ([Suckless \(2019\)](#)), cu modificările lui Luke Smith ([Smith \(2019\)](#)).

1.1 FramaC

FramaC este o unealtă de verificare formală dezvoltată de un grup de cercetători de la Commissariat à l'Énergie Atomique (CEA-list) și INRIA din Franța.

În esență, FramaC verifică programe scrise în C, dar este, de fapt, un cadru în care se pot include și utiliza unelte de verificare cu scopuri precise. În varianta cea mai simplă, FramaC folosește un dialect al limbajului C (de fapt, o variantă simplificată numită *C Intermediate Language*) și oferă metode de verificare similare cu uneltele bazate pe *adnotarea codului*, precum Dafny și altele. Limbajul se numește ACSL și este descris detaliat la [FramaC \(2019a\)](#).

FramaC însuși a fost implementat în limbajul OCaml, astfel că poate fi instalat atât individual, cât și prin managerul de pachete opam.

Odată instalat, FramaC oferă două moduri de interacțiune: la nivel de linie de comandă (CLI) sau prin interfață grafică (GUI). Este de menționat faptul că verificatorul nu oferă un editor, astfel că modalitatea recomandată de utilizare este:

- (1) se scrie programul C cu editorul preferat;
- (2) se rulează din linia de comandă pe fișierele-sursă de verificat;
- (3) fie se redirectionează rezultatul într-un fișier separat, de exemplu `frama-c *.c » log`, fie se salvează într-un format binar, specific, cu comanda `frama-c *.c -save mysession.sav`;
- (4) se consultă fișierul `log`, dacă s-a folosit prima variantă, care este un fișier text simplu, sau, preferabil, se încarcă formatul salvat specific în interfața grafică, cu comanda `frama-c-gui -load mysession.sav`.

Dacă se alege varianta simplă, se poate observa că informațiile din log nu sînt foarte explicite mereu, astfel că trebuie să știm să interpretăm rezultatele. În general, ele sînt afișate sub forma `[nivel] fișier rezultat`, unde:

- `[nivel]` este nivelul la care se face verificarea (e.g. nivelul de bază, adică direct codul sursă, dacă s-au folosit adnotări și nu se apelează vreun plugin;
- `fișier` este fișierul verificat, despre care se raportează rezultatele;
- `rezultat` este diagnosticul sau concluzia la care a ajuns analizatorul, la nivelul `[nivel]` asupra fișierului `fișier`.

În varianta cu interfață grafică, însă, putem vedea mult mai multe informații, într-o variantă mult mai flexibilă:

- În panoul din stînga se pot vedea fișierele încărcate spre analizare, care au fost „desfăcute” în funcțiile conținute, care pot fi analizate individual;
- Panoul median arată modul în care Framac a interpretat codul-sursă, eventual scriindu-și singur adnotări;
- Panoul din dreapta arată codul-sursă încărcat (în care *nu* se poate scrie);
- Panoul din stînga-jos arată uneltele folosite pentru verificare, eventual plugin-urile încărcate;
- Panoul de jos arată mesajele și erorile generate, cu cîmpurile populate în funcție de uneltele de verificare folosite.

Un exemplu simplu este prezentat în figura 1.1.

Deși poate fi utilizat în această formă, este recomandabil ca Framac să fie apelat folosind plugin-uri specifice. În verificarea proiectului Monocypher, vom folosi plugin-ul Eva (varianta actualizată [*Evolved*] a *Value Analysis*), care este descris în secțiunea următoare.

Observăm, de asemenea, că în imaginea din 1.1, în dreptul adnotărilor apar pătrate goale, cu mesajul `Never tried. No status is available for this property`. Aceasta se datorează faptului că nu s-a apelat niciun plugin, iar uneltele de bază pentru verificare incluse doar în `frama-c` nu pot verifica codul inclus. O verificare simplă poate fi rulată cu plugin-ul RTE (*RunTime Evaluation*), de exemplu, dar ne vom concentra în continuare pe Eva.

