Proiectul "Ballista" pentru testarea programelor

Mihai Budiu — mailto:mihaib+@cs.cmu.edu
http://www.cs.cmu.edu/~mihaib/

octombrie 2000

Subiect: o metodologie pentru testarea automată a robusteţii software-ului în prezenţa condiţiilor excepţionale

Cunoştinţe necesare: cunoştinţe elementare de programare

Cuvinte cheie: software engineering, excepţii, robusteţe, testare, Ballista, POSIX, Win32

Cuprins

1 Robusteţea programelor

Anul 2000 a fost aşteptat cu înfricoşare de o mulţime de oameni; de data asta însă spaimele milenariste au avut o aromă tehnologică: mai ales societatea industrializată era foarte temătoare în faţa ravagiilor pe care bug-ul Y2K le va aduce. Miezul nopţii a trecut însă fără mare tamtam, şi toate temerile de dinainte ni se par acum ridicole.

Adevărul este însă că, din ce în ce mai mult, viaţa noastră de zi cu zi depinde de calculatoare. Chiar dacă nu vor fi toate vulnerabile dintr-o dată unui virus, malfuncţia multora dintre ele poate afecta în mod dramatic existenţa multora dintre noi, în moduri nebănuite. Reţeaua telefonică digitală, reţelele de televiziune digitală, Internet-ul sunt reţele de calculatoare; reţeaua de distribuţie a energiei electrice, controlul de trafic aerian, armata, şi cine ştie cîte alte părţi ale infrastructurii unei ţări depind din ce în ce mai mult de calculatoare interconectate.

Oricare dintre dumneavoastră care a scris un program de dimensiuni chiar modeste apreciază dificultatea de a face un program să funcţioneze corect în orice circumstanţe. Nici experienţa, nici sculele, nici managementul nu pot produce la ora actuală programe fără defecte; singurul lucru la care putem spera este să avem cît mai puţine defecte, care să fie cît mai insignifiante şi care să se manifeste cît mai rar. De fapt calitatea unui program este direct proporţională cu vechimea sa şi cu frecvenţa de execuţie: cu cît a fost executat de mai multe ori cu date de intrare diferite, cu atît şansele ca eventualele bug-uri să se fi manifestat sunt mai mari. Cele mai insidioase defecte rămîn deci în fragmentele de program rar executate, cum ar fi pe bucăţile de cod care se execută numai în mod excepţional.

Acest articol va prezenta o metodologie de testare automată a unor programe, încercînd să le supună unor torturi care să le oblige să execute cazuri excepţionale. Ideile folosite sunt foarte simple şi naturale; aproape că fiecare din noi ar putea zice: "şi eu puteam face aşa ceva". Dar cu toată simplitatea metodei, rezultatele aplicării ei sunt deosebit de interesante şi totodată îngrijorătoare: practic nici unul dintre sistemele testate, provenind de la mai mult de 10 fabricanţi diferiţi, nu a exhibat o robusteţe foarte bună. Ca să vă ţin treaz interesul, voi amîna discuţia rezultatelor calitative pentru o secţiune ulterioară.

Grupul de cercetare care a dezvoltat această metodologie este condus de profesorul Philip Koopman în cadrul Grupului de sisteme software complexe din Departamentului de inginerie electrică şi calculatoare al Universităţii Carnegie Mellon. Numele proiectului este "Ballista", care înseamnă în engleză acelaşi lucru ca şi în română (mai puţin litera L dublată): catapultă. Numele vine din faptul că sistemul de testat este "bombardat" cu "proiectile" în intenţia de a-i găsi defecţiuni. Acest gen de testare se mai numeşte "injecţie de defecte" (fault injection). Proiectul Ballista (îl voi scrie în continuare cu numele englezesc) încearcă să cuantifice *robusteţea* unui sistem într-un mod concret, pentru a face posibilă compararea unor sisteme diferite care oferă aceeaşi funcţionalitate. "Robusteţea" este definită ca fiind "gradul în care un sistem (software) funcţionează corect în prezenţa unor date excepţionale sau a unui mediu stresant".

2 Interfeţe, modularitate şi paranoia

Vom vedea spre final că proiectul Ballista are foarte multe limitări, dar cu toate acestea utilitatea lui este de necontestat. Ballista poate fi utilizat pentru a testa implementarea *modulelor* software. Modulele software sunt programe (sau biblioteci de funcţii) care oferă servicii altor programe prin interfeţe bine definite.

Ballista a fost aplicat cu succes pentru a testa următoarele componente software: sisteme de operare din familia Unix, sisteme de operare din familia Windows, componente ORB din arhitecturi CORBA, şi biblioteca C standard. În total au fost examinate 15 versiuni majore de Unix, 6 versiuni de Windows şi 9 implementări ale componentei ORB (Object Request Broker) din CORBA.

Toate aceste componente au cîteva trăsături comune: toate oferă funcţionalitate altor programe. Toate vor fi folosite de programe dezvoltate de alte persoane decît cele care au scris modulele. Toate aceste module trebuie să ofere servicii mai multor programe diferite simultan (mai puţin biblioteca C standard, care de obicei este copiată într-un exemplar diferit pentru fiecare program care-i foloseşte funcţiile). Aceste module nu au "încredere" în programul care le cere serviciile; mai ales sistemele de operare şi ORB-ul, pentru că oferă servicii mai multor programe-clienţi simultan, nu trebuie să permită nici unuia dintre ei să fure resurse sau să compromită funcţionarea întregului sistem în detrimentul altor clienţi.