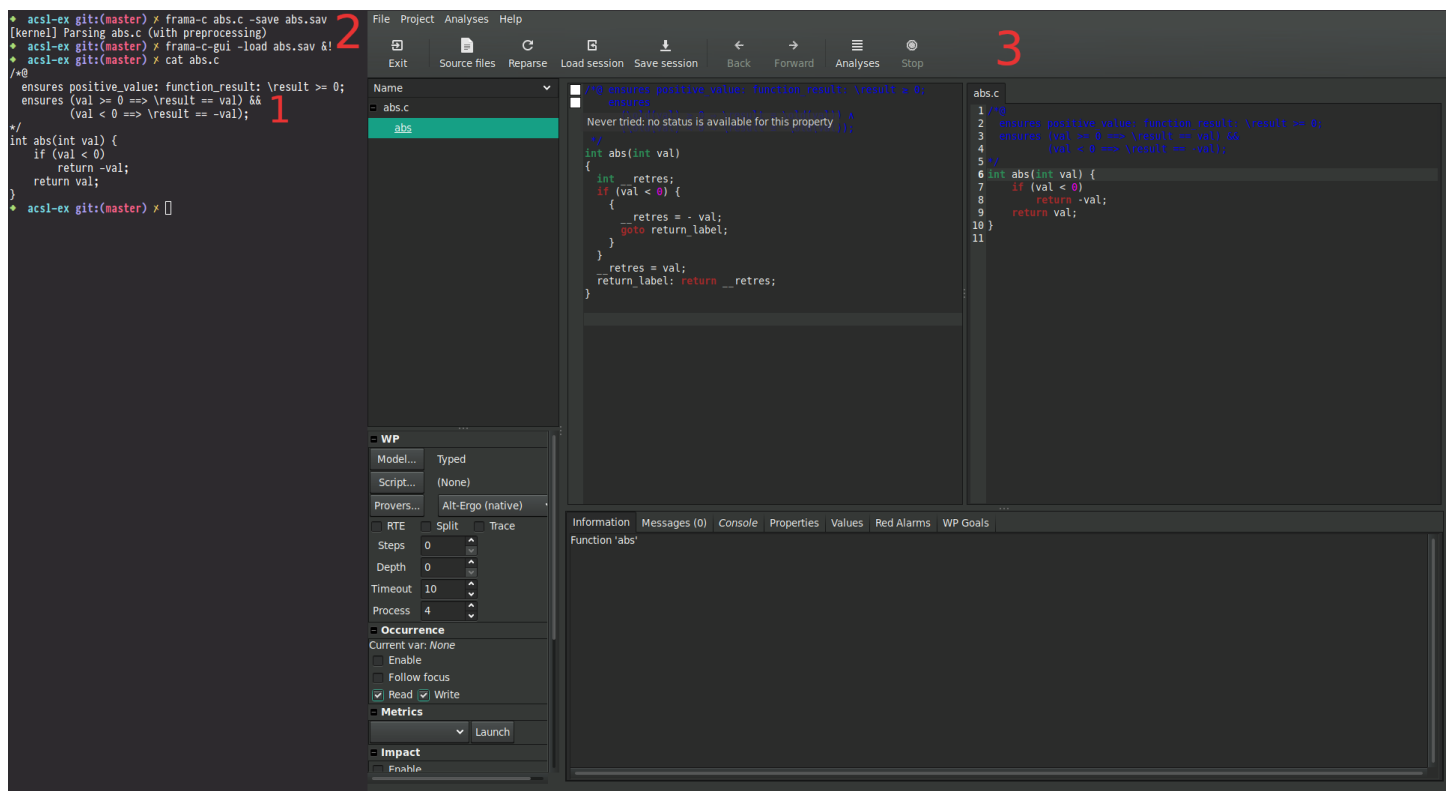


Figura 1.1: Exemplu FramaC: (1) adnotări, (2) CLI, (3) GUI

1.2 Eva

1.2.1 Motivație

Ca majoritatea uneltelor criptografice, Monocypher se bazează pe expresii și operații numerice, eventual cu valori foarte mari și prelucrări pe biți. Aceasta îl face să fie posibil vulnerabil atât la operații matematice interzise (precum împărțirea la 0 sau depășirea intervalului de valori permise de tipul de date folosit) sau la scurgeri de memorie.

Eva se bazează pe o teorie matematică numită *interpretare abstractă* (Wikipedia (2019)), conform și Framac (2019b), care îl face în special potrivit pentru verificarea programelor cu conținut aritmetic bogat. De asemenea, Eva poate detecta și scurgeri de memorie sau operații nepermise la nivel de memorie alocată, dar pentru verificări mai avansate în special asupra memoriei, este recomandabil să se folosească Valgrind (Valgrind (2019)).

1.2.2 Interpretarea abstractă

Această teorie matematică este folosită în semantica unor programe, mai precis în aproximarea valorilor pe care le au variabilele în diverse blocuri de cod. Într-un anumit sens, interpretarea abstractă poate fi asemănată cu execuția simbolică sau concolică, prin aceea că încearcă să folosească bucățile de cod dimprejurul unor operații cu variabile pentru a aproxima valorile pe care le au acele variabile. Însă, în forma simplă cel puțin, interpretarea abstractă nu folosește SMT sau SAT solve.

Un exemplu tipic care motivează și totodată introduce modul de lucru al interpretării abstracte este următorul. Presupunem că la o petrecere sînt invitate niște persoane și vrem să ne asigurăm că știm exact ce persoane au venit. Varianta care funcționează sigur este să cerem fiecărei persoane să se identifice cu un identificator unic, cum este CNP-ul în România. Atunci am avea o informație precisă, nu aproximativă. Presupunem, însă, că această variantă nu este fezabilă, deoarece CNP-ul cere prea mult spațiu de stocare, să spunem. Atunci putem folosi doar numele participanților, fără spațiu suplimentar, deoarece oricum ei și-ar fi dat numele. Dar atunci, nu mai putem spune cu exactitate dacă o anumită persoană este prezentă. Atunci cînd există conflicte de nume, putem semnală o atenționare sau, dacă situația este serioasă (e.g. căutăm pe lista invitaților persoana care a spart candelabrul), chiar putem plasa o alarmă în dreptul tuturor celor care au același nume sau diferă prin doar 1-2 litere, chiar cu riscul ca alarma să fie falsă.

Ideea de bază este că se poate da o semantică destul de precisă în cazul majorității limbajelor de programare — așa-numita *semantică concretă* —, care să țină cont și de aspecte operaționale ale limbajului, i.e. implementarea propriu-zisă plus compilatoarele. Dar, dacă este necesar să se evalueze un anumit program scris în limbajul respectiv, chiar în prezența semanticii concrete sau operaționale, nu se pot verifica precis valorile pe care le au toate variabilele din program (o problemă imposibil de rezolvat, în general, din motive de calculabilitate). Astfel că este necesar să se folosească aproximări, bazate pe diverse teorii, cu riscul asumat de supra- sau sub-aproximare.

Ideea de bază în asemenea situații este să se *abstractizeze*. Adică, pornind de la semantica concretă, care este destul de precisă din punct de vedere teoretic, dar aproape imposibil de folosit din punct de

vedere practic, se pot forma așa-numitele *semantici abstracte*, în care interpretarea programelor este dată într-un anumit cadru matematic. Detalii și exemple se pot găsi pe site-ul lui Patrick Cousot, cel care a introdus metoda interpretării abstracte în anul 1970, împreună cu Radhia Cousot ([Cousot \(2008\)](#)).