Neîncrederea nucleului sistemului de operare (SO) în cererile programelor care se execută (procesele) trebuie să fie apropiată de paranoia: SO este un program care se execută cu foarte multe privilegii, şi care poate scrie oriunde pe disc, trimite orice pachete în reţea, şi poate face alte astfel acţiuni cu potenţial distructiv. Dacă un proces încearcă să păcălească SO pentru a face acţiuni la care nu are dreptul, SO trebuie să refuze execuţia cererii. SO este singurul care gestionează resursele partajate între toate procesele în execuţie; acestea trebuie să ceară orice acces la resurse prin intermediul unor *apeluri de sistem*[1].

De exemplu, pentru a deschide un fişier, un proces în Unix trebuie să cheme următorul apel de sistem: `open(const char* nume_fisier, int mod)`. `open` are două argumente: numele unui fişier şi modul în care fişierul va fi accesat (citire, scriere, etc.). Dacă procesul trimite în loc de nume o adresă oarecare din memorie (de exemplu un număr negativ), şi dacă nucleul nu verifică faptul că această adresă este ilegală, nucleul ar putea citi zone de memorie care nu există, ceea ce ar putea duce la căderea nucleului şi deci a întregii maşini.

Codul care implementează fiecare apel de sistem începe deci prin a face teste amănunţite asupra argumentelor, verificînd dacă valorile lor sunt corecte şi au sens. După aceea, SO copiază unele din argumente în interiorul nucleului pentru procesare, după care verifică privilegiile procesului (adică dacă procesul are dreptul de a face operaţia cerută), şi abia la sfîrşit, dacă toate preliminariile au mers bine, execută operaţia cerută şi returnează rezultatele. Dacă argumentele sunt ilegale, SO trebuie să semnaleze cumva acest lucru procesului. În sistemele de operare gen Unix de obicei apelul de sistem returnează un rezultat special (un număr negativ), un cod de eroare. În sistemele de tip Windows, nucleul generează o excepţie.

3 Metodologia de testare Ballista

Ballista foloseşte modulele pe care le testează ca pe nişte "cutii negre" (black-box testing), adică nu este interesată de cum sunt construite, ci doar de reacţia vizibilă dinafară cînd primesc felurite intrări. Ballista face un lucru foarte simplu: cere serviciile modulului care este testat folosind la intrare valori care nu au sens. Ballista monitorizează apoi comportarea modulului şi răspunsurile pe care acesta le dă.

[1]Despre rolul nucleului unui sistem de operare, modul în care funcţionează, şi ce sunt apelurile de sistem şi vedeţi şi articolul meu din PC Report din decembrie 1996, disponibil din pagina mea de web.

Modulele bine scrise vor trebui sa raporteze erori (fie prin coduri de eroare ca rezultat, fie prin excepţii). Ballista verifică codul de eroare şi captează toate excepţiile. Problemele mari însă apar cînd modulele *nu* fac ceea ce ar trebui.

Pentru fiecare serviciu care este testat, Ballista genereează cîteva sute sau mii de combinaţii de parametri ilegali. Balllista apoi crează un proces separat, care execută comanda pentru a cere serviciul, folosind parametrii ilegali. Dacă procesul care execută testul se termină, returnează codul primit; dacă nu, după o vreme procesul master trage concluzia că cel de testare a murit. Rezultatele fiecărui test sunt înregistrate pe disc, pentru a preveni pierderea lor în cazul unei catastrofe.

Ballista este folosită pentru a compara între ele sisteme care implementează aceleaşi interfeţe. De exemplu, toate sistemele de tip Unix trebuie să implementeze funcţiile descrise de standardul POSIX[2]. Toate sistemele din familia Windows implementează interfaţa Win32. Astfel putem compara erorile diferitelor sisteme cînd sunt folosite pentru a executa aceleaşi funcţii.

3.1 Tipuri de eşecuri

Ballista foloseşte o clasificare a defecţiunilor într-una din cinci categorii diferite; iniţialele acestor categorii în limba engleză formează acronimul CRASH.

Catastrophic (catastrofe): sunt erorile care corup modulul testat sau cauzează întregul sistem să se blocheze complet sau să se reseteze. Astfel de erori sunt foarte greu de detectat de un program, mai ales dacă se execută pe chiar sistemul afectat. Prezenţa unei astfel de erori este o eroare severă de proiectare a sistemului.

Restart (repornire): aceste erori se petrec cînd serviciul chemat nu returnează vreodată o valoare. Calculatorul continuă să funcţioneze însă; doar serviciul cerut nu se termină. Astfel de erori sunt detectate folosind un ceas de alarmă care după o vreme de la cererea serviciului opreşte Ballista din aşteptare.

Abort (moarte): o astfel de eroare cauzează moartea procesului care cere serviciul, în sisteme Unix de obicei cu un semnal SIGSEGV sau SIGBUS iar în Windows cu o excepţie de gen Application Error.

Silent (silenţioase): astfel de erori constau în lipsa raportării unei erori. Deşi sistemul ar trebui să raporteze o eroare, el spune de fapt că cererea a fost executată cu succes.

Hindering (iritante): sunt cele în care sistemul raportează o eroare, dar nu este cea "reală". Aceste probleme sunt neplăcute, pentru că dacă cel care a cerut serviciul încearcă să remedieze eroarea, informaţia primită de la modul nu-i va fi utilă pentru a ghici cauza reală a problemei. Proiectul Ballista *nu* încearcă să măsoare astfel de eşecuri.

Corect adaug la scara anterioară (CRASH) şi comportarea corectă: sistemul ar trebui să returneze un cod de eroare sau o excepţie care să indice exact de ce serviciul cerut nu se poate executa.

[2]Vedeţi şi articolul meu despre istoria sistemului de operare UNIX din BYTE din august 1996, sau din pagina mea de web.