Interpretarea abstractă poate fi făcută *ad-hoc*, pentru un anumit program sau fragment. Prezintă un exemplu, preluat de la [Wikipedia \(2019\)](#). Presupunem că avem o mulțime de numere întregi și vrem să le „abstractizăm” către o mulțime de semne $+, -, 0$, pentru eficiența stocării.

Fie L mulțimea concretă, adică aceea a numerelor întregi pe care vrem să le abstractizăm și fie L' mulțimea abstractă, pe care vrem să o construim. Cea mai simplă metodă de a face trecerea de la L la L' este să folosim o funcție totală, care ar face corespondența clară. O astfel de funcție, $\alpha : L \rightarrow L'$ se va numi *funcție de abstractizare*. Invers, o funcție $\gamma : L' \rightarrow L$ se va numi *funcție de concretizare*.

Presupunem, în plus, că mulțimea L este și ordonată, o informație pe care vrem să o păstrăm prin abstractizare. De aceea, trebuie să presupunem că funcția α este compatibilă cu ordinea, i.e. este crescătoare, adică $x \leq y$ în L implică $\alpha(x) \leq \alpha(y)$ în L' .

Mai general, putem fi interesați și de corespondența abstract-concret pentru bucăți de program, deci putem dori funcții definite doar pe submulțimi ale lui L .

Astfel, fie L_1, L_2, L'_1, L'_2 mulțimi ordonate și presupunem că semantica concretă a mulțimii L_1 este dată de o funcție $f : L_1 \rightarrow L_2$. O funcție $f' : L'_1 \rightarrow L'_2$ se numește *abstracție validă* a lui f dacă este compatibilă atât cu abstractizarea, cât și cu concretizarea, adică are loc:

$$(f \circ \gamma)(x') \leq (\gamma \circ f')(x'), \quad \forall x' \in L'_1.$$

În funcție de structura pe care o au mulțimile L_1, L_2, L'_1, L'_2 (de exemplu, de latice cu proprietăți suplimentare, cel mai adesea), funcțiile f, f' , dar și elementele care să satisfacă inegalitatea de mai sus pot fi mai greu sau mai ușor de găsit.

Se poate arăta că există un *operator de lărgire* ∇ care poate satisface relația de mai sus, însă cu prețul considerării unei mulțimi mai mare, caz în care se realizează *supra-aproximarea* sau, în ipoteza existenței unei *conexiuni Galois* între mulțimile parțial ordonate luate în discuție, se poate realiza o sub-aproximare.

Revenind la Eva, programul are la bază implementări ale interpretării abstracte, astfel că este folosit pentru a urmări valorile unor variabile. În cazul Monocypher, Eva se va dovedi util din cel puțin două privințe:

- pe de o parte, poate să raporteze dacă există neterminare, caz în care prelucrările numerice din algoritmi sînt greșite, cel puțin pe un caz;
- poate să raporteze dacă valorile numerice folosite ies din marja tipurilor de date folosite (*overflow*), situație care este de asemenea de interes, cu atît mai mult cu cît programul pune accent pe eficiență și viteză, iar algoritmi de criptare folosiți lucrează cu tipuri de date mici.

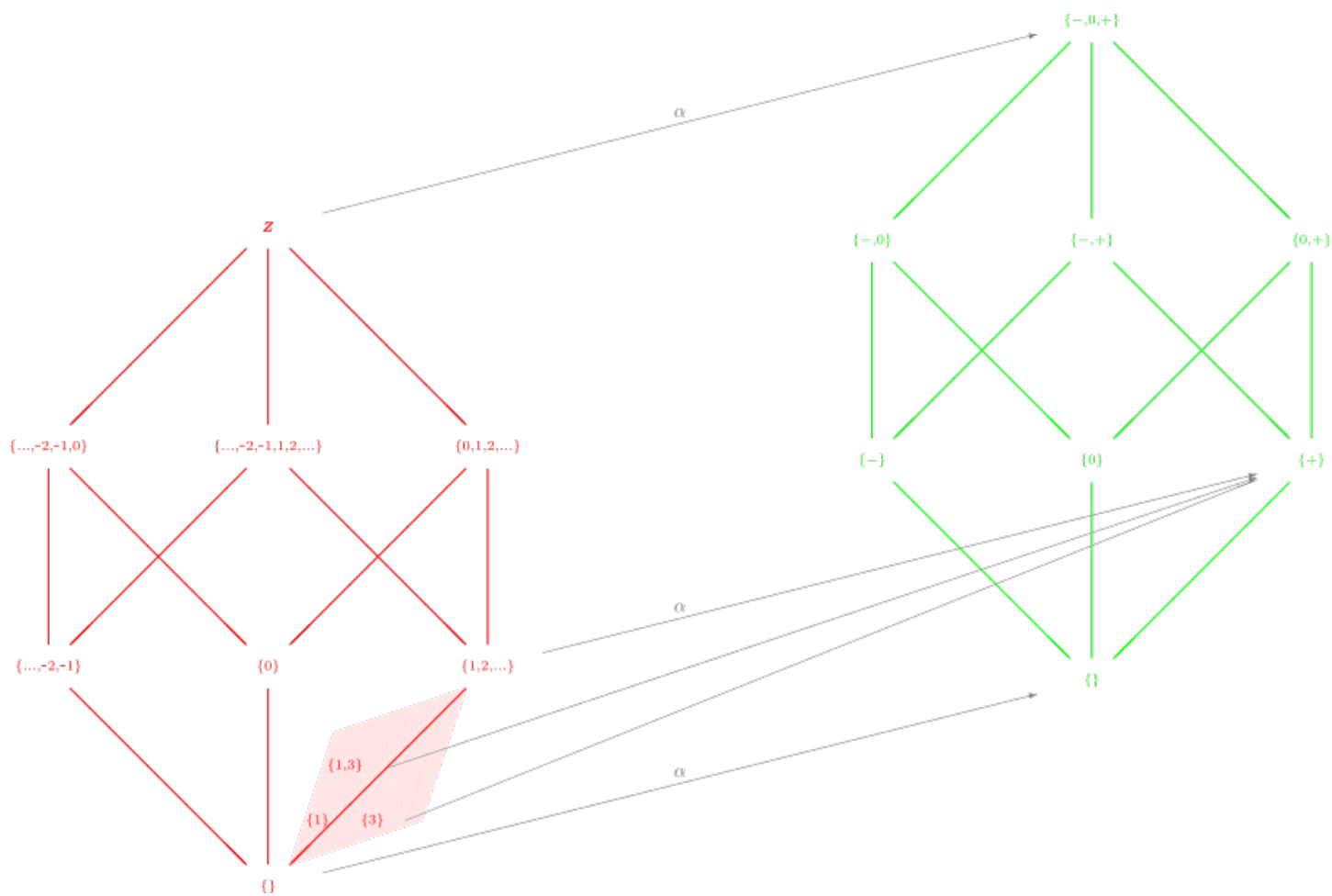


Figura 1.2: *Abstractizarea unei (sub)mulțimi de întregi* (Wikipedia (2019))

2.1 Rularea inițială, fără Eva

Recomandarea generală pentru utilizarea `frama-c`, dar și în cazul particular al acestui proiect este să se ruleze programul din linia de comandă pe toate sursele C ale programului, fără plugin-uri în primă fază, pentru a găsi eventualele erori grosolane. Opțional, se poate salva rezultatul și deschide apoi în GUI sau se poate salva într-un fișier text:

```

1  # rezultate stdout:
2  $ frama-c *.c
3  # rezultate log
4  $ frama-c *.c > log ; cat log | less
5  # rezultate salvate, apoi GUI
6  $ frama-c *.c -save firstrun.sav
7  $ frama-c-gui -load firstrun.sav

```

Codul 2.1: Verificarea inițială a proiectului

În acest caz, se vor raporta:

```

1  [kernel] Parsing monocypher.c (with preprocessing)
2  [kernel] Parsing more_speed.c (with preprocessing)
3  [kernel] more_speed.c:15:
4  syntax error:
5  , before or at token: fe_sq
6  13
7  14      // Specialised squaring function, faster than general multiplication.
8  15      sv fe_sq(fe h, const fe f)
9  ~~~~~

```

```

10 16      {
11 17          i32 f0 = f[0]; i32 f1 = f[1]; i32 f2 = f[2]; i32 f3 = f[3]; i32
      f4 = f[4];
12 [kernel] Frama-C aborted: invalid user input.

```

Codul 2.2: Rezultatele verificării inițiale

Mai multe informații pot fi extrase din acest prim rezultat:

- funcția `sv fe_sq`, care a raportat eroare, poate fi ignorată, deoarece nu face parte neapărat din program. Observăm că acel cod se găsește în `more_speed.c`, care, conform documentației, este un fișier opțional pentru optimizare. Deocamdată, ne concentrăm pe codul principal;
- Verificarea s-a terminat cu eroare, dar nu din cauzele de mai sus. Examinînd fișierul `makefile`, constatăm că programul se compilează cu opțiuni suplimentare, în acest caz, cu opțiunea de preprocesor `-DED25519_SHA512`.

Reîncercăm și specificăm și ca verificarea să se concentreze codul principal:

```

1 $ frama-c test.c sha512.c monocyper.c \
2   -cpp-extra-args="-DED25519_SHA512" -save parsed.sav

```

Codul 2.3: Verificarea inițială, corectată

Execuția nu mai raportează eroare acum și într-adevăr, putem încărca în GUI rezultatul și observăm că avem doar o mică eroare nesemnificativă într-o funcție `printf`.