3.2 Cazuri de test derivate din tipuri de date

Dar cum ştim care valori ale argumentelor unei funcţii sunt legale şi care ilegale? Fiecare funcţie are tot felul de parametri diferiţi, care codifică obiecte felurite. De exemplu, unele funcţii primesc pointeri spre buffere unde datele trebuie depozitate, numere întregi care codifică dimensiunile datelor, numere întregi care codifică descriptori de fişiere deschise, numere ale proceselor pentru a le livra semnale, şi cîte şi mai cîte.

Cea mai elegantă idee din Ballista este folosită pentru a rezolva această problemă: toate tipurile posibile sunt catalogate în cîteva clase (20 de tipuri de date sunt suficiente pentru a testa toate serviciile UNIX pe toate platformele folosite, şi doar unul în plus pentru a testa sistemele din familia Windows).

De exemplu, pentru a testa apelul de sistem Unix, care scrie într-un fişier, trebuie să oferim trei argumente: `write(int fisier, const void* date, size_t marime_date)`.

Deşi primul argument este un număr întreg, de fapt acest număr codifică un descriptor de fişier. Ballista are o listă cu valori excepţionale pentru un descriptor de fişier; iată un fragment tipic:

Argument	Semnificaţie
FD_CLOSED	Fişier care a fost închis
FD_OPEN_READ	Fişier deschis doar pentru citire
FD_DELETED	Fişier care a fost şters între timp
FD_NOEXIST	Un fişier inexistent
FD_EMPTY	Fişier gol
FD_PAST_END	Fişier la care cursorul poziţiei curente este după sfîrşit
FD_TERM	Fişier care este un terminal
FD_MAXINT	Cel mai mare număr întreg
FD_NEG_ONE	Minus unu
etc.	

Pentru a testa funcţia `write`, Ballista generează cele trei argumente luînd o valoare pentru fiecare din tabela corespunzătoare. Dacă numărul total de combinaţii este relativ mic, toate cele posibile sunt încercate. Altfel, 5000 de combinaţii posibile sunt generate aleator.

În Windows cele mai multe apeluri de sistem au mai mult de patru argumente, deci numărul de combinaţii este aproape întotdeauna prea mare.

Pentru testarea CORBA, tipurile de argumente sunt organizate într-o ierarhie de tipuri. De exemplu, tipul "număr întreg" este un subtip al lui "caracter". Astfel tipul "număr întreg" moşteneşte automat toate testele aplicate tipul caracter.

Aceste tabele sunt construite manual de către cei care au implementat sistemul, folosind valori-limită şi experienţa personală.

Dar cum generăm o valoare ca `FD_CLOSED`, care codifică un fişier închis? Fiecare valoare are asociate două proceduri: o procedură care construieşte valoarea şi una care o distruge. Structura unui test Ballista este deci următoarea:

```
test()
{
        initializari();
        valori = alege_valori_parametri();
        parametri = genereaza_parametri(valori);
        eroare = creaza_subproces(executa_test, parametri);
        exceptie = capteaza_exceptii();
        analizeaza_eroare(eroare, exceptie);
        curata(valori);
}
```

De exemplu, pentru a genera un parametru cu valoarea `FD_CLOSED`, care reprezintă un fişier închis, se va invoca o procedură care va crea un fişier, îl va deschide, îl va închide şi va da ca rezultat identificatorul obţinut la deschiderea fişierului.

După ce testul este terminat trebuie făcută "curăţenie": de exemplu trebuie să ştergem fişierul care a fost creat pentru a genera `FD_CLOSED`.

3.3 Detecţia erorilor prin votare

Dacă luăm toate combinaţiile de valori posibile pentru argumente, atunci unele dintre ele vor fi plauzibile (adică nu vor genera o eroare); combinaţiile astea nu ne interesează, pentru că nu stresează de fapt sistemul testat. Dar de unde ştim care combinaţii sunt legale şi care ilegale? E imposibil să anotăm de mînă fiecare combinaţie, pentru a le identifica pe cele legale.

Pentru a rezolva această dilemă ne folosim de faptul că supunem testului mai multe sisteme "echivalente", care oferă aceeaşi funcţionalitate. Dacă nici unul dintre diferitele sisteme pe care le testăm nu generează vreo eroare pentru o anumită combinaţie, declarăm acea combinaţie ca fiind corectă.

O altă problemă este: cum putem detecta erorile "silenţioase", care raportează OK cînd de fapt a apărut o problemă? O altă euristică este folosită aici: presupunem că dacă cel puţin unul dintre sisteme raportează o eroare, atunci combinaţia de argumente este ilegală, şi celelalte sisteme eşuează în mod silenţios.

Din păcate această din urmă metodă s-a dovedit incorectă atunci cînd unele din aceste cazuri au fost examinate manual. Cam 20% din erorile clasificate ca "silenţioase" erau cauzate de malfuncţia unuia dintre sisteme la o combinaţie corectă! De exemplu, sistemul de operare QNX nu permite fişiere al căror nume conţine spaţii, deci toate funcţiile care încercau să opereze cu astfel de fişiere eşuau, deşi standardul POSIX afirmă că acestea sunt perfect legale. O asfel de situaţie duce la clasificarea (în mod eronat) a acestei combinaţii ca fiind o eroare silenţioasă la toate celelalte sisteme, care nu vor raporta o eroare pentru că funcţionează corect.

Erorile "iritante" (hindering) sunt foarte greu de clasificat, mai ales pentru că standardele nu specifică clar ce eroare trebuie returnată în fiecare caz. De exemplu, dacă două argumente sunt eronate, care din erori trebuie raportată? Un alt exemplu este cînd încercăm să scriem 0 octeţi într-un fişier inexistent: dacă testul de lungime este făcut întîi, funcţia va returna succes, pentru că a scris într-adevăr 0 octeţi. Dar dacă întîi se descoperă că fişierul este inexistent, funcţia ar putea returna o eroare. Standardul nu specifică comportarea corectă pentru astfel de cazuri.

4 Rezultate

În fine, ajungem la secţiunea cea mai suculentă: cît de bune sunt feluritele sisteme de operare? Care e mai rezilient? Sunt sistemele folosite în aplicaţii importante mai robuste?

Răspunsurile sunt extrem de surprinzătoare: deşi robusteţea variază de la sistem la sistem, nu există nici unul foarte bun, şi cam toate sunt la fel de vulnerabile. Iată rezultatele concrete pentru fiecare familie de sisteme testate.

4.1 POSIX

Testul sistemelor Unix verifică 233 de funcţii specificate de standardul POSIX (Portable Operating System unIX). Au fost testate 15 sisteme de operare din familia Unix.

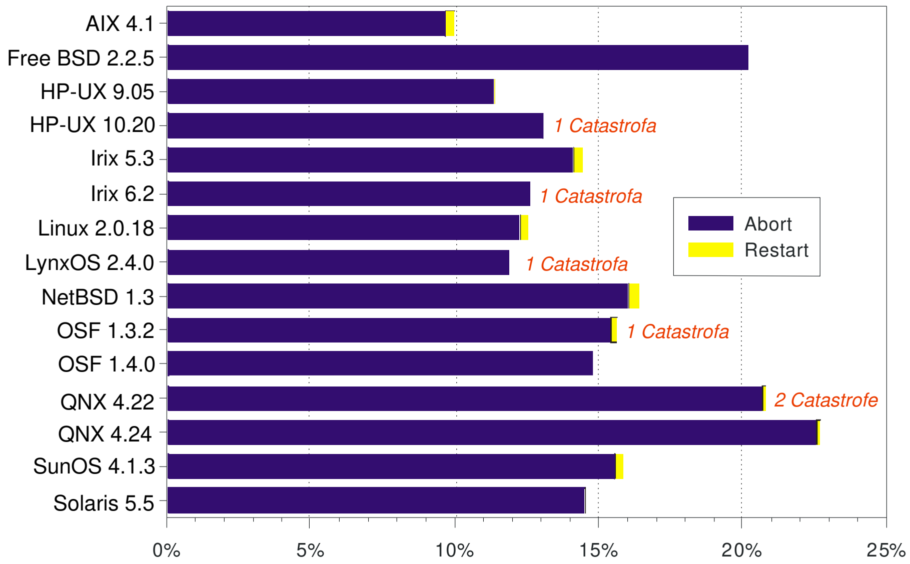

Figura 1: Robusteţea sistemelor POSIX: rata de eroare normalizată pentru 233 de apeluri de sistem. Numai erorile catastrofale, abort şi restart sunt contabilizate.

Figura 1 ilustrează *rata de eroare normalizată* pentru 15 sisteme de operare comerciale din familia Unix, unele fiind versiuni diferite ale aceluiaşi sistem. Fiecare valoare este calculată în felul următor: se numără toate apelurile executate cu cel puţin un parametru ilegal. Apoi se numără cîte din acestea au dus la: distrugerea sistemului (catastrofă), blocarea procesului (restart) şi la moartea procesului (abort). Se face apoi raportul dintre al doilea şi primul număr. Cu cît barele sunt mai scurte, cu atît sistemul e mai bun, adică depistează mai multe erori în mod civilizat.

Tabela următoare indică fabricanţii fiecăruia dintre sisteme; după cum vedeţi o grămadă de lume bună: cele mai respectabile companii de software din lumea Unix sunt toate reprezentate.

Nume	Fabricant
AIX	IBM
FreeBSD	free (bazat pe Berkeley Software Distribution)
HP-UX	Hewlet Packard
Irix	Silicon Graphics
Linux	Linus Torvalds (free)
Lynx	Lynux Works
NetBSD	free (bazat pe Berkeley Software Distribution)
OSF/1	Digital (acum numit Compaq Tru64)
QNX	QNX Software Systems
SunOs	Sun
Solaris	Sun

Putem face imediat o mulţime de observaţii interesante:

- Nici un sistem nu e mai bun de 90%. Cea mai mică bară are 10%, pentru AIX; la cel mai robust dintre sisteme, totuşi 10% din apelurile eronate eşuează.
- Toate numerele se situează între 10 şi 23 de procente, deci sunt comparabile ca mărime. Nici un sistem nu le domină pe celelalte ca fiind foarte bun sau foarte prost.
- Nu întotdeauna versiunea mai nouă a sistemului este mai bună. De exemplu versiunea mai nouă de HP-UX introduce o eroare catastrofică şi creşte numărul de erori care cauzează moartea procesului, ca şi QNX.
- În tabelă avem două sisteme de operare de timp real, care sunt folosite pentru sisteme critice, ca controlul automobilelor, avioanelor, căilor ferate, etc.: QNX şi Lynx. QNX este de departe cel mai prost dintre sisteme! Acesta este un rezultat foarte straniu şi totodată îngrijorător. Pe de altă parte există o explicaţie pentru robusteţea redusă a acestui gen de sisteme: aplicaţiile care se execută pe astfel de sisteme tind să fie mai simple şi scrise cu mult mai mare grijă, aşa încît probabil evită să streseze sistemul. Din cauza asta o mulţime de bug-uri au rămas nedescoperite.
- Linux, chiar o versiune destul de veche, este printre cele mai fiabile sisteme de operare după această metrică.
- Sisteme profesionale, de mare calitate, pot fi complet scoase din funcţiune de programe ale utilizatorilor fără nici un fel de privilegii. Iată două exemple de astfel de programe:
  - Această linie distruge Irix 6.2: `munmap(malloc(1 << 30 + 1), MAXINT);`
  - Această linie distruge Digital OSF/1 versiunea 4.0D: `mprotect(malloc(1 << 29 + 1), 65537, 0);`
- Această tabelă *nu* include eşecuri de tip silenţios sau iritant. Acest lucru este foarte important, pentru că avantajează de pildă sisteme ca AIX: cei care au implementat AIX au decis ca citirea de la adresa 0 din memorie (pointer NULL) să nu genereze nici un fel de eroare, pentru a lăsa o şansă programelor eronate să continue execuţia. Sisteme ca FreeBSD sau Linux acţionează dimpotrivă, anume omoară procesul care încearcă să facă astfel de accese.