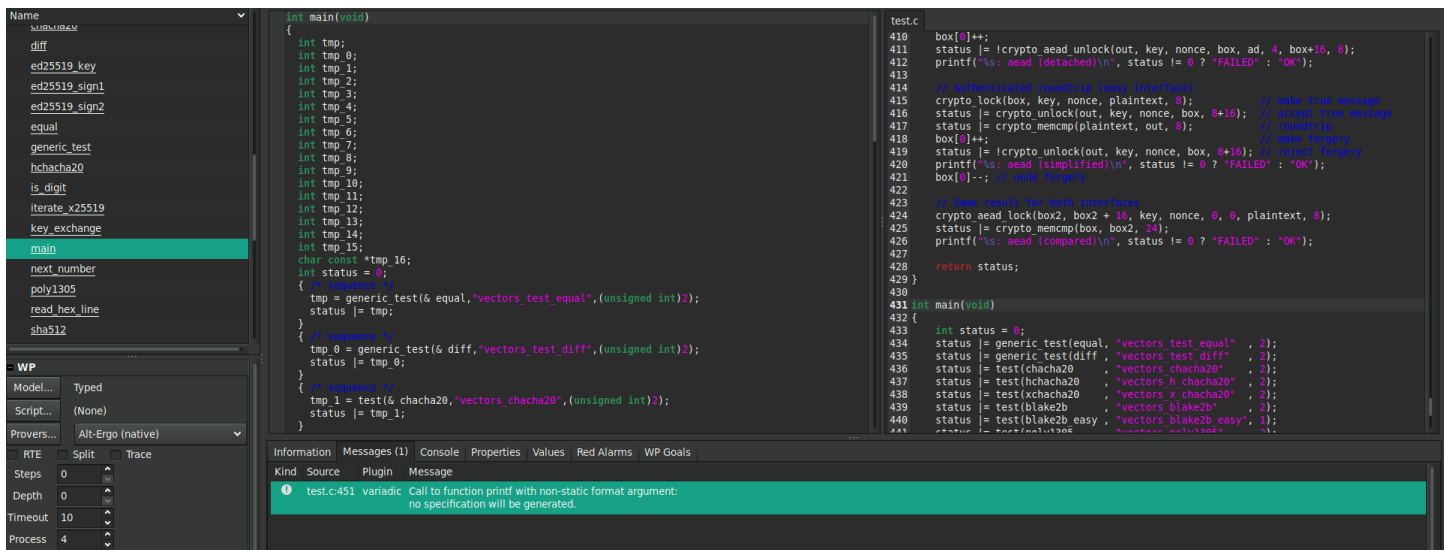


Figura 2.1: Încărcarea în GUI a rezultatelor inițiale

Concluzia acestui pas este că erorile care au mai rămas, dacă este cazul, nu sînt majore ca să împiedice compilarea programului. El poate funcționa pe mai multe cazuri, iar posibilele erori vor fi specifice unor cazuri anume de analiză. Trecem, atunci, la analiza specifică folosind Eva.

2.2 Verificarea cu Eva

Apelul la Eva se face cu opțiunea `-val`, deoarece programul a evoluat din varianta `Value Analysis`.

Rularea inițială face uz doar de câteva funcții standard, care face ca verificarea să fie mai rapidă. Încărcăm varianta salvată din rularea standard și specificăm ca verificarea să adauge funcții de verificare Eva:

```
1 $ frama-c -load parsed.sav -val-builtins-auto -val -save value.sav
```

Codul 2.4: Verificarea inițială cu Eva

Rularea acum durează mai mult și se raportează multe alarme (posibil false), din cauza multor operații numerice folosite în program.

De fapt, rularea de mai sus poate fi optimizată cu două opțiuni specifice:

- `-no-val-show-progress`, care elimină mesajele afișate când se intră într-o nouă funcție (opțiune utilă când sursa are multe funcții mici, ca în acest caz);
- `-memexec-all`, care păstrează o parte a rezultatelor calculate într-un cache din memorie, pentru a nu reface toate calculele.

Așadar, rulăm cu varianta optimizată și salvăm rezultatele atît într-un binar pentru analiză ulterioară în GUI, cît și într-un log.

```
1 $ frama-c -load parsed.sav -val -val-builtins-auto \  
2 -no-val-show-progress -memexec-all -save value.sav > log
```

Codul 2.5: Verificarea optimizată cu Eva

O selecție a liniilor din log afișează:

```
1 [value] Analyzing a complete application starting at main
2 [value] Computing initial state
3 [value] Initial state computed
4 [value:initial-state] Values of globals at initialization
5   __fc_errno in [--.--]
6   __fc_stderr in {{ NULL ; &S__fc_stderr[0] }}
7   __fc_stdin in {{ NULL ; &S__fc_stdin[0] }}
8   __fc_stdout in {{ NULL ; &S__fc_stdout[0] }}
9   __fc_fopen[0..15] in {0}
10  __fc_p_fopen in {{ &__fc_fopen[0] }}
11  __fc_heap_status in [--.--]
12  __fc_random_counter in [--.--]
13  __fc_rand_max in {32767}
14  __fc_mblen_state in [--.--]
15  __fc_mbtowc_state in [--.--]
16  __fc_wctomb_state in [--.--]
17  __fc_strtok_ptr in {0}
18  K[0] in {4794697086780616226}
19     [1] in {8158064640168781261}
20     [2] in {13096744586834688815}
21  ...
22  blake2b_compress_sigma[0][0] in {0}
23                               [0][1] in {1}
24                               [0][2] in {2}
25  ...
26 [value:alarm] test.c:126: Warning:
27   out of bounds write. assert \valid(v->buf + v->size);
28 [value:alarm] test.c:121: Warning:
29   function memcpy: precondition 'valid_src' got status unknown.
30 [value:alarm] monocypher.c:839: Warning:
31   signed overflow. assert g3 * 19 leq 2147483647;
32 [kernel] monocypher.c:494:
33   more than 200(253) locations to update in array. Approximating.
```

Codul 2.6: *Fragment din rezultatele verificării inițiale cu Eva*

Cîteva comentarii esențiale pe marginea acestui raport:

- primele 3 linii arată că se folosește plugin-ul Eva ([value]) și se calculează starea inițială a programului, dată de valorile inițiale ale variabilelor;
- liniile 5-25 arată aceste valori inițiale pentru variabilele din program;

- alarma din linia 26 arată că o valoare a depășit valorile permise și se sugerează o adnotare de tipul `assert` în cod, în `test.c:126` pentru a elimina acest risc;
- similar și în cazul funcției `memcpy` din `test.c:121`, caz în care se sugerează o adnotare de tipul unei precondiții;
- alarma din liniile 30-31 arată că nu calcul este posibil să depășească limitele tipului de dată și se sugerează o adnotare de tipul `assert`;
- mesajul din liniile 32-33 este la nivelul `kernel`, adică nivelul fundamental al verificării și arată că plugin-ul (Eva, în acest caz) are prea multe valori de verificat și actualizat, deci va folosi aproximații care pot da erori (pe lângă cele inerente interpretării abstracte).

Mai remarcăm de asemenea că toate aceste mesaje sînt *alarme*, deci nu se știe cu certitudine dacă sînt erori, putînd fi datorate metodei teoretice propriu-zise. Într-adevăr, verificarea se termină cu succes.