Cazul unui pointer 0 va genera o eroare silenţioasă în cazul AIX şi probabil o eroare de tip abort pentru FreeBSD.

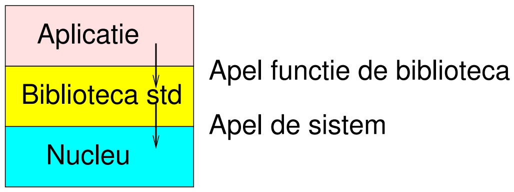

Figura 2: Un apel de sistem trece întîi prin biblioteca standard, care prelucrează argumentele şi abia apoi invocă apelul de sistem propriu-zis al sistemului de operare.

Dacă sunteţi familiar cu modul în care sunt implementate apelurile de sistem, ştiţi că programele scrise de dumneavoastră nu fac direct astfel de apeluri (vedeţi şi figura 2). Ele cheamă nişte funcţii dintr-o bibliotecă "standard", care fac ele însele tot felul de verificări, împachetează argumentele apelului de sistem în regiştri şi apoi execută apelul real.

Figura 3 arată care din erorile din figura 1 pot fi atribuite bibliotecii standard şi care sunt adevărate slăbiciuni ale sistemului de operare. Observaţi că la majoritatea sistemelor cele mai multe slăbiciuni sunt de fapt în bibliotecă (deci sistemul de operare este ceva mai robust decît părea); excepţia notabilă este din nou QNX, la care cele două bare sunt aproximativ egale.

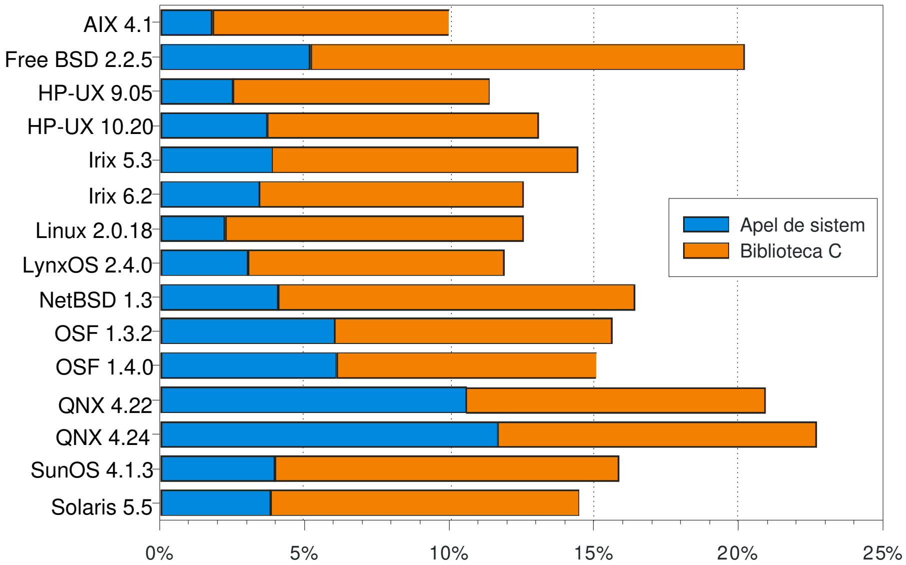

Figura 3: Erori atribuibile bibliotecii C şi apelurilor de sistem (rată de eroare normalizată).

În fine, figura 4 încearcă să califice erorile după tipul de operaţie executată. Rata de eşec este calculată separat pentru fiecare categorie de funcţii.

Şi această figură ne prilejuieşte nişte observaţii interesante:

- Cam toate sistemele au un procent asemănător (ridicat) de erori în biblioteca standard.

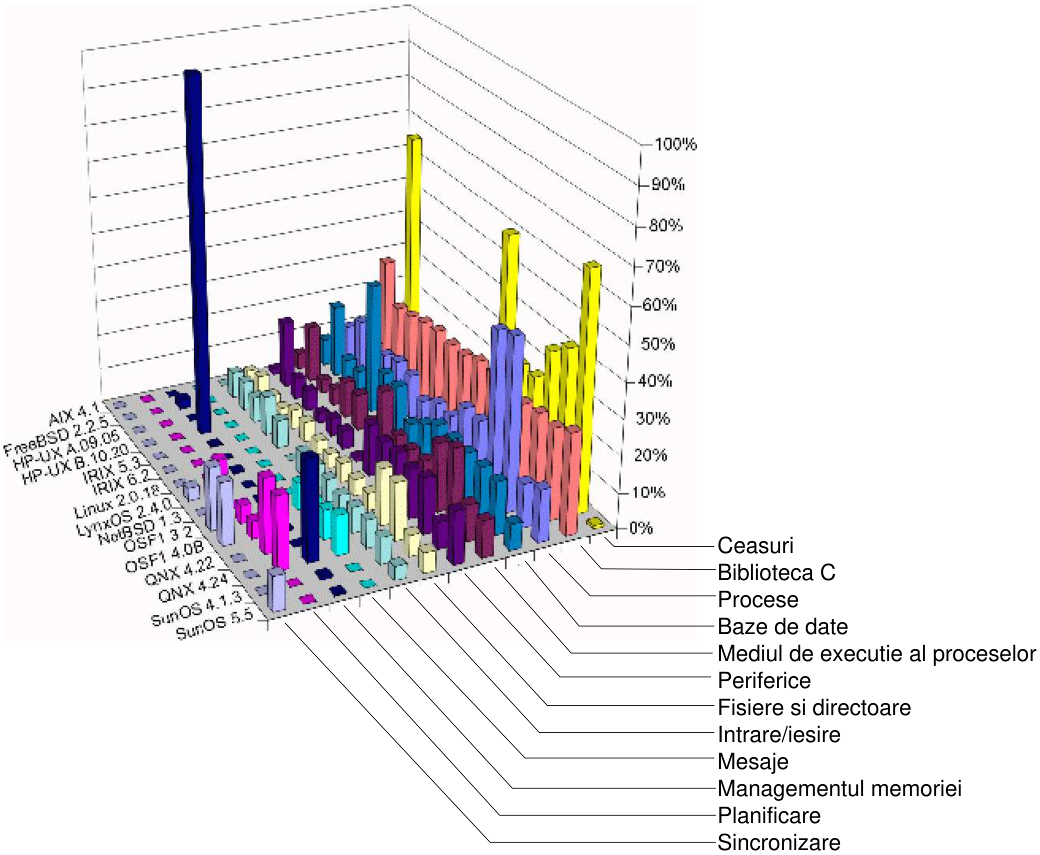

Figura 4: Erori după fiecare tip de apel de sistem pentru sistemele POSIX. (Barele care nu se văd la "ceasuri" sunt foarte mici.)

- QNX are, ca sistem de operare de timp-real, un procent îngrijorător de mare de erori în funcţiile care operează cu ceasuri.
- HP-UX 10 are 100% erori în funcţiile legate de managementul memoriei; e vorba de un subsistem nou, complet reimplementat faţă de HP-UX 9.05.

Figura 5 este construită folosind metodele de "votare" descrise mai sus pentru:

- a elimina cazurile care nu sunt de fapt erori
- a clasifica erorile silenţioase

Acum avem o mşură mai exactă a robusteţii fiecărui sistem; vedem că AIX era într-adevăr avantajat de faptul că transforma unele erori de tip abort în erori silenţioase. Cele mai bune sistem după acest grafic sunt Solaris şi Irix 6.2.

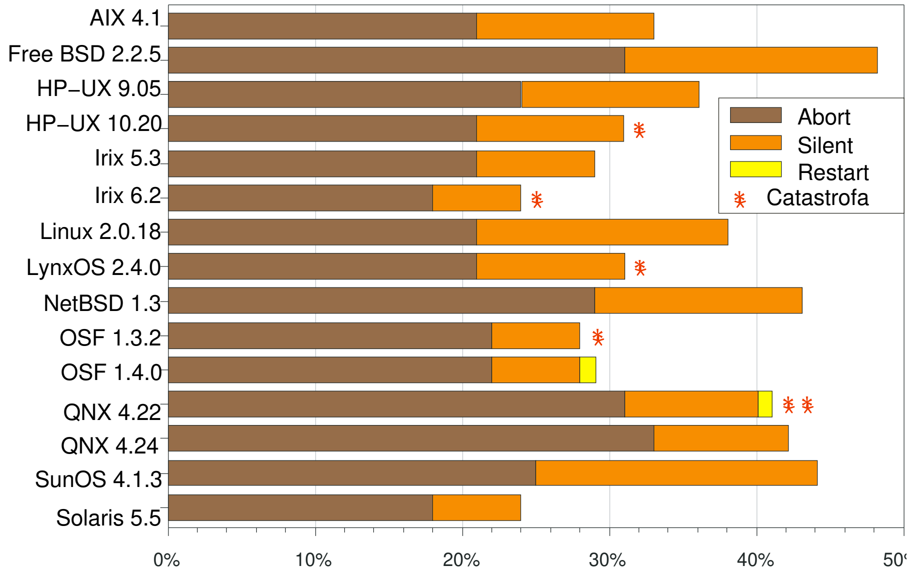

Figura 5: Folosind tehnica votării au fost eliminate combinaţiile de parametri care nu constituie erori. Apoi au fost detectat erorile silenţioase, după cum explicăm în secţiunea despre votare. Pentru că sunt mai puţine cazuri de bază la care ne raportăm, procentele de eroare sunt mai mari ca în figura 1. În plus aici arătăm erorile silenţioase, care se adaugă la cele din acea figură.

4.2 Win32

Să trecem acum la datele privitoare la sistemele firmei Microsoft. Se spune că Windows este un sistem foarte puţin fiabil, care "crapă" regulat, şi care trebuie rebootat zilnic pentru a rămîne "sănătos". Se mai spune apoi că Windows NT şi succesorul lui, Windows 2000 sunt mult mai robuste. De asemenea, lumea afirmă că Linux este mai fiabil. În ce măsură vor confirma datele aceste presupuneri?

În primul rînd, pentru a putea face o comparaţie relativ echitabilă între Windows şi Linux, au fost testate din interfaţa Win32 numai o parte din funcţii, care au echivalente aproximative în sistemele gen POSIX. De pildă, nici una dintre funcţiile care operează cu grafică nu face parte din acest test. Au fost alese 237 de funcţii Win32, care sunt comparate cu 183 de apeluri de sistem Linux. Din cauză că erorile sunt raportate ca procente, diferenţa de număr nu e o problemă.

Iată sistemele comparate:

- Windows 95 revizia B
- Windows 98 cu service pack 1
- Windows 98 ediţia a două, service pack 1 (98/SE)
- Windows NT 4.0 Workstation service pack 5
- Windows 2000 Professional Beta 3 (Build 2031)
- Windows CE 2.11

• RedHat Linux 6.0 (nucleu 2.2.5, altă versiune decît în testele POSIX).