2.3 Rafinarea verificării

Acum, putem rafina căutările în mai multe moduri. De exemplu, variantele mai complicate ar fi:

- să facem toate adnotările sugerate de rezultatele de mai sus, pentru a elimina alarmele;
- să folosim așa-numitele *stubs*, adică să ne asigurăm manual că toate funcțiile sînt definite și apelate în același fișier sau cît mai aproape, pentru a ușura munca prin apeluri între fișiere și între funcții. O tehnică standard este să se creeze header separate sau să se includă manual prototipurile funcțiilor în fișierele unde sînt apelate sau definite.

De asemenea, un alt aspect de care se mai poate ține cont este faptul că, din cauza unor valori ieșite din limitele permise, pot exista bucăți de cod care nu se execută. Acestea sînt raportate în funcții de alarma `unreachable return`. Din fericire, aceasta nu se găsește în codul analizat.

Vom apela, însă, la o tehnică ceva mai simplă, anume vom încărca raportul în GUI, unde putem filtra mesajele, concentrîndu-ne pe cele care ne interesează.

Încărcăm, așadar, binarul `value.sav` din analiza anterioară în `frama-c-gui`.

Din analiza capturii de ecran din figura 2.2, constatăm următoarele:

- avem, în total, 948 de mesaje, vizibile în panoul de jos. Mesajele afișează tipul (eroare, alarmă etc.), fișierul sursă care le conține, plugin-ul care a raportat eroarea și conținutul mesajului de eroare;
- click pe un mesaj de eroare deschide exact locul unde s-a raportat eroarea și observăm că **FramaC** a încercat deja să adauge adnotările sugerate;

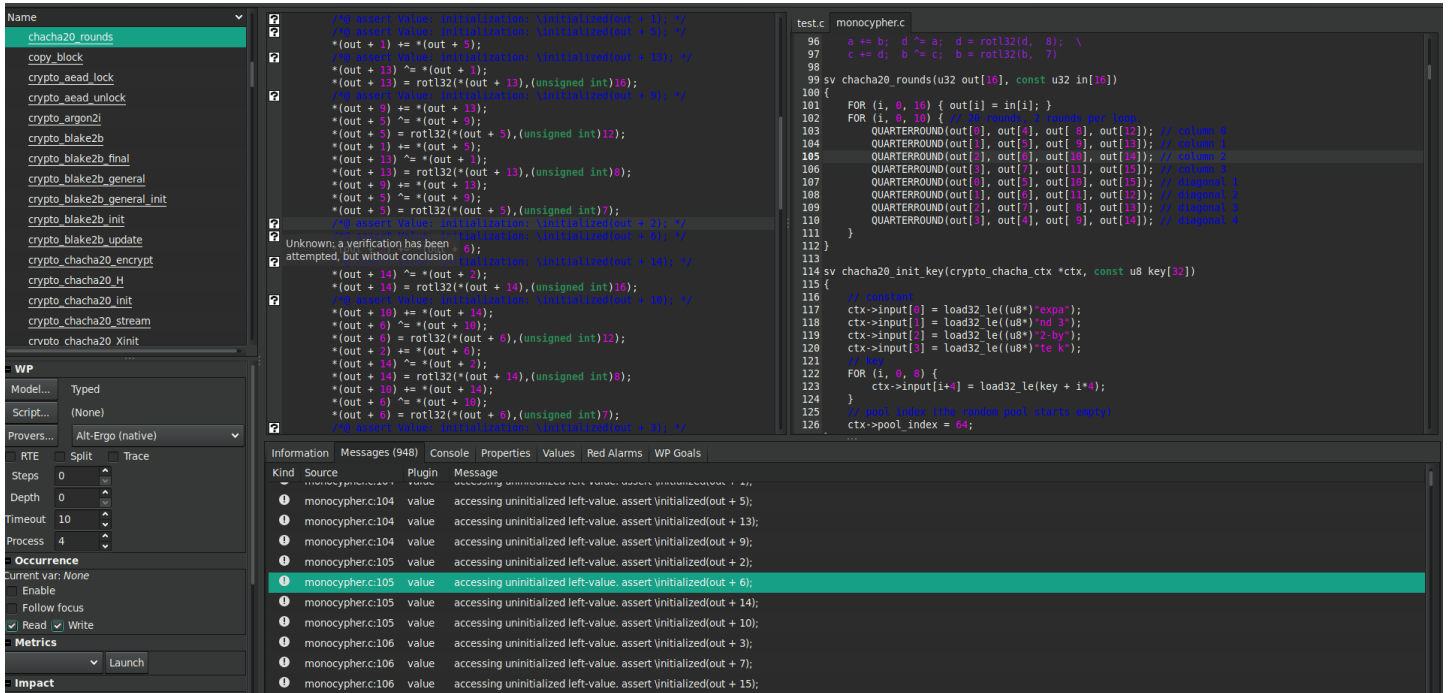


Figura 2.2: Rezultatele r  rii Eva, cu op  iunile de optimizare

-   n banda din st  nga verific  rilor, observ  m codificat   starea verific  rii.   n imagine, se pot vedea multe rezultate de tip `?`, care arat   c   verificarea nu este concludent  : nu se poate spune cu certitudine c     n locul respectiv este o eroare sau o alarm   fals  .