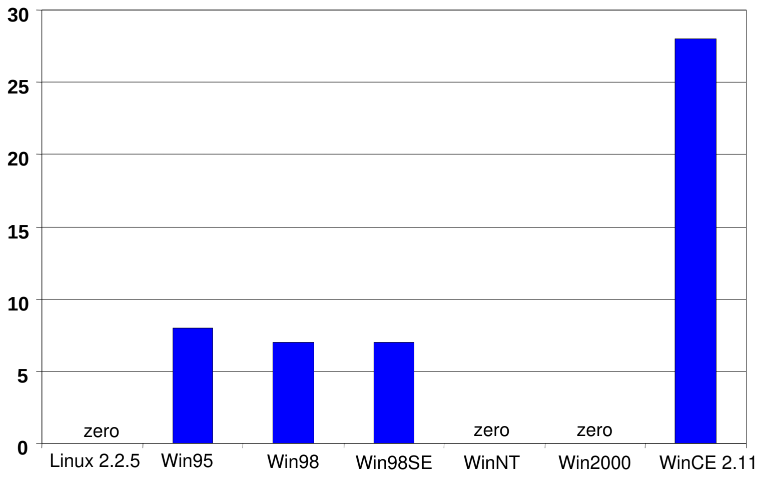

Figura 6: Funcţii cu erori catastrofice în Windows şi Linux.

În figura 6 vedem numărul de funcţii care apelate cu argumente incorecte pot face sistemul să eşueze în mod catastrofic. Într-adevăr, Linux, NT şi Windows 2000 nu pot fi paradite chiar aşa de uşor, dar celelalte sisteme sunt mult mai vulnerabile. Windows CE în particular este foarte vulnerabil.

Figura 7 arată doar apelurile de sistem (excluzînd biblioteca standard), pentru sistemele Windows şi referinţa Linux. Observaţi că aproape întotdeauna Linux este mai robust decît toate celelalte sisteme, cu excepţia managementului proceselor. Surprinzător în această figură este că NT şi Windows 2000 au printre cele mai proaste rezultate (cele mai proaste la 4 categorii).

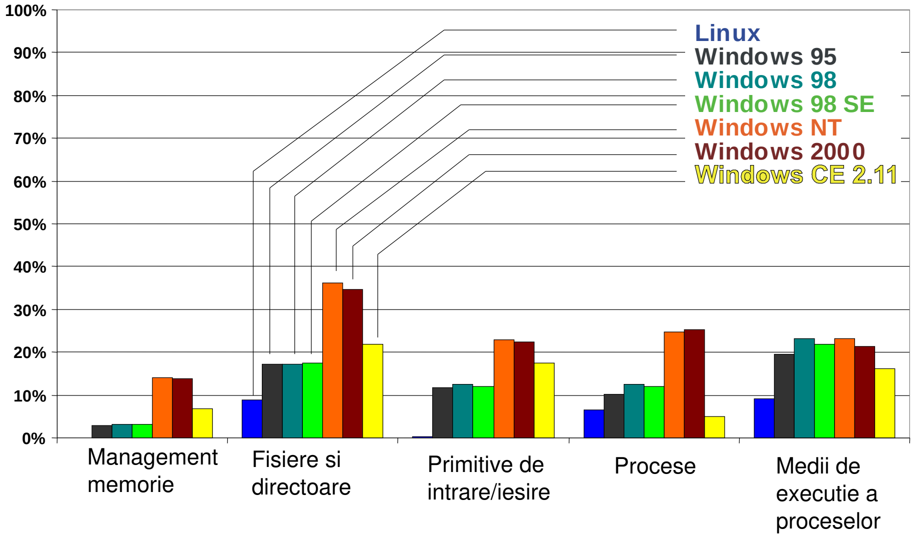

Figura 7: Procente de eşecuri în apeluri de sistem pentru sisteme Windows + Linux.

Dacă ne uităm însă în figura 8 la erorile datorate doar bibliotecii standard, atunci Linux trece cam pe ultimul loc. Nu toate funcţiile de bibliotecă sunt implementate în Windows CE, aşa că X-uri în figură indică absenţa unei categorii, şi nu 0 erori. Nici de data aceasta NT sau Windows 2000 nu se detaşează clar.

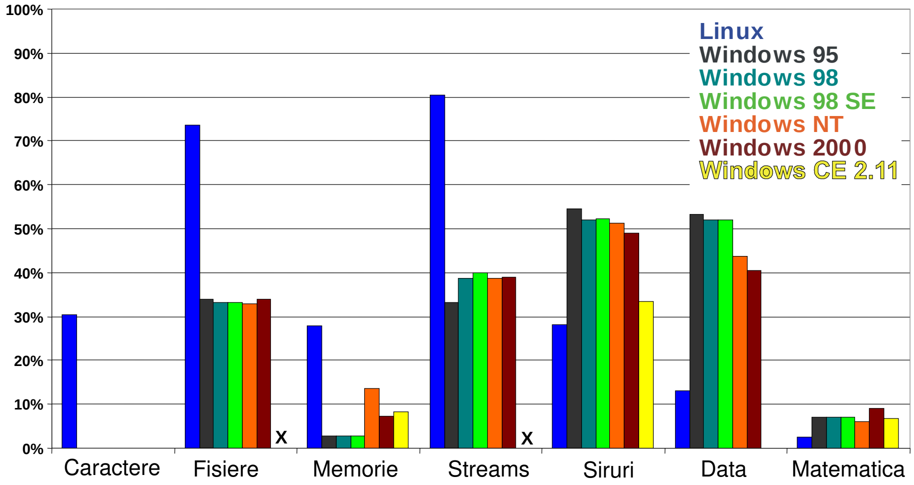

Figura 8: Procente de eşecuri generate de biblioteca C standard. Un "x" înseamnă că acel sistem nu avea astfel de funcţii, deci nu a fost testat.