De asemenea,   n panoul de jos mai putem vedea   i o sec  iune   n care putem afi  a valorile salvate   n diverse variabile.

  n exemplul din figura 2.3, putem s   ne concentr  m pe o valoare (inputs) din `test.c:217`,   n acest caz   i, dac   verific  m valorile sale exact   nainte   i dup   apelul func  iei `free`, putem vedea valoarea ini  ial     i cea de dup   dealocare (care devine, desigur, `BOTTOM`). Totodat  , s  ntem siguri   i c   verificarea s-a f  cut cu succes, deoarece   n dreptul acestor instruc  iuni, avem semnul bifat, care expliciteaz     n `Surely valid`.

```

vec_del(& expected);
{
    size_t i_0 = (unsigned int)0;
    while (i_0 < nb_vectors) {
        {
            vec_del(inputs + i_0);
        }
        i_0 += (size_t)1;
    }
}
/* assert Value: signed overflow: nb_tests + 1 = 2147483647; */
nb_tests ++;
}
/* sequence */
;
;
{
    if (status != 0) {
        tmp_4 = "FAILED";
    }
    else {
        tmp_4 = "OK";
    }
}
printf va 2("%s %3d tests: %s\n", (char *)tmp_4, nb_tests, (char *)filename);
free((void *)inputs);
stream_close(& stream);
Surely valid; verified (including all of its dependencies)

```



```

test.c monocypher.c
208     status |= vec_cmp(&output, &expected);
209
210     vec_del(&output);
211     vec_del(&expected);
212     FOR (i, 0, nb_vectors) { vec_del(inputs + i); }
213     nb_tests++;
214 }
215     printf("%s %3d tests: %s\n",
216           status != 0 ? "FAILED" : "OK", nb_tests, filename);
217     free(inputs);
218     stream_close(&stream);
219     return status;
220 }
221
222 //////////////////////////////////////////////////
223 // Test the test suite //
224 //////////////////////////////////////////////////
225 static int equal(const vector v[]) { return vec_cmp(v, v + 1); }
226 static int diff (const vector v[]) { return !vec_cmp(v, v + 1); }
227
228 //////////////////////////////////////////////////
229 // The tests proper //
230 //////////////////////////////////////////////////
231 sv chacha20(const vector in[], vector *out)
232 {
233     const vector *key = in;
234     const vector *nonce = in + 1;
235     crypto_chacha ctx;

```

Information Messages (948) Console Properties Values Red Alarms WP Goals

☒ Multiple selections

Selection  

Callstack	inputs	inputs (after)	(void *)inputs	(void *)inputs (after)
all	{{ (vector *)&_malloc_alloc_122_5; (vector *)&_malloc_alloc_122_11; (vector *)&_malloc_alloc_122_17; (vector *)&_malloc_alloc_122_23; (vector *)&_malloc_alloc_122_29; (vector *)&_malloc_alloc_122_35; (vector *)&_malloc_alloc_122_41; (vector *)&_malloc_alloc_122_48; (vector *)&_malloc_alloc_122_54; (vector *)&_malloc_alloc_122_60; (vector *)&_malloc_alloc_122_66; (vector *)&_malloc_alloc_122_72; (vector *)&_malloc_alloc_122_78 }}	ESCAPINGADDR	{{ (void *)&_malloc_alloc_122_5; (void *)&_malloc_alloc_122_11; (void *)&_malloc_alloc_122_17; (void *)&_malloc_alloc_122_23; (void *)&_malloc_alloc_122_29; (void *)&_malloc_alloc_122_35; (void *)&_malloc_alloc_122_41; (void *)&_malloc_alloc_122_48; (void *)&_malloc_alloc_122_54; (void *)&_malloc_alloc_122_60; (void *)&_malloc_alloc_122_66; (void *)&_malloc_alloc_122_72; (void *)&_malloc_alloc_122_78 }}	BOTTOM
main	{{ (vector *)&_malloc_alloc_122_5 }}	ESCAPINGADDR	{{ (void *)&_malloc_alloc_122_5 }}	BOTTOM
main	{{ (vector *)&_malloc_alloc_122_11 }}	ESCAPINGADDR	{{ (void *)&_malloc_alloc_122_11 }}	BOTTOM
main	{{ (vector *)&_malloc_alloc_122_17 }}	ESCAPINGADDR	{{ (void *)&_malloc_alloc_122_17 }}	BOTTOM
main	{{ (vector *)&_malloc_alloc_122_23 }}	ESCAPINGADDR	{{ (void *)&_malloc_alloc_122_23 }}	BOTTOM
main	{{ (vector *)&_malloc_alloc_122_29 }}	ESCAPINGADDR	{{ (void *)&_malloc_alloc_122_29 }}	BOTTOM

Figura 2.3: Panoul de valori după verificarea Eva

Verificarea programului Monocypher folosind Eva în Framac a evidențiat puterea pe care o are interpretarea abstractă în ce privește analiza statică a valorilor variabilelor dintr-un program complex.

Pentru versiunea concretă a programului studiat, următoarele mențiuni sînt esențiale:

- În versiunile dinaintea 0.3, programul conținea atît scurgeri de memorie, cît și neterminare. Cazul a fost analizat chiar de echipa Framac și expus la [Maroneze \(2017\)](#), dar și de autor, pe pagina personală [LoupVaillant \(2017\)](#). Eroarea este raportată printre mesaje ca în figura 3.1, iar detaliile se pot accesa prin concentrarea pe această eroare (click dreapta, `Focus on this callstack`), ca în imaginea din figura 3.2 (ambele imagini preluate din [Maroneze \(2017\)](#)). Tot la [Maroneze \(2017\)](#) sînt sugerate și idei de analiză a codului care să rezolve această problemă sau măcar să afle cît mai precis cauza.
- În versiunea actuală ([LoupVaillant \(2019a\)](#)), programul nu mai conține bug-uri detectabile de Eva. De fapt, chiar autorul a inclus o serie de testuri și un script care rulează `frama-c` pe toate sursele C ale programului, rezultatul final fiind perfect, fără niciun mesaj de eroare, alarmă sau atenționare.