Ce înseamnă aceste numere? Vom discuta despre limitările măsurătorilor Ballista un pic mai jos, dar se cuvine menţionat că toate aceste măsurători fac teste foarte simple. Faptul că Windows NT nu are nici o eroare catastrofală nu înseamnă că nu se pot întîlni astfel de erori, ci că programe de cîteva linii nu au găsit niciuna.

CORBA: Din cauză că articolul de faţă este deja cam dens, voi sări peste măsurătorile sistemelor CORBA. Demn de spus este că sunt comparabile în calitate cu sistemele de operare prezentate mai sus.

5 Creşterea robusteţii prin împachetare

Defecţiunile găsite sugerează şi o soluţie practică: implementarea unor funcţii care le împachetează pe cele defecte (wrappers). Wrapper-ele sunt scrise cu grijă şi testează toate argumentele înainte de a chema funcţia reală.

Pentru sistemele CORBA a fost chiar elaborată o metodologie semi-automată de protecţie împotriva erorilor: înainte de a trimite o cerere la distanţă, crează un nou fir de execuţie (thread) care trimite cererea. Dacă firul de execuţie moare sau se blochează, programul principal foloseşte o alarmă pentru a continua execuţia şi pentru a raporta o eroare. În felul acesta, erori de implementare în biblioteci sau în modulul chemat nu afectează clientul, care-şi poate continua execuţia.

6 Limitări ale metodologiei Ballista

Am văzut că, deşi bazată pe o idee foarte simplă, Ballista este surprinzător de eficace în a dezgropa o sumedenie de erori în sisteme care sunt extrem de folosite în viaţa de zi cu zi. Dacă reparăm defecţiunile descoperite de Ballista am rezolvat problema corectitudinii programelor?

Nu, deloc. Ballista descoperă numai cele mai simple dintre erori, care sunt cauzate de programe foarte mici. Ballista este neputincioasă în a găsi de pildă erori cauzate de încărcarea mare a sistemului (cînd multă memorie este ocupată sau cînd o mulţime de programe aşteaptă o felie de timp a procesorului).

Ballista nu găseşte erori care depind de *starea internă* a sistemului. De pildă, dacă un program nu dealocă memoria pe care nu o mai foloseşte, spunem că are *scurgeri de memorie* (memory leaks). Scurgerile de memorie nu influenţează corectitudinea comportării programului, dar dacă programul se execută pentru mult timp, acaparează prea multă memorie pentru sine, ceea ce împiedică execuţia celorlalte programe. În momentul în care memoria disponibilă este epuizată se pot produce accidente. Astfel de erori ar fi captate prin repetarea comenzii care cauzează scurgeri, dar Ballista execută fiecare comandă doar o dată cu un set de parametri.

De asemenea, Ballista nu găseşte erori care survin din interacţiunea a multiple componente: fiecare test evaluează o singură funcţie din interfaţă. Dacă apelul a mai multe funcţii într-o anumită ordine ar putea duce la blocare (deadlock), Ballista nu va găsi acest gen de defecţiune.

În fine, cea mai mare limitare a lui Ballista provine din faptul că testează sistemele ca pe nişte cutii negre: nu ştie de fapt care ar trebui să fie răspunsul corect. Ballista se uită doar la condiţiile care trebuie să genereze erori, dar este complet neinteresată de modul în care programul funcţionează cînd poate să-şi facă treaba.

7 Concluzii

Chiar dacă am încheiat cu o listă a limitărilor acestui proiect, nu trebuie să-i neglijăm meritele: am văzut în acest articol o metodologie foarte simplă de a testa robusteţea sistemelor în faţa condiţiilor excepţionale. Atunci cînd testează sisteme care implementează funcţionalităţi asemănătoare, Ballista poate fi folosită pentru a compara cantitativ mai multe sisteme distincte, folosind rata normalizată de eşec.

Pentru că orice sistem de operare trebuie să ofere anumite funcţii elementare, Ballista poate fi folosită chiar pentru a pune faţă-n faţă sisteme atît de diferite ca Windows şi Linux.

Poate cel mai important dintre mesajele acestui text este că software-ul este mereu plin de găuri, şi că aparent este foarte greu să produci soft de o calitate foarte ridicată: toate sistemele testate, de la mai mult de 10 fabricanţi diferiţi, sunt aproximativ la fel de puţin robuste.

8 Alte surse de informaţie

Pagina de web a proiectului Ballista este foarte bine asortată: `http://www.ece.cmu.edu/ballista`. Graficele şi informaţiile au fost extrase din articole şi prezentări din această pagină.

În jurul lunii decembrie proiectul Ballista va face disponibil pe web un sistem de test automat, prin care puteţi genera cod pentru testare de la distanţă; adresa este `http://ballista.ece.cmu.edu/test/`.

Proiectul "Fuzz" de la universitatea Wisconsin a aplicat teste asemănătoare unor utilitare Unix. Două rapoarte ale lor sunt disponibile de la `ftp://grilled.cs.wisc.edu/technical_papers/fuzz.ps.Z` şi `ftp://grilled.cs.wisc.edu/technical_papers/fuzz-revisited.ps.Z`. Acestea arată că programele dezvoltate open-source sunt în general mai robuste decît cele comerciale.

Proiectul "CRASHME" încearcă să "buşească" un sistem de operare generînd cod la întîmplare (folosind numere aleatoare) şi punînd sistemul să-l execute: `http://people.delphi.com/gjc/crashme.html`