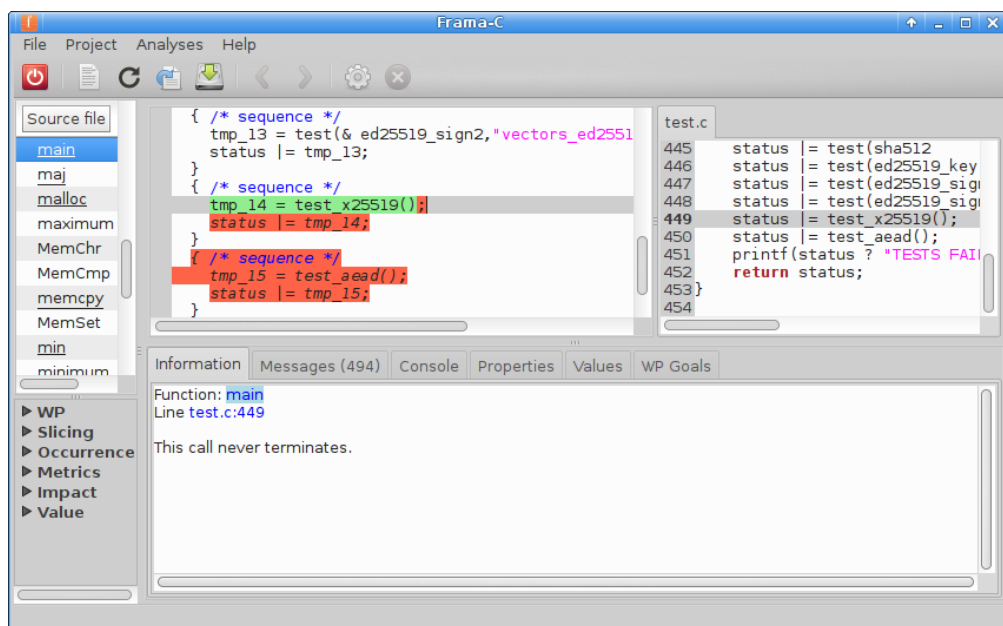


Figura 3.1: Neterminarea din Monocypher<0.3

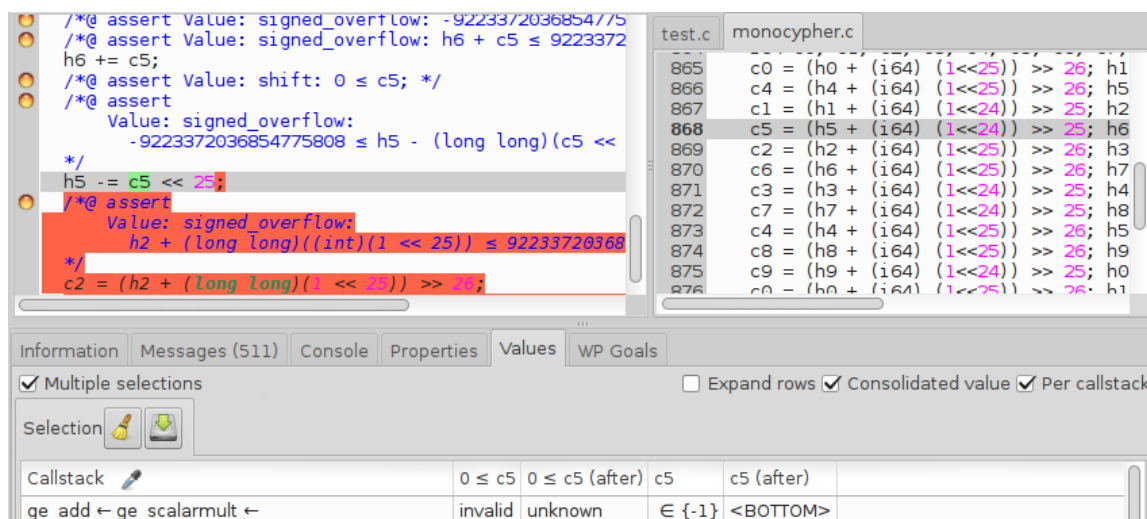


Figura 3.2: Detalii asupra neterminării din 3.1

- Baudin, P. h. (2019). FramaC. [site oficial](#). online, accesat aprilie-mai 2019.
- Cousot, P. (2008). Abstract interpretation. [site oficial](#). online, accesat aprilie-mai 2019.
- FramaC (2016). Emacs acsl-mode. [GitHub](#). online, accesat aprilie-mai 2019.
- FramaC (2019a). ACSL. [site oficial](#). online, accesat aprilie-mai 2019.
- FramaC, G. (2019b). Eva. [manual oficial](#). online, accesat aprilie-mai 2019.
- GNU (2019). Gnu Emacs. [site oficial](#). online, accesat aprilie-mai 2019.
- LoupVaillant (2017). How I Implemented My Own Crypto. [eseu online](#). online, accesat aprilie-mai 2019.
- LoupVaillant (2019a). Monocypher. [GitHub](#). online, accesat aprilie-mai 2019.
- LoupVaillant (2019b). Monocypher. [site oficial](#). online, accesat aprilie-mai 2019.
- Manjaro (2019). Manjaro community edition i3. [site oficial](#). online, accesat aprilie-mai 2019.
- Maroneze, A. (2017). A simple Eva tutorial. [blogul FramuC](#). online, accesat aprilie-mai 2019.
- Smith, L. (2019). st — simple terminal (fork). [GitHub](#). online, accesat aprilie-mai 2019.
- Suckless (2019). st — simple terminal. [site oficial](#). online, accesat aprilie-mai 2019.
- Valgrind, E. (2019). Valgrind. [site oficial](#). online, accesat aprilie-mai 2019.
- Wikipedia (2019). Abstract interpretation. [link](#). online, accesat aprilie-mai 2019.